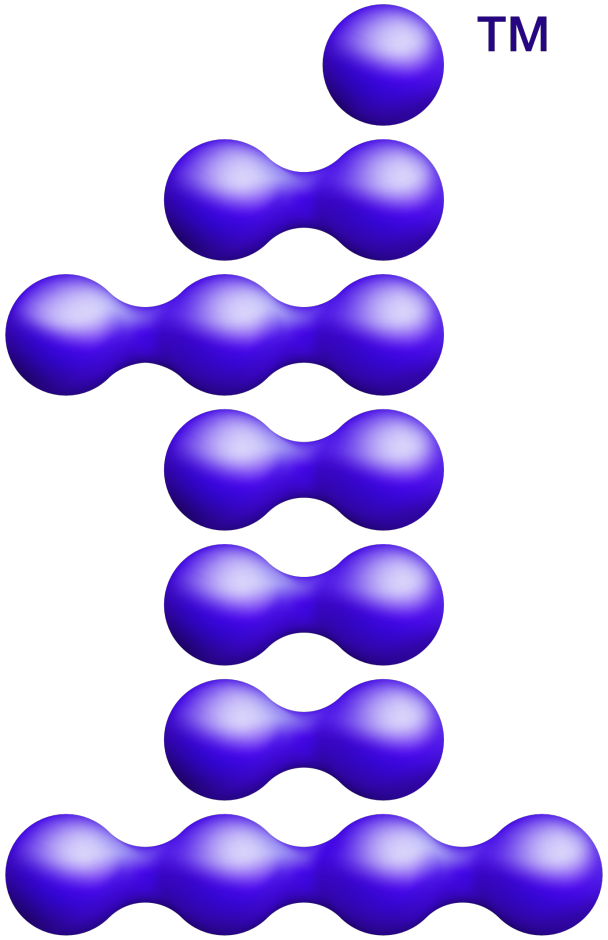


oneAPI

oneAPI Specification

Release 1.3-rev-1

Intel

Aug 22, 2024

CONTENTS

oneAPI is an open, free, and standards-based programming system that provides portability and performance across accelerators and generations of hardware. oneAPI consists of a language and libraries for creating parallel applications:

- *oneDPL*: A companion to the DPC++ Compiler for programming oneAPI devices with APIs from C++ standard library, Parallel STL, and extensions.
- *oneDNN*: High performance implementations of primitives for deep learning frameworks
- *oneCCL*: Communication primitives for scaling deep learning frameworks across multiple devices
- *Level Zero*: System interface for oneAPI languages and libraries
- *oneDAL*: Algorithms for accelerated data science
- *oneTBB*: Library for adding thread-based parallelism to complex applications on multiprocessors
- *oneMKL*: High performance math routines for science, engineering, and financial applications

CHAPTER

ONE

INTRODUCTION

oneAPI simplifies software development by providing the same languages and programming models across accelerator architectures. In this section, we introduce the programming model.

Parallel application development is a combination of *API programming*, where the parallel algorithm is hidden behind an API provided by the system, and *direct programming*, where the application programmer writes the parallel algorithm.

When using API programming, a developer implements performance critical sections of the program with library calls. Well-defined and mature problem domains have high-performance solutions packaged as libraries. oneAPI defines a set of APIs for the most used data parallel domains, and oneAPI platforms provide library implementations across a variety of accelerators. Where possible, the API is based on established standards like BLAS. API programming enables a programmer to attain high performance across a diverse set of accelerators with minimal coding & tuning.

Some problem domains are not well suited to API programming because no standard solution exists or because solutions require a level of customization that cannot be easily implemented in a library. In this case, a developer uses direct programming and must explicitly code the parallel algorithm. oneAPI's programming model is based on data parallelism, where the same computation is performed on each data element, and parallelism of the application scales as the data scales. By allowing the programmer to directly express parallelism, data parallel algorithms make it possible to productively create highly efficient algorithms for parallel architectures.

Data parallel algorithms are used for many of the most computationally demanding problems including scientific computing, artificial intelligence, and visualization. Data parallel algorithms can be efficiently mapped to a diverse set of architectures: multi-core CPUs, GPUs, systolic arrays, and FPGAs.

1.1 Target Audience

The expected audience for this specification includes: application developers, middleware developers, system software providers, and hardware providers. As a *contributor* to this specification, you will shape the accelerator software ecosystem. A productive and high performing system must take into account the constraints at all levels of the software stack. As a *user* of this document, you can ensure that your components will inter-operate with applications and system software for the oneAPI platform.

1.2 Goals of the Specification

oneAPI seeks to provide:

- *Source-level compatibility*: oneAPI applications and middleware port to a conformant oneAPI platform through recompilation and re-tuning.
- *Performance transparency*: API's and language construct allow the programmer enough control over the mapping to hardware to create an efficient solution
- *Software stack portability*: Platform providers can port a oneAPI software stack by implementing the oneAPI Level Zero interface.

1.3 Definitions

This specification uses the definition of must, must not, required, and so on specified in RFC 2119.

1.4 Contribution Guidelines

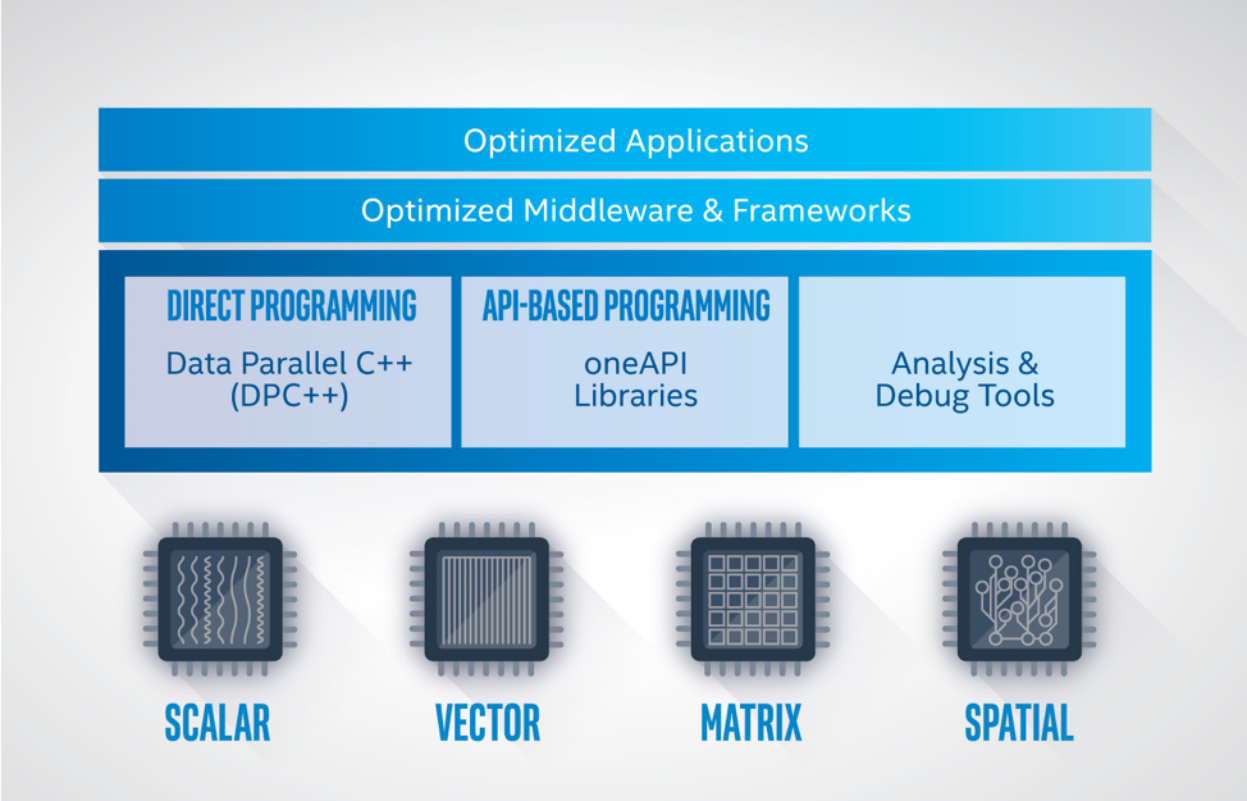
See Contributing.

CHAPTER

TWO

SOFTWARE ARCHITECTURE

oneAPI provides a common developer interface across a range of data parallel accelerators (see the figure below). Programmers use SYCL for both API programming and direct programming. The capabilities of a oneAPI platform are determined by the Level Zero interface, which provides system software a common abstraction for a oneAPI device.



2.1 oneAPI Platform

A oneAPI platform is comprised of a *host* and a collection of *devices*. The host is typically a multi-core CPU, and the devices are one or more GPUs, FPGAs, and other accelerators. The processor serving as the host can also be targeted as a device by the software.

Each device has an associated command *queue*. A application that employs oneAPI runs on the host, following standard C++ execution semantics. To run a *function object* on a device, the application submits a *command group* containing the function object to the device's queue. A function object contains a function definition together with associated variables. A function object submitted to a queue is also referred to as a *data parallel kernel* or simply a *kernel*.

The application running on the host and the functions running on the devices communicate through *memory*. oneAPI defines several mechanisms for sharing memory across the platform, depending on the capabilities of the devices:

Memory Sharing Mechanism	Description
Buffer objects	An application can create *buffer objects* to pass data to devices. A buffer is an array of data. A command group will define *accessor objects* to identify which buffers are accessed in this call to the device. The oneAPI runtime will ensure the data in the buffer is accessible to the function running on the device. The buffer-accessor mechanism is available on all oneAPI platforms
Unified addressing	Unified addressing guarantees that the host and all devices will share a unified address space. Pointer values in the unified address space will always refer to the same location in memory.
Unified shared memory	Unified shared memory enables data to be shared through pointers without using buffers and accessors. There are several levels of support for this feature, depending on the capabilities of the underlying device.

The *scheduler* determines when a command group is run on a device. The following mechanisms are used to determine when a command group is ready to run.

- If the buffer-accessor method is used, the command group is ready when the buffers are defined and copied to the device as necessary.
- If an ordered queue is used for a device, the command group is ready as soon as the prior command groups in the queue are finished.
- If unified shared memory is used, you must specify a set of event objects which the command group depends on, and the command group is ready when all of the events are completed.

The application on the host and the functions on the devices can *synchronize* through *events*, which are objects that can coordinate execution. If the buffer-accessor mechanism is used, the application and device can also synchronize through a *host accessor*, through the destruction of a buffer object, or through other more advanced mechanisms.

2.2 API Programming Example

API programming requires the programmer to specify the target device and the memory communication strategy. In the following example, we call the oneMKL matrix multiply routine, GEMM. We are writing in SYCL and omitting irrelevant details.

We create a queue initialized with a *gpu_selector* to specify that we want the computation performed on a GPU, and we define buffers to hold the arrays allocated on the host. Compared to a standard C++ GEMM call, we add a parameter to specify the queue, and we replace the references to the arrays with references to the buffers that contain the arrays. Otherwise this is the standard GEMM C++ interface.

```
using namespace cl::sycl;

// declare host arrays
double *A = new double[M*N];
double *B = new double[N*P];
double *C = new double[M*P];

{
    // Initializing the devices queue with a gpu_selector
    queue q{gpu_selector()};

    // Creating 1D buffers for matrices which are bound to host arrays
    buffer<double, 1> a{A, range<1>{M*N}};
    buffer<double, 1> b{B, range<1>{N*P}};
    buffer<double, 1> c{C, range<1>{M*P}};

    mkl::transpose nT = mkl::transpose::nontrans;
    // Syntax
    //   void gemm(queue &exec_queue, transpose transa, transpose transb,
    //             int64_t m, int64_t n, int64_t k, T alpha,
    //             buffer<T,1> &a, int64_t lda,
    //             buffer<T,1> &b, int64_t ldb, T beta,
    //             buffer<T,1> &c, int64_t ldc);
    // call gemm
    mkl::blas::gemm(q, nT, nT, M, P, N, 1.0, a, M, b, N, 0.0, c, M);
}
// when we exit the block, the buffer destructor will write result back to C.
```

2.3 Direct Programming Example

With direct programming, we specify the target device and the memory communication strategy, as we do for API programming. In addition, we must define and submit a command group to perform the computation. In the following example, we write a simple data parallel matrix multiply. We are writing in SYCL and omitting irrelevant details.

We create a queue initialized with a *gpu_selector* to specify that the command group should run on the GPU, and we define buffers to hold the arrays allocated on the host. We then submit the command group to the queue to perform the computation. The command group defines accessors to specify we are reading arrays A and B and writing to C. We then write a C++ lambda to create a function object that computes one element of the resulting matrix multiply. We specify this function object as a parameter to a `parallel_for` which maps the function across the arrays `A` and `B` in parallel. When we leave the scope, the destructor for the buffer object holding `C` writes the data back to the host array.

```
#include <CL/sycl.hpp>
using namespace sycl;

int main() {
  // declare host arrays
  double *Ahost = new double[M*N];
  double *Bhost = new double[N*P];
  double *Chost = new double[M*P];

  {
    // Initializing the devices queue with a gpu_selector
    queue q{gpu_selector()};

    // Creating 2D buffers for matrices which are bound to host arrays
    buffer<double, 2> a{Ahost, range<2>{M,N}};
    buffer<double, 2> b{Bhost, range<2>{N,P}};
    buffer<double, 2> c{Chost, range<2>{M,P}};

    // Submitting command group to queue to compute matrix c=a*b
    q.submit([&](handler &h){
                // Read from a and b, write to c
                auto A = a.get_access<access::mode::read>(h);
                auto B = b.get_access<access::mode::read>(h);
                auto C = c.get_access<access::mode::write>(h);

                int WidthA = a.get_range()[1];

                // Executing kernel
                h.parallel_for(range<2>{M, P},
                               [=](id<2> index){
                                 int row = index[0];
                                 int col = index[1];

                                 // Compute the result of one element in c
                                 double sum = 0.0;
                                 for (int i = 0; i < WidthA; i++) {
                                   sum += A[row][i] * B[i][col];
                                 }
                                 C[index] = sum;
                               });
            });
  }
  // when we exit the block, the buffer destructor will write result back to C.
}
```

CHAPTER

THREE

ONEDPL

The oneAPI DPC++ Library (oneDPL) provides the functionality specified in the C++ standard, with extensions to support data parallelism and offloading to devices, and with extensions to simplify its usage for implementing data parallel algorithms.

Note: Unless specified otherwise, in this document the C++ standard refers to ISO/IEC 14882:2017 Programming languages - C++, commonly known as C++17.

The library is comprised of the following components:

- *Parallel API*:
  - Parallel algorithms, complemented with execution policies and companion APIs for running on oneAPI devices.
  - An additional set of library classes and functions that are known to be useful in practice but are not yet included into C++ or SYCL specifications.
- *SYCL Kernels API*:
  - A subset of the C++ standard library which can be used with buffers and data parallel kernels.
  - Support of random number generation including engines and distributions.
  - Various utilities in addition to the C++ standard functionality.

3.1 Namespaces

oneDPL uses `namespace oneapi::dpl` and a shorter variant `namespace dpl` for all functionality including parallel algorithms, oneDPL execution policies, etc. For the subset of the standard C++ library for kernels, the standard class and function names are also aliased in `namespace oneapi::dpl`.

oneDPL uses nested namespaces for the functionality aligned with the C++ standard. The names of those namespaces are the same as in `namespace std`. For example, oneDPL execution policies are provided in `namespace oneapi::dpl::execution`.

3.2 Parallel API

oneDPL provides the set of parallel algorithms as defined by the C++ Standard, including parallel algorithms added in the 6th edition known as C++20. All those algorithms work with *C++ Standard aligned execution policies* and with *device execution policies*.

Additionally, oneDPL provides wrapper functions for SYCL buffers, special iterators, and a set of non-standard parallel algorithms.

3.2.1 Execution Policies

C++ Standard Aligned Execution Policies

oneDPL has the set of execution policies and related utilities that are semantically aligned with the C++ Standard, 6th edition (C++20):

```
// Defined in <oneapi/dpl/execution>

namespace oneapi {
  namespace dpl {
    namespace execution {

      class sequenced_policy { /*unspecified*/ };
      class parallel_policy { /*unspecified*/ };
      class parallel_unsequenced_policy { /*unspecified*/ };
      class unsequenced_policy { /*unspecified*/ };

      inline constexpr sequenced_policy seq { /*unspecified*/ };
      inline constexpr parallel_policy par { /*unspecified*/ };
      inline constexpr parallel_unsequenced_policy par_unseq { /*unspecified*/ };
      inline constexpr unsequenced_policy unseq { /*unspecified*/ };

      template <class T>
      struct is_execution_policy;

      template <class T>
      inline constexpr bool is_execution_policy_v = oneapi::dpl::execution::is_execution_
↪policy<T>::value;
    }
  }
}
```

See "Execution policies" in the C++ Standard for more information.

Device Execution Policy

A device execution policy class `oneapi::dpl::execution::device_policy` specifies the SYCL device and queue to run oneDPL algorithms.

```
// Defined in <oneapi/dpl/execution>

namespace oneapi {
  namespace dpl {
    namespace execution {

      template <typename KernelName = /*unspecified*/>
      class device_policy;

      const device_policy<> dpcpp_default;

      template <typename KernelName = /*unspecified*/>
      device_policy<KernelName>
      make_device_policy( sycl::queue );

      template <typename KernelName = /*unspecified*/>
      device_policy<KernelName>
      make_device_policy( sycl::device );

      template <typename NewKernelName, typename OldKernelName>
      device_policy<NewKernelName>
      make_device_policy( const device_policy<OldKernelName>& = dpcpp_default );
    }
  }
}
```

`dpcpp_default` is a predefined execution policy object to run algorithms on the default SYCL device.

device_policy Class

```
template <typename KernelName = /*unspecified*/>
class device_policy
{
public:
    using kernel_name = KernelName;

    device_policy();
    template <typename OtherName>
    device_policy( const device_policy<OtherName>& );
    explicit device_policy( sycl::queue );
    explicit device_policy( sycl::device );

    sycl::queue queue() const;
    operator sycl::queue() const;
};
```

An object of the `device_policy` type is associated with a `sycl::queue` that is used to run algorithms on a SYCL device. When an algorithm runs with `device_policy` it is capable of processing SYCL buffers (passed via

`oneapi::dpl::begin/end`), data in the host memory and data in Unified Shared Memory (USM), including USM device memory. Data placed in the host memory and USM can only be passed to oneDPL algorithms as pointers and random access iterators. The way to transfer data from the host memory to a device and back is unspecified; per-element data movement to/from a temporary storage is a possible valid implementation.

The `KernelName` template parameter, also aliased as `kernel_name` within the class template, is to explicitly provide a name for SYCL kernels executed by an algorithm the policy is passed to.

```
device_policy()
```

Construct a policy object associated with a queue created with the default device selector.

```
template <typename OtherName>
device_policy( const device_policy<OtherName>& policy )
```

Construct a policy object associated with the same queue as `policy`, by changing the kernel name of the given policy to `kernel_name` defined for the new policy.

```
explicit device_policy( sycl::queue queue )
```

Construct a policy object associated with the given queue.

```
explicit device_policy( sycl::device device )
```

Construct a policy object associated with a queue created for the given device.

```
sycl::queue queue() const
```

Return the queue the policy is associated with.

```
operator sycl::queue() const
```

Allow implicit conversion of the policy to a `sycl::queue` object.

make_device_policy Function

The `make_device_policy` function templates simplify `device_policy` creation.

```
template <typename KernelName = /*unspecified*/>
device_policy<KernelName>
make_device_policy( sycl::queue queue )
```

Return a policy object associated with `queue`, with a kernel name possibly provided as the template argument, otherwise unspecified.

```
template <typename KernelName = /*unspecified*/>
device_policy<KernelName>
make_device_policy( sycl::device device )
```

Return a policy object to run algorithms on `device`, with a kernel name possibly provided as the template argument, otherwise unspecified.

```
template <typename NewKernelName, typename OldKernelName>
device_policy<NewKernelName>
make_device_policy( const device_policy<OldKernelName>& policy = dpcpp_default )
```

Return a policy object constructed from `policy`, with a new kernel name provided as the template argument. If no policy object is provided, the new policy is constructed from `dpcpp_default`.

3.2.2 Buffer Wrappers

```
// Defined in <oneapi/dpl/iterator>

namespace oneapi {
  namespace dpl {

    template < typename T, typename AllocatorT, typename TagT >
    /*unspecified*/ begin( sycl::buffer<T, /*dim=*/1, AllocatorT> buf,
                           TagT tag = sycl::read_write );

    template < typename T, typename AllocatorT, typename TagT >
    /*unspecified*/ begin( sycl::buffer<T, /*dim=*/1, AllocatorT> buf,
                           TagT tag, sycl::property::no_init );

    template < typename T, typename AllocatorT >
    /*unspecified*/ begin( sycl::buffer<T, /*dim=*/1, AllocatorT> buf,
                           sycl::property::no_init );


    template < typename T, typename AllocatorT, typename TagT >
    /*unspecified*/ end( sycl::buffer<T, /*dim=*/1, AllocatorT> buf,
                         TagT tag = sycl::read_write );

    template < typename T, typename AllocatorT, typename TagT >
    /*unspecified*/ end( sycl::buffer<T, /*dim=*/1, AllocatorT> buf,
                         TagT tag, sycl::property::no_init );

    template < typename T, typename AllocatorT >
    /*unspecified*/ end( sycl::buffer<T, /*dim=*/1, AllocatorT> buf,
                         sycl::property::no_init );

  }
}
```

`oneapi::dpl::begin` and `oneapi::dpl::end` are helper functions for passing SYCL buffers to oneDPL algorithms. These functions accept a buffer and return an object of an unspecified type that satisfies the following requirements:

- it is `CopyConstructible`, `CopyAssignable`, and comparable with operators == and `!=`;
- the following expressions are valid: `a + n`, `a - n`, `a - b`, where `a` and `b` are objects of the type, and `n` is an integer value;
- it provides the `get_buffer()` method that returns the buffer passed to the `begin` or `end` function.

When invoking an algorithm, the buffer passed to `begin` should be the same as the buffer passed to `end`. Otherwise, the behavior is undefined.

SYCL deduction tags (the `TagT` parameters) and `sycl::property::no_init` allow to specify an access mode to be used by algorithms for accessing the buffer. The mode serves as a hint, and can be overridden depending on semantics of the algorithm. When invoking an algorithm, the same access mode arguments should be used for `begin` and `end`. Otherwise, the behavior is undefined.

```
using namespace oneapi;
auto buf_begin = dpl::begin(buf, sycl::write_only);
auto buf_end_1 = dpl::end(buf, sycl::write_only);
auto buf_end_2 = dpl::end(buf, sycl::write_only, sycl::no_init);
dpl::fill(dpl::execution::dpcpp_default, buf_begin, buf_end_1, 42); // allowed
dpl::fill(dpl::execution::dpcpp_default, buf_begin, buf_end_2, 42); // not allowed
```

3.2.3 Iterators

The oneDPL iterators are defined in the <oneapi/dpl/iterator> header, in `namespace oneapi::dpl`.

```
template <typename Integral>
class counting_iterator
{
  public:
    using difference_type = /* a signed integer type of the same size as Integral */;
    using value_type = Integral;
    using reference = Integral;

    counting_iterator();
    explicit counting_iterator(Integral init);

    reference operator*() const;
    reference operator[](difference_type i) const;

    difference_type operator-(const counting_iterator& it) const;

    counting_iterator operator+(difference_type forward) const;
    counting_iterator operator-(difference_type backward) const;

    counting_iterator& operator+=(difference_type forward);
    counting_iterator& operator-=(difference_type backward);

    counting_iterator& operator++();
    counting_iterator& operator--();
    counting_iterator& operator++(int);
    counting_iterator& operator--(int);

    bool operator==(const counting_iterator& it) const;
    bool operator!=(const counting_iterator& it) const;
    bool operator<(const counting_iterator& it) const;
    bool operator>(const counting_iterator& it) const;
    bool operator<=(const counting_iterator& it) const;
    bool operator>=(const counting_iterator& it) const;
};
```

`counting_iterator` is a random access iterator-like type that represents an integer counter. When dereferenced, `counting_iterator` provides an Integral rvalue equal to the value of the counter; dereference operations cannot be used to modify the counter. The arithmetic and comparison operators of `counting_iterator` behave as if applied to the values of Integral type representing the counters of the iterator instances passed to the operators.

```
class discard_iterator
{
  public:
    using difference_type = std::ptrdiff_t;
    using value_type = /* unspecified */;
    using reference = /* unspecified */;

    discard_iterator();
    explicit discard_iterator(difference_type init);

    reference operator*() const;
    reference operator[](difference_type) const;

    difference_type operator-(const discard_iterator& it) const;

    discard_iterator operator+(difference_type forward) const;
    discard_iterator operator-(difference_type backward) const;

    discard_iterator& operator+=(difference_type forward);
    discard_iterator& operator-=(difference_type backward);

    discard_iterator& operator++();
    discard_iterator& operator--();
    discard_iterator operator++(int);
    discard_iterator operator--(int);

    bool operator==(const discard_iterator& it) const;
    bool operator!=(const discard_iterator& it) const;
    bool operator<(const discard_iterator& it) const;
    bool operator>(const discard_iterator& it) const;
};
```

`discard_iterator` is a random access iterator-like type that, when dereferenced, provides an lvalue that may be assigned an arbitrary value. The assignment has no effect on the `discard_iterator` instance; the write is discarded. The arithmetic and comparison operators of `discard_iterator` behave as if applied to integer counter values maintained by the iterator instances to determine their position relative to each other.

```
template <typename SourceIterator, typename IndexMap>
class permutation_iterator
{
  public:
    using difference_type =
        typename std::iterator_traits<SourceIterator>::difference_type;
    using value_type = typename std::iterator_traits<SourceIterator>::value_type;
    using pointer = typename std::iterator_traits<SourceIterator>::pointer;
    using reference = typename std::iterator_traits<SourceIterator>::reference;

    permutation_iterator(const SourceIterator& input1, const IndexMap& input2,
                         std::size_t index = 0);

    SourceIterator base() const;

    reference operator*() const;
```

(continues on next page)

(continued from previous page)

```
    reference operator[](difference_type i) const;

    difference_type operator-(const permutation_iterator& it) const;

    permutation_iterator operator+(difference_type forward) const;
    permutation_iterator operator-(difference_type backward) const;

    permutation_iterator& operator+=(difference_type forward);
    permutation_iterator& operator-=(difference_type forward);

    permutation_iterator& operator++();
    permutation_iterator& operator--();
    permutation_iterator operator++(int);
    permutation_iterator operator--(int);

    bool operator==(const permutation_iterator& it) const;
    bool operator!=(const permutation_iterator& it) const;
    bool operator<(const permutation_iterator& it) const;
    bool operator>(const permutation_iterator& it) const;
    bool operator<=(const permutation_iterator& it) const;
    bool operator>=(const permutation_iterator& it) const;
};
```

`permutation_iterator` is a random access iterator-like type whose dereferenced value set is defined by the source iterator provided, and whose iteration order over the dereferenced value set is defined by either another iterator or a functor that maps the `permutation_iterator` index to the index of the source iterator. The arithmetic and comparison operators of `permutation_iterator` behave as if applied to integer counter values maintained by the iterator instances to determine their position in the index map.

`permutation_iterator::operator*` uses the counter value of the instance on which it is invoked to index into the index map. The corresponding value in the map is then used to index into the value set defined by the source iterator. The resulting lvalue is returned as the result of the operator.

`permutation_iterator::operator[]` uses the parameter `i` to index into the index map. The corresponding value in the map is then used to index into the value set defined by the source iterator. The resulting lvalue is returned as the result of the operator.

```
template <typename SourceIterator, typename IndexMap>
permutation_iterator<SourceIterator, IndexMap>
make_permutation_iterator(SourceIterator source, IndexMap map);
```

`make_permutation_iterator` constructs and returns an instance of `permutation_iterator` using the source iterator and index map provided.

```
template <typename Iterator, typename UnaryFunc>
class transform_iterator
{
  public:
    using difference_type = typename std::iterator_traits<Iterator>::difference_type;
    using reference = typename std::invoke_result<UnaryFunc,
                          typename std::iterator_traits<Iterator>::reference>::type;
    using value_type = typename std::remove_reference<reference>::type;
    using pointer = typename std::iterator_traits<Iterator>::pointer;
```

(continues on next page)

(continued from previous page)

```
    Iterator base() const;

    transform_iterator(Iterator it, UnaryFunc unary_func);
    transform_iterator(const transform_iterator& input);
    transform_iterator& operator=(const transform_iterator& input);

    reference operator*() const;
    reference operator[](difference_type i) const;

    difference_type operator-(const transform_iterator& it) const

    transform_iterator operator+(difference_type forward) const;
    transform_iterator operator-(difference_type backward) const;

    transform_iterator& operator+=(difference_type forward);
    transform_iterator& operator-=(difference_type backward);

    transform_iterator& operator++();
    transform_iterator& operator--();
    transform_iterator operator++(int);
    transform_iterator operator--(int);

    bool operator==(const transform_iterator& it) const;
    bool operator!=(const transform_iterator& it) const;
    bool operator<(const transform_iterator& it) const;
    bool operator>(const transform_iterator& it) const;
    bool operator<=(const transform_iterator& it) const;
    bool operator>=(const transform_iterator& it) const;
};
```

`transform_iterator` is a random access iterator-like type whose dereferenced value set is defined by the unary function and source iterator provided. When dereferenced, `transform_iterator` provides the result of the unary function applied to the corresponding element of the source iterator; dereference operations cannot be used to modify the elements of the source iterator unless the unary function result includes a reference to the element. The arithmetic and comparison operators of `transform_iterator` behave as if applied to the source iterator itself.

```
template <typename UnaryFunc, typename Iterator>
transform_iterator<UnaryFunc, Iterator>
make_transform_iterator(Iterator, UnaryFunc);
```

`make_transform_iterator` constructs and returns an instance of `transform_iterator` using the source iterator and unary function object provided.

```
template <typename... Iterators>
class zip_iterator
{
  public:
    using difference_type = typename std::make_signed<std::size_t>::type;
    using value_type =
        std::tuple<typename std::iterator_traits<Iterators>::value_type...>;
    using reference = /* unspecified tuple of reference types */;
```

(continues on next page)

(continued from previous page)

```
    using pointer =
        std::tuple<typename std::iterator_traits<Iterators>::pointer...>;

    std::tuple<Iterators...> base() const;

    zip_iterator();
    explicit zip_iterator(Iterators... args);
    zip_iterator(const zip_iterator& input);
    zip_iterator& operator=(const zip_iterator& input);

    reference operator*() const;
    reference operator[](difference_type i) const;

    difference_type operator-(const zip_iterator& it) const;
    zip_iterator operator-(difference_type backward) const;
    zip_iterator operator+(difference_type forward) const;

    zip_iterator& operator+=(difference_type forward);
    zip_iterator& operator-=(difference_type backward);

    zip_iterator& operator++();
    zip_iterator& operator--();
    zip_iterator operator++(int);
    zip_iterator operator--(int);

    bool operator==(const zip_iterator& it) const;
    bool operator!=(const zip_iterator& it) const;
    bool operator<(const zip_iterator& it) const;
    bool operator>(const zip_iterator& it) const;
    bool operator<=(const zip_iterator& it) const;
    bool operator>=(const zip_iterator& it) const;
};
```

`zip_iterator` is an iterator-like type defined over one or more iterators. When dereferenced, the value returned from `zip_iterator` is a tuple of the values returned by dereferencing the source iterators over which the `zip_iterator` is defined. The arithmetic operators of `zip_iterator` update the source iterators of a `zip_iterator` instance as though the operation were applied to each of these iterators.

```
template <typename... Iterators>
zip_iterator<Iterators...>
make_zip_iterator(Iterators...);
```

`make_zip_iterator` constructs and returns an instance of `zip_iterator` using the set of source iterators provided.

3.2.4 Parallel Algorithms

The parallel algorithms are defined in the <oneapi/dpl/algorithm> header, in `namespace oneapi::dpl`.

```
template<typename Policy, typename InputKeyIt, typename InputValueIt,
    typename OutputValueIt,
    typename T = typename std::iterator_traits<InputValueIt>::value_type,
    typename BinaryPred =
        std::equal_to<typename std::iterator_traits<InputKeyIt>::value_type>,
    typename BinaryOp =
        std::plus<typename std::iterator_traits<InputValueIt>::value_type>>
OutputValueIt
exclusive_scan_by_segment(Policy&& policy, InputKeyIt keys_first,
    InputKeyIt keys_last, InputValueIt values_first, OutputValueIt values_result,
    T initial_value = 0,
    BinaryPred binary_pred =
        std::equal_to<typename std::iterator_traits<InputKeyIt>::value_type>(),
    BinaryOp binary_op =
        std::plus<typename std::iterator_traits<InputValueIt>::value_type>());
```

`oneapi::dpl::exclusive_scan_by_segment` performs partial prefix scans by applying the `binary_op` operation to a sequence of values. Each partial scan applies to a contiguous subsequence determined by the keys associated with the values being equal according to the `binary_pred` predicate, and the first element of each scan is the initial value provided. The return value is an iterator targeting the end of the result sequence.

The initial value used if one is not provided is an instance of the `value_type` of the `InputValueIt` iterator type initialized to 0. If no binary predicate is provided for the comparison of keys an instance of `std::equal_to` with the `value_type` of the `InputKeyIt` iterator type is used. Finally, an instance of `std::plus` with the `value_type` of the `InputValueIt` iterator type is used if no binary operator is provided to combine the elements of the value subsequences.

```
template<typename Policy, typename InputKeyIt, typename InputValueIt,
    typename OutputValueIt,
    typename BinaryPredcate =
        std::equal_to<typename std::iterator_traits<InputKeyIt>::value_type,
    typename BinaryOp =
        std::plus<typename std::iterator_traits<InputValueIt>::value_type>>
OutputValueIt
inclusive_scan_by_segment(Policy&& policy, InputKeyIt keys_first,
    InputKeyIt keys_last, InputValueIt values_first, OutputValueIt values_result
    BinaryPred binary_pred =
        std::equal_to<typename std::iterator_traits<InputKeyIt>::value_type>(),
    BinaryOp binary_op =
        std::plus<typename std::iterator_traits<InputValueIt>::value_type>());
```

`oneapi::dpl::inclusive_scan_by_segment` performs partial prefix scans by applying the `binary_op` operation to a sequence of values. Each partial scan applies to a contiguous subsequence determined by the keys associated with the values being equal according to the `binary_pred` predicate. The return value is an iterator targeting the end of the result sequence.

If no binary predicate is provided for the comparison of keys an instance of `std::equal_to` with the `value_type` of the `InputKeyIt` iterator type is used. An instance of `std::plus` with the `value_type` of the `InputValueIt` iterator type is used if no binary operator is provided to combine the elements of the value subsequences.

```
template<typename Policy, typename InputKeyIt, typename InputValueIt,
    typename OutputKeyIt, typename OutputValueIt,
    typename BinaryPredcate =
        std::equal_to<typename std::iterator_traits<InputKeyIt>::value_type,
    typename BinaryOp =
        std::plus<typename std::iterator_traits<InputValueIt>::value_type>>
std::pair<OutputKeyIt,OutputValueIt>
reduce_by_segment(Policy&& policy, InputKeyIt keys_first, InputKeyIt keys_last,
    InputValueIt values_first, OutputKeyIt keys_result,
    OutputValueIt values_result,
    BinaryPred binary_pred =
        std::equal_to<typename std::iterator_traits<InputKeyIt>::value_type>(),
    BinaryOp binary_op =
        std::plus<typename std::iterator_traits<InputValueIt>::value_type>());
```

`oneapi::dpl::reduce_by_segment` performs partial reductions on a sequence of values. Each reduction is computed with the `binary_op` operation for a contiguous subsequence of values determined by the associated keys being equal according to the `binary_pred` predicate. For each subsequence the first of the equal keys is stored into `keys_result` and the computed reduction is stored into `values_result`. The return value is a pair of iterators holding the end of the resulting sequences.

If no binary predicate is provided for the comparison of keys an instance of `std::equal_to` with the `value_type` of the `InputKeyIt` iterator type is used. An instance of `std::plus` with the `value_type` of the `InputValueIt` iterator type is used to combine the values in each subsequence identified if a binary operator is not provided.

```
template<typename Policy, typename InputIt1, typename InputIt2, typename OutputIt,
    typename Comparator =
        std::less<typename std::iterator_traits<InputIt>::value_type>>
OutputIt
binary_search(Policy&& policy, InputIt1 start, InputIt1 end,
    InputIt2 value_first, InputIt2 value_last, OutputIterator result,
    Comparator comp =
        std::less<typename std::iterator_traits<InputIt1>::value_type>());
```

`oneapi::dpl::binary_search` performs a binary search over the data in `[start, end)` for each value in `[value_first, value_last)`. If the value exists in the data searched then the corresponding element in `[result, result + distance(value_first, value_last))` is set to true, otherwise it is set to false.

If no comparator is provided, `operator<` is used to determine when the search value is less than an element in the range being searched.

The elements of `[start, end)` must be partitioned with respect to the comparator used. For all elements `e` in `[start, end)` and a given search value `v` in `[value_first, value_last)`, `comp(e, v)` implies `!comp(v, e)`.

```
template<typename Policy, typename InputIt1, typename InputIt2, typename OutputIt,
    typename Comparator =
        std::less<typename std::iterator_traits<InputIt>::value_type>>
OutputIt
lower_bound(Policy&& policy, InputIt1 start, InputIt1 end,
    InputIt2 value_first, InputIt2 value_last, OutputIterator result,
    Comparator comp =
        std::less<typename std::iterator_traits<InputIt1>::value_type>());
```

`oneapi::dpl::lower_bound` performs a binary search over the data in `[start, end)` for each value in `[value_first, value_last)` to find the lowest index at which the search value could be inserted in `[start,`

end) without violating the ordering defined by the comparator provided. That lowest index is then assigned to the corresponding element in [result, result + distance(value_first, value_last)).

If no comparator is provided, operator< is used to determine when the search value is less than an element in the range being searched.

The elements of [start, end) must be partitioned with respect to the comparator used.

```
template<typename Policy, typename InputIt1, typename InputIt2, typename OutputIt,
    typename Comparator =
        std::less<typename std::iterator_traits<InputIt>::value_type>>
OutputIt
upper_bound(Policy&& policy, InputIt1 start, InputIt1 end,
    InputIt2 value_first, InputIt2 value_last, OutputIterator result,
    Comparator comp =
        std::less<typename std::iterator_traits<InputIt1>::value_type>());
```

oneapi::dpl::upper_bound performs a binary search over the data in [start, end) for each value in [value_first, value_last) to find the highest index at which the search value could be inserted in [start, end) without violating the ordering defined by the comparator provided. That highest index is then assigned to the corresponding element in [result, result + distance(value_first, value_last)).

If no comparator is provided, operator< is used to determine when the search value is less than an element in the range being searched.

The elements of [start, end) must be partitioned with respect to the comparator used.

3.3 SYCL Kernels API

3.3.1 Supported C++ Standard Library APIs and Algorithms

oneDPL defines a subset of the C++ Standard library APIs for use in SYCL kernels. These APIs can be employed in the kernels similarly to how they are employed in code for a typical CPU-based platform.

3.3.2 Random Number Generation

oneDPL provides a subset of the standard C++ pseudo-random number generation functionality suitable to use within SYCL kernels. The APIs are defined in the <oneapi/dpl/random> header.

Supported Functionality

- **Engine class templates:**
  - linear_congruential_engine
  - subtract_with_carry_engine
- **Engine adaptor class templates:**
  - discard_block_engine
- **Engines and engine adaptors with predefined parameters:**
  - minstd_rand0
  - minstd_rand

  - `ranlux24_base`
  - `ranlux48_base`
  - `ranlux24`
  - `ranlux48`
- **Distribution class templates:**
  - `uniform_int_distribution`
  - `uniform_real_distribution`
  - `normal_distribution`
  - `exponential_distribution`
  - `bernoulli_distribution`
  - `geometric_distribution`
  - `weibull_distribuion`
  - `lognormal_distribution`
  - `cauchy_distribution`
  - `extreme_value_distribution`

`linear_congruential_engine` and `subtract_with_carry_engine` satisfy the uniform random bit generator requirements.

Limitations

The following deviations from the C++ Standard may apply:

- `random_device` and `seed_seq` classes and related APIs in other classes are not required;
- specifying the size of a random number engine's state is not required;
- distributions are only required to operate with floating point types applicable to supported SYCL devices.

Extensions

As an extension to the C++ Standard, `sycl::vec<Type, N>` can be used as the data type template parameter for engines, engine adaptors, and distributions, where `Type` is one of data types supported by the corresponding class template in the standard. For such template instantiations, the `result_type` is also defined to `sycl::vec<Type, N>`.

Engines, engine adaptors, and distributions additionally define `scalar_type`, equivalent to the following:

- `using scalar_type = typename result_type::element_type;` if `result_type` is `sycl::vec<Type, N>`,
- otherwise, `using scalar_type = result_type;`

The `scalar_type` is used instead of `result_type` in all contexts where a scalar data type is expected, including

- the type of configuration parameters and properties,
- the seed value type,
- the input parameters of constructors,
- the return value type of `min()` and `max()` methods, etc.

Since `scalar_type` is the same as `result_type` except for template instantiations with `sycl::vec`, class templates still meet the applicable requirements of the C++ Standard.

When instantiated with `sycl::vec<Type,N>`, `linear_congruential_engine` and `subtract_with_carry_engine` may not formally satisfy the uniform random bit generator requirements defined by the C++ Standard. Instead, the following alternative requirements apply: for an engine object `g` of type `G`,

- `G::scalar_type` is an unsigned integral type same as `sycl::vec<Type,N>::element_type`,
- `G::min()` and `G::max()` return a value of `G::scalar_type`,
- for each index `i` in the range `[0, N)`, `G::min() <= g()[i]` and `g()[i] <= G::max()`.

Effectively, these engines satisfy the standard *uniform random bit generator* requirements for each element of a `sycl::vec` returned by their `operator()`.

Similarly, for a distribution object `d` of a type `D` that is a template instantiated with `sycl::vec<Type,N>`:

- `D::scalar_type` is the same as `sycl::vec<Type,N>::element_type`,
- `D::min()` and `D::max()` return a value of `D::scalar_type`, and `D::min() <= D::max()`,
- `operator()` of a distribution returns a `sycl::vec<Type,N>` filled with random values in the closed interval `[D::min(), D::max()]`;

The following engines and engine adaptors with predefined parameters are defined:

```
template <int N>
using minstd_rand0_vec = linear_congruential_engine<sycl::vec<::std::uint_fast32_t, N>,␣
↪16807, 0, 2147483647>;

template <int N>
using minstd_rand_vec = linear_congruential_engine<sycl::vec<uint_fast32_t, N>, 48271, 0,
↪ 2147483647>;

template <int N>
using ranlux24_base_vec = subtract_with_carry_engine<sycl::vec<uint_fast32_t, N>, 24, 10,
↪ 24>;

template <int N>
using ranlux48_base_vec = subtract_with_carry_engine<sycl::vec<uint_fast64_t, N>, 48, 5,␣
↪12>;

template <int N>
using ranlux24_vec = discard_block_engine<ranlux24_base_vec<N>, 223, 23>;

template <int N>
using ranlux48_vec = discard_block_engine<ranlux48_base_vec<N>, 389, 11>;
```

Except for producing a `sycl::vec` of random values per invocation, the behavior of these engines is equivalent to the corresponding scalar engines, as described in the following table:

Engines and engine adaptors based on `sycl::vec<>`	C++ standard analogue	The 10000th scalar random value consecutively produced by a default-constructed object
`minstd_rand0_vec`	`minstd_rand0`	1043618065
`minstd_rand_vec`	`minstd_rand`	399268537
`ranlux24_base_vec`	`ranlux24_base`	7937952
`ranlux48_base_vec`	`ranlux48_base`	61839128582725
`ranlux24_vec`	`ranlux24`	9901578
`ranlux48_vec`	`ranlux48`	1112339016

3.3.3 Function Objects

The oneDPL function objects are defined in the <`oneapi/dpl/functional`> header.

```
namespace oneapi {
namespace dpl {
    struct identity
    {
        template <typename T>
        constexpr T&&
        operator()(T&& t) const noexcept;
    };
}
}
```

The `oneapi::dpl::identity` class implements an identity operation. Its function operator receives an instance of a type and returns the argument unchanged.

CHAPTER

FOUR

ONEDNN

oneAPI Deep Neural Network Library (oneDNN) is a performance library containing building blocks for deep learning applications and frameworks. oneDNN supports:

- CNN primitives (Convolutions, Inner product, Pooling, etc.)
- RNN primitives (LSTM, Vanilla, RNN, GRU)
- Normalizations (LRN, Batch, Layer)
- Elementwise operations (ReLU, Tanh, ELU, Abs, etc.)
- Softmax, Sum, Concat, Shuffle
- Reorders from/to optimized data layouts
- 8-bit integer, 16-, 32-bit, and bfloat16 floating point data types

```
// Tensor dimensions
int N, C, H, W;

// User-owned DPC++ objects
sycl::device dev {sycl::gpu_selector {}}; // Device
sycl::context ctx {dev}; // Context
sycl::queue queue {dev}; // Queue
std::vector<sycl::event> dependencies; // Input events dependencies
// Source
float *buf_src = static_cast<float *>(
        sycl::malloc_device((N * C * H * W) * sizeof(float), dev, ctx));
// Results
float *buf_dst = static_cast<float *>(
        sycl::malloc_device((N * C * H * W) * sizeof(float), dev, ctx));

// Create an engine encapsulating users' DPC++ GPU device and context
dnnl::engine engine = dnnl::sycl_interop::make_engine(dev, ctx);
// Create a stream encapsulating users' DPC++ GPU queue
dnnl::stream stream = dnnl::sycl_interop::make_stream(engine, queue);
// Create memory objects that use buf_src and buf_dst as the underlying storage
dnnl::memory mem_src({{N, C, H, W}, dnnl::memory::data_type::f32,
                              dnnl::memory::format_tag::nhwc},
        engine, buf_src);
dnnl::memory mem_dst({{N, C, H, W}, dnnl::memory::data_type::f32,
                              dnnl::memory::format_tag::nhwc},
        engine, buf_dst);
// Create a ReLU elementwise primitive
```

(continues on next page)

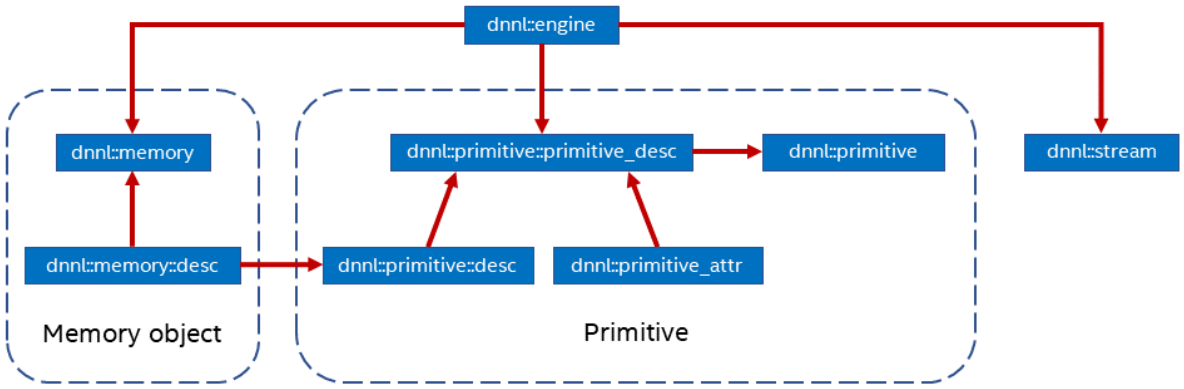
(continued from previous page)

```
dnnl::eltwise_forward relu {
        {{dnnl::prop_kind::forward_inference, dnnl::algorithm::eltwise_relu,
                mem_src.get_desc(), 0.f, 0.f},
               engine}};
// Execute the ReLU primitive in the stream passing input dependencies and
// retrieving the output dependency
sycl::event event = dnnl::sycl_interop::execute(relu, stream,
        {{DNNL_ARG_SRC, mem_src}, {DNNL_ARG_DST, mem_dst}}, dependencies);
```

4.1 Introduction

Although the origins of this specification are in the existing open source implementation, its goal is to define a *portable* set of APIs. To this end, for example, it intentionally omits implementation-specific details like tiled or blocked memory formats (layouts), and instead describes plain multi-dimensional memory formats and defines opaque *optimized* memory format that can be implementation specific.

oneDNN main concepts are *primitives*, *engines*, *streams*, and *memory objects*.



A *primitive* (`dnnl::primitive`) is a functor object that encapsulates a particular computation such as forward convolution, backward LSTM computations, or a data transformation operation. A single primitive can sometimes represent more complex *fused* computations such as a forward convolution followed by a ReLU. Fusion, among other things, is controlled via the *primitive attributes* mechanism.

The most important difference between a primitive and a pure function is that a primitive can be specialized for a subset of input parameters.

For example, a convolution primitive stores parameters like tensor shapes and can pre-compute other dependent parameters like cache blocking. This approach allows oneDNN primitives to pre-generate code specifically tailored for the requested operation to be performed. The oneDNN programming model assumes that the time it takes to perform the pre-computations is amortized by reusing the same primitive to perform computations multiple times.

A primitive may also need a mutable memory buffer that it may use for temporary storage only during computations. Such buffer is called a scratchpad. It can either be owned by a primitive object (which makes that object non-thread safe) or be an execution-time parameter.

Primitive creation is a potentially expensive operation. Users are expected to create primitives once and reuse them multiple times. Alternatively, implementations may reduce the primitive creation cost by caching primitives that have the same parameters. This optimization falls outside of the scope of this specification.

Engines (`dnnl::engine`) are an abstraction of a computational device: a CPU, a specific GPU card in the system, etc.

Most primitives are created to execute computations on one specific engine. The only exceptions are reorder primitives that may transfer data between two different engines.

Streams (`dnnl::stream`) encapsulate execution context tied to a particular engine. For example, they can correspond to DPC++ command queues.

Memory objects (`dnnl::memory`) encapsulate handles to memory allocated on a specific engine, tensor dimensions, data type, and memory format – the way tensor indices map to offsets in linear memory space. Memory objects are passed to primitives during execution.

Levels of Abstraction

oneDNN has multiple levels of abstractions for primitives and memory objects in order to expose maximum flexibility to its users.

On the logical level, the library provides the following abstractions:

- Memory descriptors (`dnnl::memory::desc`) define the logical dimensions of a tensor, data type, and the format in which the data is laid out in memory. The special format any (`dnnl::memory::format_tag::any`) indicates that the actual format will be defined later.
- Primitive descriptors (`dnnl::primitive_desc_base` is the base class and each of the supported primitives have their own version) can be used to inspect details of a specific primitive implementation like expected memory formats via queries to implement memory format propagation (see *Memory format propagation*) without having to fully instantiate a primitive.

Abstraction level	Memory object	Primitive objects
Logical description	Memory descriptor	Primitive descriptor
Implementation	Memory object	Primitive

4.1.1 Graph Extension

The graph extension is a high level abstraction in oneDNN that allows to work with a computation graph instead of individual primitives. This approach allows to make operation fusion:

- transparent: the integration efforts are reduced by abstracting engine-aware fusion logic.
- scalable: no integration code change is necessary to benefit from new fusion patterns enabled in the oneDNN implementation.

The programming model for the graph extension is detailed in the *graph programming model section*.

4.1.2 General API notes

oneDNN objects can be *empty* in which case they are not valid for any use. Memory descriptors are special in this regard, as their empty versions are regarded as *zero* memory descriptors that can be used to indicate absence of a memory descriptor. Empty objects are usually created using default constructors, but also may be a result of an error during object construction (see the next section).

4.1.3 Error Handling

All oneDNN functions throw the following exception in case of error.

struct **error** : public std::exception

oneDNN exception class.

This class captures the status returned by a failed function call

Additionally, many oneDNN functions that construct or return oneDNN objects have a boolean `allow_empty` parameter that defaults to `false` and that makes the library to return an empty object (a zero object in case of memory descriptors) when an object cannot be constructed instead of throwing an error.

4.1.4 Namespaces

All oneDNN functions and classes reside in `::dnnl` namespace. The functions that accept or return DPC++ objects such as command queues or buffers reside in `::dnnl::sycl_interop` namespace.

Furthermore, oneDNN defines `::oneapi::dnnl` namespace, that is an alias for the `::dnnl` namespace.

4.2 Conventions

oneDNN specification relies on a set of standard naming conventions for variables. This section describes these conventions.

4.2.1 Variable (Tensor) Names

Neural network models consist of operations of the following form:

$$\mathrm{dst} = f(\mathrm{src}, \mathrm{weights}),$$

where dst and src are activation tensors, and $\mathrm{weights}$ are learnable tensors.

The backward propagation therefore consists in computing the gradients with respect to the src ‘$and : math$:weights` respectively:

$$\mathrm{diff\_src} = \mathrm{d}f_{\mathrm{src}}(\mathrm{diff\_dst}, \mathrm{src}, \mathrm{weights}, \mathrm{dst}),$$

and

$$\mathrm{diff\_weights} = \mathrm{d}f_{\mathrm{weights}}(\mathrm{diff\_dst}, \mathrm{src}, \mathrm{weights}, \mathrm{dst}).$$

While oneDNN uses *src*, *dst*, and *weights* as generic names for the activations and learnable tensors, for a specific operation there might be commonly used and widely known specific names for these tensors. For instance, the *convolution* operation has a learnable tensor called *bias*. For usability reasons, oneDNN primitives use such names in initialization and other functions.

oneDNN uses the following commonly used notations for tensors:

Name	Meaning
`src`	Source tensor
`dst`	Destination tensor
`weights`	Weights tensor
`bias`	Bias tensor (used in *convolution*, *inner product* and other primitives)
`scale_shift`	Scale and shift tensors (used in *Batch Normalization* and *Layer normalization* primitives)
`workspace`	Workspace tensor that carries additional information from the forward propagation to the backward propagation
`scratchpad`	Temporary tensor that is required to store the intermediate results
`diff_src`	Gradient tensor with respect to the source
`diff_dst`	Gradient tensor with respect to the destination
`diff_weights`	Gradient tensor with respect to the weights
`diff_bias`	Gradient tensor with respect to the bias
`diff_scale`	Gradient tensor with respect to the scale
`diff_shift`	Gradient tensor with respect to the shift
`*_layer`	RNN layer data or weights tensors
`*_iter`	RNN recurrent data or weights tensors

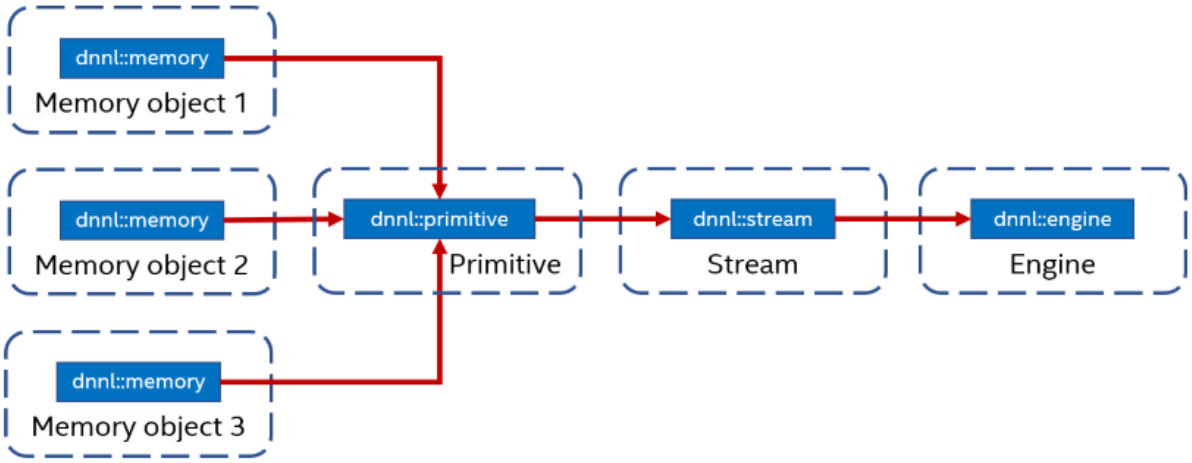
4.2.2 RNN-Specific Notation

The following notations are used when describing RNN primitives.

Name	Semantics
$\cdot$	matrix multiply operator
$*$	elementwise multiplication operator
W	input weights
U	recurrent weights
$\square^T$	transposition
B	bias
h	hidden state
a	intermediate value
x	input
$\square_t$	timestamp index
$\square_l$	layer index
activation	tanh, relu, logistic
c	cell state
$\tilde{c}$	candidate state
i	input gate
f	forget gate
o	output gate
u	update gate
r	reset gate

4.3 Execution Model

To execute a primitive, a user needs to pass memory arguments and a stream to the *dnnl::primitive::execute()* member function.



The primitive's computations are executed on the computational device corresponding to the engine on which the primitive (and memory arguments) were created and happens within the context of the stream.

4.3.1 Engine

Engine is abstraction of a computational device: a CPU, a specific GPU card in the system, etc. Most primitives are created to execute computations on one specific engine. The only exceptions are reorder primitives that transfer data between two different engines.

Engines correspond to and can be constructed from pairs of the DPC++ `sycl::device` and `sycl::context` objects. Alternatively, oneDNN itself can create and own the corresponding objects.

struct **engine**

An execution engine.

Public Types

enum class **kind**

Kinds of engines.

Values:

enumerator **any**

An unspecified engine.

enumerator **cpu**

CPU engine.

enumerator **gpu**

GPU engine.

Public Functions

engine() = default

Constructs an empty engine. An empty engine cannot be used in any operations.

engine(*kind* akind, size_t index)

Constructs an engine.

Parameters

- **akind** – The kind of engine to construct.
- **index** – The index of the engine. Must be less than the value returned by *get_count()* for this particular kind of engine.

kind **get_kind**() const

Returns the kind of the engine.

Returns

The kind of the engine.

Public Static Functions

static size_t **get_count**(*kind* akind)

Returns the number of engines of a certain kind.

Parameters

akind – The kind of engines to count.

Returns

The number of engines of the specified kind.

engine dnnl::sycl_interop::**make_engine**(const cl::sycl::device &adevice, const cl::sycl::context &acontext)

Creates an engine object using a specified SYCL device and SYCL context objects.

Parameters

- **adevice** – SYCL device.
- **acontext** – SYCL context.

Returns

Engine object for the `adevice` SYCL device, within the specified `acontext` SYCL context.

cl::sycl::device dnnl::sycl_interop::**get_device**(const *engine* &aengine)

Returns the SYCL device underlying a specified engine object.

Parameters

aengine – Engine object.

Returns

SYCL device object underlying the `aengine` engine object.

cl::sycl::context dnnl::sycl_interop::**get_context**(const *engine* &aengine)

Returns the SYCL context underlying a specified engine object.

Parameters
aengine – Engine object.

Returns
SYCL context object underlying the aengine engine object.

4.3.2 Stream

A *stream* is an encapsulation of execution context tied to a particular engine. They are passed to *dnnl::primitive::execute()* when executing a primitive.

Streams correspond to and can be constructed from DPC++ sycl::queue objects. Alternatively, oneDNN itself can create and own the corresponding objects. Streams are considered to be ephemeral and can be created / destroyed as long these operation do not violate DPC++ synchronization requirements.

Similar to DPC++ queues, streams can be in-order and out-of-order (see the relevant portion of the DPC++ specification for the explanation). The desired behavior can be specified using *dnnl::stream::flags* value. A stream created from a DPC++ queue inherits its behavior.

struct **stream**

An execution stream.

Public Types

enum class **flags** : unsigned

Stream flags. Can be combined using the bitwise OR operator.

Values:

enumerator **in_order**

In-order execution.

enumerator **out_of_order**

Out-of-order execution.

enumerator **default_flags**

Default stream configuration.

Public Functions

stream()

Constructs an empty stream. An empty stream cannot be used in any operations.

stream(const *engine* &aengine, *flags* aflags = *flags*::*default_flags*)

Constructs a stream for the specified engine and with behavior controlled by the specified flags.

Parameters

- **aengine** – Engine to create the stream on.

- **aflags** – Flags controlling stream behavior.

engine **get_engine**() const

Returns
The associated engine.

stream &**wait**()

Waits for all primitives executing in the stream to finish.

Returns
The stream itself.

stream dnnl::sycl_interop::**make_stream**(const *engine* &aengine, cl::sycl::queue &aqueue)

Creates a stream for a specified engine and SYCL queue objects.

Parameters

- **aengine** – Engine object to use for the stream.
- **aqueue** – SYCL queue to use for the stream.

Returns
Stream object for the `aengine` engine object, which holds the `aqueue` SYCL queue object.

cl::sycl::queue dnnl::sycl_interop::**get_queue**(const *stream* &astream)

Returns the SYCL queue underlying a specified stream object.

Parameters
astream – Stream object.

Returns
SYCL queue underlying the `astream` stream object.

4.4 Data model

Data in oneDNN is stored in *memory objects* that both store and describe data that can be of various types and be stored in different formats (layouts).

4.4.1 Data types

oneDNN supports multiple data types. However, the 32-bit IEEE single-precision floating-point data type is the fundamental type in oneDNN. It is the only data type that must be supported by an implementation. All the other types discussed below are optional.

Primitives operating on the single-precision floating-point data type consume data, produce, and store intermediate results using the same data type.

Moreover, single-precision floating-point data type is often used for intermediate results in the mixed precision computations because it provides better accuracy. For example, the elementwise primitive and elementwise post-ops always use it internally.

Note: Implicit downconversion can be enabled in order to speedup computations, and are controlled using the *fpmath mode controls*

---

oneDNN uses the following enumeration to refer to data types it supports:

enum class dnnl::*memory*::**data_type**

Data type specification.

Values:

enumerator **undef**

Undefined data type (used for empty memory descriptors).

enumerator **f16**

16-bit/half-precision floating point.

enumerator **bf16**

non-standard 16-bit floating point with 7-bit mantissa.

enumerator **f32**

32-bit/single-precision floating point.

enumerator **s32**

32-bit signed integer.

enumerator **s8**

8-bit signed integer.

enumerator **u8**

8-bit unsigned integer.

oneDNN supports training and inference with the following data types:

Usage mode	Data types
inference	*dnnl::memory::data_type::f32*, *dnnl::memory::data_type::bf16*, *dnnl::memory::data_type::f16*, *dnnl::memory::data_type::s8*/*dnnl::memory::data_type::u8*
training	*dnnl::memory::data_type::f32*, *dnnl::memory::data_type::bf16*

Note: Using lower precision arithmetic may require changes in the deep learning model implementation.

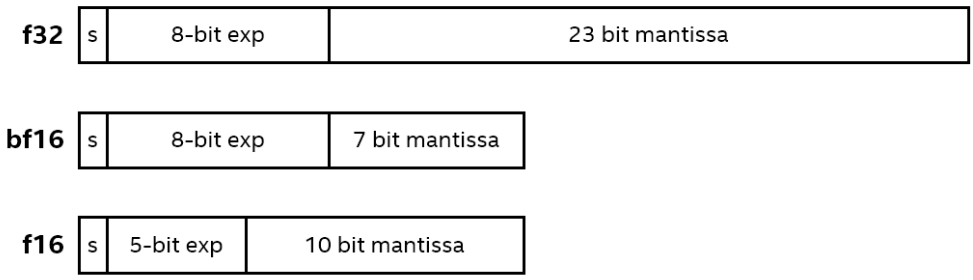
Individual primitives may have additional limitations with respect to data type support based on the precision requirements. The list of data types supported by each primitive is included in the corresponding sections of the specification guide.

Bfloat16

Note: In this section we abbreviate data types names for readability. For example, *dnnl::memory::data_type::f32* is abbreviated to *f32*.

Bfloat16 (*bf16*) is a 16-bit floating point data type based on the IEEE 32-bit single-precision floating point data type (*f32*).

Both *bf16* and *f32* have an 8-bit exponent. However, while *f32* has a 23-bit mantissa, *bf16* has only a 7-bit one, keeping only the most significant bits. As a result, while these data types support a very close numerical range of values, *bf16* has a significantly reduced precision. Therefore, *bf16* occupies a spot between *f32* and the IEEE 16-bit half-precision floating point data type, *f16*. Compared directly to *f16*, which has a 5-bit exponent and a 10-bit mantissa, *bf16* trades increased range for reduced precision.



More details of the bfloat16 data type can be found here.

One of the advantages of using *bf16* versus *f32* is reduced memory footprint and, hence, increased memory access throughput.

Workflow

The main difference between implementing training with the *f32* data type and with the *bf16* data type is the way the weights updates are treated. With the *f32* data type, the weights gradients have the same data type as the weights themselves. This is not necessarily the case with the *bf16* data type as oneDNN allows some flexibility here. For example, one could maintain a master copy of all the weights, computing weights gradients in *f32* and converting the result to *bf16* afterwards.

Support

Most of the primitives can support the *bf16* data type for source and weights tensors. Destination tensors can be specified to have either the *bf16* or *f32* data type. The latter is intended for cases in which the output is to be fed to operations that do not support bfloat16 or require higher precision.

Int8

To push higher performance during inference computations, recent work has focused on computations that use activations and weights stored at lower precision to achieve higher throughput. Int8 computations offer improved performance over higher-precision types because they enable packing more computations into a single instruction, at the cost of reduced (but acceptable) accuracy.

Workflow

oneDNN support symmetric and asymmetric quantization models.

Quantization Model

For each int8 tensor, the oneDNN library allows to specify scaling factors and zero-points (also referred to as quantization parameters), and assumes the following mathematical relationship:

$$x_{f32}[:] = scale_x \cdot (x_{int8}[:] - zp_x)$$

where $scale_x$ is a *scaling factor* in float format, zp_x is the zero point in int32 format, and $[:]$ is used to denote elementwise application of the formula to the arrays. In order to provide best performance, oneDNN does not compute those scaling factors and zero-points as part of primitive computation. Those should be provided by the user through the *attribute mechanism*.

These quantization parameters can either be computed ahead of time using calibration tools (*static* quantization) or at runtime based on the actual minimum and maximum values of a tensor (*dynamic* quantization). Either method can be used in conjunction with oneDNN, as the quantization parameters are passed to the oneDNN primitives at execution time.

To support int8 quantization, primitives should be created and executed as follow:

- during primitive creation, if one or multiple inputs are int8 (signed or not), then the primitive will behave as a quantized integer operation.
- still during primitive creation, the dimensionality of the scaling factors and zero-point should be provided using masks (e.g. one scale per tensor, one scale per channel, ...).
- finally, during primitive execution, the user must provide the actual quantization parameters as arguments to the execute function. Scales shall be f32 values, and zero-points shall be int32 values.

Note: For performance reasons, each primitive implementation can support only a subset of quantization parameter masks. For example, convolution typically supports per-channel scales (no zero-point) for weights, and per-tensor scaling factor and zero-point for activation.

Note: Some primitives might use quantization parameters in order to dequantize/quantize intermediate values. This is for example the case for the *RNN* primitive, which will dequantize before applying non linear functions, and will requantize before executing matrix multiplication operations.

Numerical behavior

Primitive implementations are allowed to convert int8 inputs to wider datatypes (e.g. int16 or int32), as those conversions do not impact accuracy.

During execution, primitives should avoid integer overflows and maintain integer accuracy by using wider datatypes (e.g. int32) for intermediate values and accumulators. Those are then converted as necessary before the result is written to the output memory objects. During that conversion, the behavior in case of underflow/overflow is undefined (e.g. when converting *s32* to int8). However, it is highly encouraged for implementations to saturate values.

When multiple operations are fused in a single primitive using the *post-op mecanism*, those are assumed to be computed in f32 precision. As a result the destination quantization parameters are applied after the post-ops as follow:

$$\mathrm{dst}[:] = post\_ops(OP(src[:], weights[:], ...))/scale_{\mathrm{dst}} + zp_{\mathrm{dst}}$$

Quantizing/dequantizing values between post-operations can still be achieved using one of *eltwise post-ops*, *binary post-ops*, or the scale parameter of the appropriate post-operation.

Example: Convolution Quantization Workflow

Consider a convolution without bias. The tensors are represented as:

- $\mathrm{src}_{f32}[:] = scale_{\mathrm{src}} \cdot (\mathrm{src}_{int8}[:] - zp_{\mathrm{src}})$
- $\mathrm{weights}_{f32}[:] = scale_{\mathrm{weights}} \cdot \mathrm{weights}_{int8}[:]$
- $\mathrm{dst}_{f32}[:] = scale_{\mathrm{dst}} \cdot (\mathrm{dst}_{int8}[:] - zp_{\mathrm{dst}})$

Here the $\mathrm{src}_{f32}, \mathrm{weights}_{f32}, \mathrm{dst}_{f32}$ are not computed at all, the whole work happens with int8 tensors.So the task is to compute the dst_{int8} tensor, using the *src_{int8}*, *weights_{int8}* tensors passed at execution time, as well as the corresponding quantization parameters *scale_{src}, scale_{weights}, scale_{dst}* and *zero_point{src}, zero_point_{dst}*. Mathematically, the computations are:

$$\mathrm{dst}_{int8}[:] = \mathrm{f32\_to\_int8}(scale_{\mathrm{src}} \cdot scale_{\mathrm{weights}} \cdot \mathrm{s32\_to\_f32}(conv_{s32}(\mathrm{src}_{int8}, \mathrm{weights}_{int8}) - zp_{\mathrm{src}} \cdot comp_{s32})/scale_{\mathrm{dst}} + zp_{\mathrm{dst}})$$

where

- $conv_{s32}$ is just a regular convolution which takes source and weights with int8 data type and compute the result in int32 data type (int32 is chosen to avoid overflows during the computations);
- $comp_{s32}$ is a compensation term to account for *src* non-zero zero point. This term is computed by the oneDNN library and can typically be pre-computed ahead of time, for example during weights reorder.
- $\mathrm{f32\_to\_s8}()$ converts an *f32* value to *s8* with potential saturation if the values are out of the range of the int8 data type.
- $\mathrm{s32\_to\_f32}()$ converts an *int8* value to *f32* with potential rounding. This conversion is typically necessary to apply *f32* scaling factors.

Per-Channel Scaling

Primitives may have limited support of multiple scales for a quantized tensor. The most popular use case is the *Convolution and Deconvolution* primitives that support per-output-channel scaling factors for the weights, meaning that the actual convolution computations would need to scale different output channels differently.

- $\text{src}_{f32}(n, ic, ih, iw) = scale_{\text{src}} \cdot \text{src}_{int8}(n, ic, ih, iw)$
- $\text{weights}_{f32}(oc, ic, kh, kw) = scale_{\text{weights}}(oc) \cdot \text{weights}_{int8}(oc, ic, kh, kw)$
- $\text{dst}_{f32}(n, oc, oh, ow) = scale_{\text{dst}} \cdot \text{dst}_{int8}(n, oc, oh, ow)$

Note that now the weights' scaling factor depends on oc.

To compute the dst_{int8} we need to perform the following:

$$\text{dst}_{int8}(n, oc, oh, ow) = \text{f32\_to\_int8}(\frac{scale_{\text{src}} \cdot scale_{\text{weights}}(oc)}{scale_{\text{dst}}} \cdot conv_{s32}(\text{src}_{int8}, \text{weights}_{int8})|_{(n,oc,oh,ow)}).$$

The user is responsible for preparing quantized weights accordingly. To do that, oneDNN provides reorders that can perform per-channel scaling:

$$\text{weights}_{int8}(oc, ic, kh, kw) = \text{f32\_to\_int8}(\text{weights}_{f32}(oc, ic, kh, kw)/scale_{weights}(oc)).$$

The *Quantization* describes what kind of quantization model oneDNN supports.

Support

oneDNN supports int8 computations for inference by allowing to specify that primitive input and output memory objects use int8 data types.

4.4.2 Memory

There are two levels of abstraction for memory in oneDNN.

1. *Memory descriptor* – engine-agnostic logical description of data (number of dimensions, dimension sizes, *data type*, and *format*.
2. *Memory object* – an engine-specific object combines memory descriptor with storage.

oneDNN defines the following convenience aliases to denote tensor dimensions

using dnnl::*memory*::**dim** = int64_t

Integer type for representing dimension sizes and indices.

using dnnl::*memory*::**dims** = std::vector<*dim*>

Vector of dimensions. Implementations are free to force a limit on the vector's length.

Memory Formats

In oneDNN memory format is how a multidimensional tensor is stored in 1-dimensional linear memory address space. oneDNN specifies two kinds of memory formats: *plain* which correspond to traditional multidimensional arrays, and *optimized* which are completely opaque.

Plain Memory Formats

Plain memory formats describe how multidimensional tensors are laid out in memory using an array of $\mathrm{dimensions}$ and an array of $\mathrm{strides}$ both of which have length equal to the rank of the tensor. In oneDNN the order of dimensions is fixed and different dimensions can have certain canonical interpretation depending on the primitive. For example, for CNN primitives the order for activation tensors is $\{N, C, ..., D, H, W\}$, where N stands for minibatch (or batch size), C stands for channels, and D, H, and W stand for image spatial dimensions: depth, height and width respectively. Spatial dimensions may be omitted in the order from outermost to innermost; for example, it is not possible to omit H when D is present and it is never possible to omit W. Canonical interpretation is documented for each primitive. However, this means that the $\mathrm{strides}$ array plays an important role defining the order in which different dimension are laid out in memory. Moreover, the $\mathrm{strides}$ need to agree with $\mathrm{dimensions}$.

More precisely, let T be a tensor of rank n and let σ be the permutation of the $\mathrm{strides}$ array that sorts it, i.e. $\mathrm{strides}[i] \geq \mathrm{strides}[j]$ if $\sigma(i) < \sigma(j)$ for all $0 \leq i, j < n$. Then the following must hold:

$$\mathrm{strides}[i] \geq \mathrm{strides}[j] * \mathrm{dimensions}[j] \text{ if } \sigma(i) < \sigma(j) \text{ for all } 0 \leq i, j < n.$$

For an element with coordinates $(i_0, \ldots, i_{n-1})$ such that $0 \leq i_j < \mathrm{dimensions}[j]$ for $0 \leq j < n$, its offset in memory is computed as:

$$\mathrm{offset}(i_0, \ldots, i_{n-1}) = \mathrm{offset}_0 + \sum_{j=0}^{n-1} i_j * \mathrm{strides}[j].$$

Here offset_0 is the offset from the *parent* memory and is non-zero only for *submemory* memory descriptors created using *dnnl::memory::desc::submemory_desc()*. Submemory memory descriptors inherit strides from the parent memory descriptor. Their main purpose is to express in-place concat operations.

As an example, consider an $M \times N$ matrix A (M rows times N columns). Regardless of whether A is stored transposed or not, $\mathrm{dimensions}_A = \{M, N\}$. However, $\mathrm{strides}_A = \{LDA, 1\}$ if it is not transposed and $\mathrm{strides}_A = \{1, LDA\}$ if it is, where LDA is such that $LDA \geq N$ if A is not transposed, and $LDA \geq M$ if it is. This also shows that A does not have to be stored *densly* in memory.

Note: The example above shows that oneDNN assumes data to be stored in row-major order.

Code example:

```
int M, N;
dnnl::memory::dims dims {M, N}; // Dimensions always stay the same

// Non-transposed matrix
dnnl::memory::dims strides_non_transposed {N, 1};
dnnl::memory::desc A_non_transposed {dims, dnnl::memory::data_type::f32,
        strides_non_transposed};

// Transposed matrix
dnnl::memory::dims strides_transposed {1, M};
dnnl::memory::desc A_transposed {dims, dnnl::memory::data_type::f32,
        strides_transposed};
```

Format Tags

In addition to strides, oneDNN provides named *format tags* via the *dnnl::memory::format_tag* enum type. The enumerators of this type can be used instead of strides for dense plain layouts.

The format tag names for N-dimensional memory formats use first N letters of the English alphabet which can be arbitrarily permuted. This permutation is used to compute strides for tensors with up to 6 dimensions. The resulting strides specify dense storage, in other words, using the nomenclature from the previous section, the following equality holds:

$$\text{strides}[i] = \text{strides}[j] * \text{dimensions}[j] \text{ if } \sigma(i) + 1 = \sigma(j) \text{ for all } 0 \leq i, j < n - 1.$$

In the matrix example, we could have used *dnnl::memory::format_tag::ab* for the non-transposed matrix above, and *dnnl::memory::format_tag::ba* for the transposed:

```
int M, N;
dnnl::memory::dims dims {M, N}; // Dimensions always stay the same

// Non-transposed matrix
dnnl::memory::desc A_non_transposed {dims, dnnl::memory::data_type::f32,
        dnnl::memory::format_tag::ab};

// Transposed matrix
dnnl::memory::desc A_transposed {dims, dnnl::memory::data_type::f32,
        dnnl::memory::format_tag::ba};
```

Note: In what follows in this section we abbreviate memory format tag names for readability. For example, *dnnl::memory::format_tag::abcd* is abbreviated to *abcd*.

In addition to abstract format tag names, oneDNN also provides convenience aliases. Some examples for CNNs and RNNs:

- *nchw* is an alias for *abcd* (see the canonical order order of dimensions for CNNs discussed above).
- *oihw* is an alias for *abcd*.
- *nhwc* is an alias for *acdb*.
- *tnc* is an alias for *abc*.
- *ldio* is an alias for *abcd*.
- *ldoi* is an alias for *abdc*.

Optimized Format 'any'

Another kind of format that oneDNN supports is an opaque *optimized* memory format that cannot be created directly from strides and dimensions arrays. A memory descriptor for an optimized memory format can only be created by passing *any* when creating certain primitive descriptor. That primitive descriptor can then querying them for memory descriptors. Data in plain memory format should then be reordered into the data in optimized data format before computations. Since reorders are expensive, the optimized memory format needs to be propagated through computations graph.

Optimized formats can employ padding, blocking and other data transformations to keep data in layout optimal for a certain architecture. This means that it in general operations like *dnnl::memory::desc::permute_axes()* or

dnnl::memory::desc::submemory_desc() may fail. It is in general incorrect to use product of dimension sizes to calculate amount of memory required to store data: *dnnl::memory::desc::get_size()* must be used instead.

Memory Format Propagation

Memory format propagation is one of the central notions that needs to be well-understood to use oneDNN correctly.

Convolution, matmul, RNN and inner product primitives choose the memory format when you create them with the placeholder memory format *any* for input or output. The memory format chosen is based on different circumstances such as hardware and convolution parameters. Using the placeholder *any* memory format is the recommended practice for convolutions, since they are the most compute-intensive operations in most topologies where they are present.

Other primitives, such as Elementwise, LRN, batch normalization and other, on forward propagation should use the same memory format as the preceding layer thus propagating the memory format through multiple oneDNN primitives. This avoids unnecessary reorders which may be expensive and should be avoided unless a compute-intensive primitive requires a different format. For performance reasons, backward computations of such primitives requires consistent memory format with the corresponding forward computations. Hence, when initializing there primitives for backward computations you should use *dnnl::memory::format_tag::any* memory format tag as well.

Below is the short summary when to use and not to use memory format *any* during primitive descriptor construction:

Primitive Kinds	Forward Propagation	Backward Propagation	No Propagation
Compute intensive: (De-)convolution, Matmul, Inner product, RNN	Use *any*	Use *any*	N/A
Memory-bandwidth limited: Pooling, Layer and Batch Normalization, Local Response Normalization, Elementwise, Shuffle, Softmax	Use memory format from preceding layer for source tensors, and *any* for destination tensors	Use *any* for gradient tensors, and actual memory formats for data tensors	N/A
Memory-bandwidth limited: Reorder, Concat, Sum, Binary	N/A	N/A	Use memory format from preceding layer for source tensors, and *any* for destination tensors

Additional format synchronization is required between forward and backward propagation when running training workloads. This is achieved via the `hint_pd` arguments of primitive descriptor constructors for primitives that implement backward propagation.

API

enum class `dnnl::memory::format_tag`

Memory format tag specification.

Memory format tags can be further divided into two categories:

- Domain-agnostic names, i.e. names that do not depend on the tensor usage in the specific primitive. These names use letters from `a` to `f` to denote logical dimensions and form the order in which the dimensions are laid in memory. For example, *dnnl::memory::format_tag::ab* is used to denote a 2D tensor where the second logical dimension (denoted as `b`) is the innermost, i.e. has stride = 1, and the first logical

dimension (a) is laid out in memory with stride equal to the size of the second dimension. On the other hand, *dnnl::memory::format_tag::ba* is the transposed version of the same tensor: the outermost dimension (a) becomes the innermost one.

- Domain-specific names, i.e. names that make sense only in the context of a certain domain, such as CNN. These names are aliases to the corresponding domain-agnostic tags and used mostly for convenience. For example, *dnnl::memory::format_tag::nc* is used to denote 2D CNN activations tensor memory format, where the channels dimension is the innermost one and the batch dimension is the outermost one. Moreover, *dnnl::memory::format_tag::nc* is an alias for *dnnl::memory::format_tag::ab*, because for CNN primitives the logical dimensions of activations tensors come in order: batch, channels, spatial. In other words, batch corresponds to the first logical dimension (a), and channels correspond to the second one (b).

The following domain-specific notation applies to memory format tags:

- 'n' denotes the mini-batch dimension
- 'c' denotes a channels dimension
- When there are multiple channel dimensions (for example, in convolution weights tensor), 'i' and 'o' denote dimensions of input and output channels
- 'g' denotes a groups dimension for convolution weights
- 'd', 'h', and 'w' denote spatial depth, height, and width respectively

Values:

enumerator **undef**

Undefined memory format tag.

enumerator **any**

Placeholder memory format tag. Used to instruct the primitive to select a format automatically.

enumerator **a**

plain 1D tensor

enumerator **ab**

plain 2D tensor

enumerator **ba**

permuted 2D tensor

enumerator **abc**

plain 3D tensor

enumerator **acb**

permuted 3D tensor

enumerator **bac**

permuted 3D tensor

enumerator **bca**

permuted 3D tensor

enumerator **cba**
permuted 3D tensor

enumerator **abcd**
plain 4D tensor

enumerator **abdc**
permuted 4D tensor

enumerator **acdb**
permuted 4D tensor

enumerator **bacd**
permuted 4D tensor

enumerator **bcda**
permuted 4D tensor

enumerator **cdba**
permuted 4D tensor

enumerator **dcab**
permuted 4D tensor

enumerator **abcde**
plain 5D tensor

enumerator **abdec**
permuted 5D tensor

enumerator **acbde**
permuted 5D tensor

enumerator **acdeb**
permuted 5D tensor

enumerator **bacde**
permuted 5D tensor

enumerator **bcdea**
permuted 5D tensor

enumerator **cdeba**
permuted 5D tensor

enumerator **decab**

permuted 5D tensor

enumerator **abcdef**

plain 6D tensor

enumerator **acbdef**

plain 6D tensor

enumerator **defcab**

plain 6D tensor

enumerator **x**

1D tensor; an alias for *dnnl::memory::format_tag::a*

enumerator **nc**

2D CNN activations tensor; an alias for *dnnl::memory::format_tag::ab*

enumerator **cn**

2D CNN activations tensor; an alias for *dnnl::memory::format_tag::ba*

enumerator **tn**

2D RNN statistics tensor; an alias for *dnnl::memory::format_tag::ab*

enumerator **nt**

2D RNN statistics tensor; an alias for *dnnl::memory::format_tag::ba*

enumerator **ncw**

3D CNN activations tensor; an alias for *dnnl::memory::format_tag::abc*

enumerator **nwc**

3D CNN activations tensor; an alias for *dnnl::memory::format_tag::acb*

enumerator **nchw**

4D CNN activations tensor; an alias for *dnnl::memory::format_tag::abcd*

enumerator **nhwc**

4D CNN activations tensor; an alias for *dnnl::memory::format_tag::acdb*

enumerator **chwn**

4D CNN activations tensor; an alias for *dnnl::memory::format_tag::bcda*

enumerator **ncdhw**

5D CNN activations tensor; an alias for *dnnl::memory::format_tag::abcde*

enumerator **ndhwc**

5D CNN activations tensor; an alias for *dnnl::memory::format_tag::acdeb*

enumerator **oi**

2D CNN weights tensor; an alias for *dnnl::memory::format_tag::ab*

enumerator **io**

2D CNN weights tensor; an alias for *dnnl::memory::format_tag::ba*

enumerator **oiw**

3D CNN weights tensor; an alias for *dnnl::memory::format_tag::abc*

enumerator **owi**

3D CNN weights tensor; an alias for *dnnl::memory::format_tag::acb*

enumerator **wio**

3D CNN weights tensor; an alias for *dnnl::memory::format_tag::cba*

enumerator **iwo**

3D CNN weights tensor; an alias for *dnnl::memory::format_tag::bca*

enumerator **oihw**

4D CNN weights tensor; an alias for *dnnl::memory::format_tag::abcd*

enumerator **hwio**

4D CNN weights tensor; an alias for *dnnl::memory::format_tag::cdba*

enumerator **ohwi**

4D CNN weights tensor; an alias for *dnnl::memory::format_tag::acdb*

enumerator **ihwo**

4D CNN weights tensor; an alias for *dnnl::memory::format_tag::bcda*

enumerator **iohw**

4D CNN weights tensor; an alias for *dnnl::memory::format_tag::bacd*

enumerator **oidhw**

5D CNN weights tensor; an alias for *dnnl::memory::format_tag::abcde*

enumerator **dhwio**

5D CNN weights tensor; an alias for *dnnl::memory::format_tag::cdeba*

enumerator **odhwi**

5D CNN weights tensor; an alias for *dnnl::memory::format_tag::acdeb*

enumerator **iodhw**

5D CNN weights tensor; an alias for *dnnl::memory::format_tag::bacde*

enumerator **idhwo**

5D CNN weights tensor; an alias for *dnnl::memory::format_tag::bcdea*

enumerator **goiw**

4D CNN weights tensor with groups; an alias for *dnnl::memory::format_tag::abcd*

enumerator **wigo**

4D CNN weights tensor with groups; an alias for *dnnl::memory::format_tag::dcab*

enumerator **goihw**

5D CNN weights tensor with groups; an alias for *dnnl::memory::format_tag::abcde*

enumerator **hwigo**

5D CNN weights tensor with groups; an alias for *dnnl::memory::format_tag::decab*

enumerator **giohw**

5D CNN weights tensor with groups; an alias for *dnnl::memory::format_tag::acbde*

enumerator **goidhw**

6D CNN weights tensor with groups; an alias for *dnnl::memory::format_tag::abcdef*

enumerator **giodhw**

6D CNN weights tensor with groups; an alias for *dnnl::memory::format_tag::abcdef*

enumerator **dhwigo**

6D CNN weights tensor with groups; an alias for *dnnl::memory::format_tag::defcab*

enumerator **tnc**

3D RNN data tensor in the format (seq_length, batch, input channels).

enumerator **ntc**

3D RNN data tensor in the format (batch, seq_length, input channels).

enumerator **ldnc**

4D RNN states tensor in the format (num_layers, num_directions, batch, state channels).

enumerator **ldigo**

5D RNN weights tensor in the format (num_layers, num_directions, input_channels, num_gates, output_channels).

- For LSTM cells, the gates order is input, forget, candidate and output gate.
- For GRU cells, the gates order is update, reset and output gate.

enumerator **ldgoi**

5D RNN weights tensor in the format (num_layers, num_directions, num_gates, output_channels, input_channels).

- For LSTM cells, the gates order is input, forget, candidate and output gate.
- For GRU cells, the gates order is update, reset and output gate.

enumerator **ldio**

4D LSTM projection tensor in the format (num_layers, num_directions, num_channels_in_hidden_state, num_channels_in_recurrent_projection).

enumerator **ldoi**

4D LSTM projection tensor in the format (num_layers, num_directions, num_channels_in_recurrent_projection, num_channels_in_hidden_state).

enumerator **ldgo**

4D RNN bias tensor in the format (num_layers, num_directions, num_gates, output_channels).

- For LSTM cells, the gates order is input, forget, candidate and output gate.
- For GRU cells, the gates order is update, reset and output gate.

Memory Descriptors and Objects

Descriptors

Memory descriptor is an engine-agnostic logical description of data (number of dimensions, dimension sizes, and data type), and, optionally, the information about the physical format of data in memory. If this information is not known yet, a memory descriptor can be created with format tag set to *dnnl::memory::format_tag::any*. This allows compute-intensive primitives to chose the most appropriate physical format for the computations. The user is then responsible for reordering their data into the proper format they do not match. See *Memory Format Propagation* for more details.

A memory descriptor can be initialized either by specifying dimensions, and memory format tag or strides for each of them.

User can query amount of memory required by a memory descriptor using the *dnnl::memory::desc::get_size()* function. The size of data in general cannot be computed as the product of dimensions multiplied by the size of the data type. So users are required to use this function for better code portability.

Two memory descriptors can be compared using the equality and inequality operators. The comparison is especially useful when checking whether it is necessary to reorder data from the user's data format to a primitive's format.

Along with ordinary memory descriptors with all dimensions being positive, oneDNN supports *zero-volume* memory descriptors with one or more dimensions set to zero. This is used to support the NumPy* convention. If a zero-volume memory is passed to a primitive, the primitive typically does not perform any computations with this memory. For example:

- The concatenation primitive would ignore all memory object with zeroes in the concatenation dimension / axis.

- A forward convolution with a source memory object with zero in the minibatch dimension would always produce a destination memory object with a zero in the minibatch dimension and perform no computations.
- However, a forward convolution with a zero in one of the weights dimensions is ill-defined and is considered to be an error by the library because there is no clear definition on what the output values should be.

Data handle of a zero-volume memory is never accessed.

Note: Some primitives support implicit broadcast semantic when a given tensor has a dimensions set to 1 (similar to NumPY broadcast semantic). In particular, if an operation expects two tensors with same dimensions, and one of the descriptors has some dimension set to 1, that dimension will be implicitly broadcasted to match the other tensor dimension.

API

struct **desc**

A memory descriptor.

Public Functions

desc()

Constructs a zero (empty) memory descriptor. Such a memory descriptor can be used to indicate absence of an argument.

desc(const *dims* &adims, *data_type* adata_type, *format_tag* aformat_tag, bool allow_empty = false)

Constructs a memory descriptor.

Note: The logical order of dimensions corresponds to the `abc...` format tag, and the physical meaning of the dimensions depends both on the primitive that would operate on this memory and the operation context.

Parameters

- **adims** – Tensor dimensions.
- **adata_type** – Data precision/type.
- **aformat_tag** – Memory format tag.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case a zero memory descriptor will be constructed. This flag is optional and defaults to false.

desc(const *dims* &adims, *data_type* adata_type, const *dims* &strides, bool allow_empty = false)

Constructs a memory descriptor by strides.

Note: The logical order of dimensions corresponds to the `abc...` format tag, and the physical meaning of the dimensions depends both on the primitive that would operate on this memory and the operation context.

Parameters

- **adims** – Tensor dimensions.
- **adata_type** – Data precision/type.
- **strides** – Strides for each dimension.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case a zero memory descriptor will be constructed. This flag is optional and defaults to false.

desc **submemory_desc**(const *dims* &adims, const *dims* &offsets, bool allow_empty = false) const

Constructs a memory descriptor for a region inside an area described by this memory descriptor.

Parameters

- **adims** – Sizes of the region.
- **offsets** – Offsets to the region from the encompassing memory object in each dimension.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case a zero memory descriptor will be returned. This flag is optional and defaults to false.

Returns

A memory descriptor for the region.

desc **reshape**(const *dims* &adims, bool allow_empty = false) const

Constructs a memory descriptor by reshaping an existing one. The new memory descriptor inherits the data type.

The operation ensures that the transformation of the physical memory format corresponds to the transformation of the logical dimensions. If such transformation is impossible, the function either throws an exception (default) or returns a zero memory descriptor depending on the `allow_empty` flag.

The reshape operation can be described as a combination of the following basic operations:

i. Add a dimension of size 1. This is always possible.

ii. Remove a dimension of size 1.

iii. Split a dimension into multiple ones. This is possible only if the product of all tensor dimensions stays constant.

iv. Join multiple consecutive dimensions into a single one. This requires that the dimensions are dense in memory and have the same order as their logical counterparts.

- Here, 'dense' means: `stride for dim[i] == (stride for dim[i + 1]) * dim[i + 1]`;
- And 'same order' means: `i < j` if and only if `stride for dim[j] <= stride for dim[i]`.

Note: Reshape may fail for optimized memory formats.

Parameters

- **adims** – New dimensions. The product of dimensions must remain constant.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case a zero memory descriptor will be returned. This flag is optional and defaults to false.

Returns
A new memory descriptor with new dimensions.

desc **permute_axes**(const std::vector<int> &permutation, bool allow_empty = false) const

Constructs a memory descriptor by permuting axes in an existing one.

The physical memory layout representation is adjusted accordingly to maintain the consistency between the logical and physical parts of the memory descriptor. The new memory descriptor inherits the data type.

The logical axes will be permuted in the following manner:

```
for (i = 0; i < ndims(); i++)
    new_desc.dims()[permutation[i]] = dims()[i];
```

Example:

```
std::vector<int> permutation = {1, 0}; // swap the first and
                                       // the second axes
dnnl::memory::desc in_md(
        {2, 3}, data_type, memory::format_tag::ab);
dnnl::memory::desc expect_out_md(
        {3, 2}, data_type, memory::format_tag::ba);

assert(in_md.permute_axes(permutation) == expect_out_md);
```

Parameters

- **permutation** – Axes permutation.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case a zero memory descriptor will be returned. This flag is optional and defaults to false.

Returns
A new memory descriptor with new dimensions.

memory::*dims* **dims**() const

Returns dimensions of the memory descriptor.

Potentially expensive due to the data copy involved.

Returns
A copy of the dimensions vector.

memory::*data_type* **data_type**() const

Returns the data type of the memory descriptor.

Returns
The data type.

size_t **get_size**() const

Returns size of the memory descriptor in bytes.

Returns
The number of bytes required to allocate a memory buffer for the memory object described by this memory descriptor.

bool **is_zero**() const

Checks whether the memory descriptor is zero (empty).

Returns
`true` if the memory descriptor describes an empty memory and `false` otherwise.

bool **operator==**(const *desc* &other) const

An equality operator.

Parameters
other – Another memory descriptor.

Returns
Whether this and the other memory descriptors have the same format tag, dimensions, strides, etc.

bool **operator!=**(const *desc* &other) const

An inequality operator.

Parameters
other – Another memory descriptor.

Returns
Whether this and the other memory descriptors describe different memory.

Objects

Memory objects combine memory descriptors with storage for data (a data handle). With USM, the data handle is simply a pointer to `void`. The data handle can be queried using *dnnl::memory::get_data_handle()* and set using *dnnl::memory::set_data_handle()*. The underlying SYCL buffer, when used, can be queried using *dnnl::sycl_interop::get_buffer()* and set using *dnnl::sycl_interop::set_buffer()*. In addition, the memory descriptor and the engine underlying a memory object can be queried using *dnnl::memory::get_desc()* and *dnnl::memory::get_engine()* respectively.

API

struct **memory**

Memory object.

A memory object encapsulates a handle to a memory buffer allocated on a specific engine, tensor dimensions, data type, and memory format, which is the way tensor indices map to offsets in linear memory space. Memory objects are passed to primitives during execution.

Public Functions

memory()

Default constructor.

Constructs an empty memory object, which can be used to indicate absence of a parameter.

memory(const *desc* &md, const *engine* &aengine, void *handle)

Constructs a memory object.

Unless `handle` is equal to *DNNL_MEMORY_NONE*, the constructed memory object will have the underlying buffer set. In this case, the buffer will be initialized as if *dnnl::memory::set_data_handle()* has been called.

See also:

memory::set_data_handle()

Parameters

- **md** – Memory descriptor.
- **aengine** – Engine to store the data on.
- **handle** – Handle of the memory buffer to use.
  - A pointer to the user-allocated buffer. In this case the library doesn't own the buffer.
  - The *DNNL_MEMORY_ALLOCATE* special value. Instructs the library to allocate the buffer for the memory object. In this case the library owns the buffer and the memory allocation kind of the underlying buffer is *dnnl::sycl_interop::memory_kind::usm*.
  - *DNNL_MEMORY_NONE* to create *dnnl::memory* without an underlying buffer.

memory(const *desc* &md, const *engine* &aengine)

Constructs a memory object.

The underlying buffer for the memory will be allocated by the library. The memory allocation kind of the underlying buffer is *dnnl::sycl_interop::memory_kind::usm*.

Parameters

- **md** – Memory descriptor.
- **aengine** – Engine to store the data on.

desc **get_desc**() const

Returns the associated memory descriptor.

engine **get_engine**() const

Returns the associated engine.

void ***get_data_handle**() const

Returns the underlying memory buffer.

On the CPU engine, or when using USM, this is a pointer to the allocated memory.

void **set_data_handle**(void *handle, const *stream* &astream) const

Sets the underlying memory buffer.

This function may write zero values to the memory specified by the `handle` if the memory object has a zero padding area. This may be time consuming and happens each time this function is called. The operation is always blocking and the stream parameter is a hint.

Note: Even when the memory object is used to hold values that stay constant during the execution of the program (pre-packed weights during inference, for example), the function will still write zeroes to the padding area if it exists. Hence, the `handle` parameter cannot and does not have a const qualifier.

Parameters

- **handle** – Memory buffer to use. On the CPU engine or when USM is used, the memory buffer is a pointer to the actual data. It must have at least *dnnl::memory::desc::get_size()* bytes allocated.
- **astream** – Stream to use to execute padding in.

void **set_data_handle**(void *handle) const

Sets the underlying memory buffer.

See documentation for *dnnl::memory::set_data_handle(void *, const stream &) const* for more information.

Parameters

handle – Memory buffer to use. For the CPU engine, the memory buffer is a pointer to the actual data. It must have at least *dnnl::memory::desc::get_size()* bytes allocated.

template<typename **T** = void>
T ***map_data**() const

Maps a memory object and returns a host-side pointer to a memory buffer with a copy of its contents.

Mapping enables read/write directly from/to the memory contents for engines that do not support direct memory access.

Mapping is an exclusive operation - a memory object cannot be used in other operations until it is unmapped via *dnnl::memory::unmap_data()* call.

Note: Any primitives working with the memory should be completed before the memory is mapped. Use *dnnl::stream::wait()* to synchronize the corresponding execution stream.

Note: The map_data and unmap_data functions are provided mainly for debug and testing purposes and their performance may be suboptimal.

Template Parameters

T – Data type to return a pointer to.

Returns

Pointer to the mapped memory.

void **unmap_data**(void *mapped_ptr) const

Unmaps a memory object and writes back any changes made to the previously mapped memory buffer.

Note: The map_data and unmap_data functions are provided mainly for debug and testing purposes and their performance may be suboptimal.

Parameters

mapped_ptr – A pointer previously returned by *dnnl::memory::map_data()*.

enum class dnnl::sycl_interop::**memory_kind**

Memory allocation kinds.

Values:

enumerator **usm**

USM memory allocation kind.

enumerator **buffer**

Buffer memory allocation kind.

memory dnnl::sycl_interop::**make_memory**(const *memory*::*desc* &adesc, const *engine* &aengine, *memory_kind* akind, void *ahandle = DNNL_MEMORY_ALLOCATE)

Creates a memory object of a specified description and of a specified memory allocation kind, for a specified engine.

Note: If `akind` is *dnnl::sycl_interop::memory_kind::buffer*, and `ahandle` is not *DNNL_MEMORY_ALLOCATE* or *DNNL_MEMORY_NONE*, an exception is thrown.

Parameters

- **adesc** – Memory descriptor that describes the data.
- **aengine** – Engine to store the data on.
- **akind** – Memory allocation kind.
- **ahandle** – Handle of the memory data to use. This parameter is optional. By default, the underlying memory buffer is allocated internally, its memory allocation kind is *dnnl::sycl_interop::memory_kind::usm*, and the library owns the buffer. If `handle` is provided, the library does not own the buffer.

Returns

Memory object described by `adesc` memory descriptor, which has `akind` memory allocation kind, and is attached to the `aengine` engine.

memory dnnl::sycl_interop::**make_memory**(const *memory*::*desc* &adesc, const *stream* &astream, *memory_kind* akind, void *ahandle = DNNL_MEMORY_ALLOCATE)

Creates a memory object of a specified description and of a specified memory allocation kind, for a specified stream.

Note: If `akind` is *dnnl::sycl_interop::memory_kind::buffer*, and `ahandle` is not *DNNL_MEMORY_ALLOCATE* or *DNNL_MEMORY_NONE*, an exception is thrown.

Parameters

- **adesc** – Memory descriptor that describes the data.
- **astream** – Stream object where the data is used.
- **akind** – Memory allocation kind.

- **ahandle** – Handle of the memory data to use. This parameter is optional. By default, the underlying memory buffer is allocated internally, its memory allocation kind is *dnnl::sycl_interop::memory_kind::usm*, and the library owns the buffer. If `handle` is provided, the library does not own the buffer.

Returns

Memory object described by `adesc` memory descriptor, which has `akind` memory allocation kind, and used withing the `astream` stream.

template<typename **T**, int **ndims**>
memory `dnnl::sycl_interop::`**`make_memory`**(const *memory*::*desc* &adesc, const *engine* &aengine,
cl::sycl::buffer<*T*, *ndims*> abuffer)

Creates a memory object using a specified SYCL buffer.

Note: When such memory object is created, it is implied that its memory allocation kind is *dnnl::sycl_interop::memory_kind::buffer*.

Template Parameters

- **T** – Data type of the specified SYCL buffer.
- **ndims** – Number of dimensions of the specified SYCL buffer.

Parameters

- **adesc** – Memory descriptor that describes the data within the specified buffer.
- **aengine** – Engine to store the data on.
- **abuffer** – SYCL buffer.

Returns

Memory object which holds a `abuffer` SYCL buffer described by the `adesc` memory descriptor and attached to the `aengine` engine.

template<typename **T**, int **ndims**>
memory `dnnl::sycl_interop::`**`make_memory`**(const *memory*::*desc* &adesc, const *stream* &astream,
cl::sycl::buffer<*T*, *ndims*> abuffer)

Creates a memory object using a specified SYCL buffer.

Note: When such memory object is created, it is implied that its memory allocation kind is *dnnl::sycl_interop::memory_kind::buffer*.

Template Parameters

- **T** – Data type of the specified SYCL buffer.
- **ndims** – Number of dimensions of the specified SYCL buffer.

Parameters

- **adesc** – Memory descriptor that describes the data within the specified buffer.
- **astream** – Stream object where the data is used.
- **abuffer** – SYCL buffer.

Returns
Memory object which holds a `abuffer` SYCL buffer described by the `adesc` memory descriptor and used within the `astream` stream.

memory_kind `dnnl::sycl_interop::get_memory_kind`(const *memory* &amemory)

Returns the memory allocation kind of a specified memory object.

Note: The memory allocation kind of a memory object could be changed during its lifetime, by setting the USM handle or SYCL buffer of said memory object.

Parameters
amemory – Memory object.

Returns
Memory allocation kind of the `amemory` memory object.

template<typename **T**, int **ndims**>
void `dnnl::sycl_interop::set_buffer`(*memory* &amemory, cl::sycl::buffer<*T*, *ndims*> abuffer)

Sets the SYCL buffer underlying a specified memory object.

Note: By setting the SYCL buffer of a memory object its memory allocation kind will be changed to *dnnl::sycl_interop::memory_kind::buffer*.

Template Parameters

- **T** – Data type of the specified SYCL buffer.
- **ndims** – Number of dimensions of the specified SYCL buffer.

Parameters

- **amemory** – Memory object that will store the `abuffer` SYCL buffer.
- **abuffer** – SYCL buffer to be stored in the `amemory` memory object.

template<typename **T**, int **ndims**>
void `dnnl::sycl_interop::set_buffer`(*memory* &amemory, cl::sycl::buffer<*T*, *ndims*> abuffer, *stream* &astream)

Sets the SYCL buffer underlying a specified memory object in a specified stream.

Template Parameters

- **T** – Data type of the specified SYCL buffer.
- **ndims** – Number of dimensions of the specified SYCL buffer.

Parameters

- **amemory** – Memory object that will store the `abuffer` SYCL buffer.
- **abuffer** – SYCL buffer to be stored in the `amemory` memory object and used in the `astream` stream.
- **astream** – Stream object within which the `amemory` memory object is used.

template<typename **T**, int **ndims** = 1>

cl::sycl::buffer<*T*, *ndims*> dnnl::sycl_interop::**get_buffer**(const *memory* &amemory)

Returns the SYCL buffer underlying a specified memory object.

Template Parameters

- **T** – Data type of the specified SYCL buffer.
- **ndims** – Number of dimensions of the specified buffer.

Parameters

amemory – Memory object.

Returns

SYCL buffer of type T with ndims dimensions, underlying the amemory memory object.

DNNL_MEMORY_NONE

Special pointer value that indicates that a memory object should not have an underlying buffer.

DNNL_MEMORY_ALLOCATE

Special pointer value that indicates that the library needs to allocate an underlying buffer for a memory object.

4.5 Primitives

Primitives are functor objects that encapsulate a particular computation such as forward convolution, backward LSTM computations, or a data transformation operation. A single primitive can sometimes represent more complex fused computations such as a forward convolution followed by a ReLU.

The most important difference between a primitive and a pure function is that a primitive can store state.

One part of the primitive's state is immutable. For example, convolution primitives store parameters like tensor shapes and can pre-compute other dependent parameters like cache blocking. This approach allows oneDNN primitives to pre-generate code specifically tailored for the operation to be performed. The oneDNN programming model assumes that the time it takes to perform the pre-computations is amortized by reusing the same primitive to perform computations multiple times.

The mutable part of the primitive's state is referred to as a scratchpad. It is a memory buffer that a primitive may use for temporary storage only during computations. The scratchpad can either be owned by a primitive object (which makes that object non-thread safe) or be an execution-time parameter.

Conceptually, oneDNN establishes several layers of how to describe a computation from more abstract to more concrete:

- Primitives descriptors fully defines an operations's computation using the memory descriptors (*dnnl::memory::desc*) passed at construction, as well as the attributes. It also dispatches specific implementation based on the engine. Primitive descriptors can be used to query various primitive implementation details and, for example, to implement *memory format propagation* by inspecting expected memory formats via queries without having to fully instantiate a primitive. oneDNN may contain multiple implementations for the same primitive that can be used to perform the same particular computation. Primitive descriptors allow one-way iteration which allows inspecting multiple implementations. The library is expected to order the implementations from most to least preferred, so it should always be safe to use the one that is chosen by default.
- Primitives, which are the most concrete, and embody the actual executable code that will be run to perform the primitive computation.

On the API level:

- Primitives are represented as a class on the top level of the dnnl namespace that have *dnnl::primitive* as their base class, for example *dnnl::convolution_forward*

- Primitive descriptors are represented as classes named `primitive_desc` and nested within the corresponding primitive classes that have *dnnl::primitive_desc_base* as their base class (except for RNN primitives that derive from *dnnl::rnn_primitive_desc_base*), for example *dnnl::convolution_forward::primitive_desc*. The *dnnl::primitive_desc::next_impl()* member function provides a way to iterate over implementations.

```
namespace dnnl {
   struct something_forward : public primitive {
      struct desc {
         // Primitive-specific constructors.
      }
      struct primitive_desc : public primitive_desc_base {
         // Constructors and primitive-specific memory descriptor queries.
      }
   };
}
```

The sequence of actions to create a primitive is:

1. Create a primitive descriptor with the proper memory descriptors, engine and attributes. The primitive descriptor can contain memory descriptors with placeholder *dnnl::memory::format_tag::any* memory formats if the primitive supports it.
2. Create a primitive based on the primitive descriptor obtained in step 1.

4.5.1 Common Definitions

This section lists common types and definitions used by all or multiple primitives.

Base Class for Primitives

struct **primitive**

Base class for all computational primitives.

Subclassed by dnnl::augru_backward, dnnl::augru_forward, *dnnl::batch_normalization_backward*, *dnnl::batch_normalization_forward*, *dnnl::binary*, *dnnl::concat*, *dnnl::convolution_backward_data*, *dnnl::convolution_backward_weights*, *dnnl::convolution_forward*, *dnnl::deconvolution_backward_data*, *dnnl::deconvolution_backward_weights*, *dnnl::deconvolution_forward*, *dnnl::eltwise_backward*, *dnnl::eltwise_forward*, *dnnl::gru_backward*, *dnnl::gru_forward*, *dnnl::inner_product_backward_data*, *dnnl::inner_product_backward_weights*, *dnnl::inner_product_forward*, *dnnl::layer_normalization_backward*, *dnnl::layer_normalization_forward*, dnnl::lbr_augru_backward, dnnl::lbr_augru_forward, *dnnl::lbr_gru_backward*, *dnnl::lbr_gru_forward*, *dnnl::lrn_backward*, *dnnl::lrn_forward*, *dnnl::lstm_backward*, *dnnl::lstm_forward*, *dnnl::matmul*, *dnnl::pooling_backward*, *dnnl::pooling_forward*, *dnnl::prelu_backward*, *dnnl::prelu_forward*, *dnnl::reduction*, *dnnl::reorder*, *dnnl::resampling_backward*, *dnnl::resampling_forward*, *dnnl::shuffle_backward*, *dnnl::shuffle_forward*, *dnnl::softmax_backward*, *dnnl::softmax_forward*, *dnnl::sum*, *dnnl::vanilla_rnn_backward*, *dnnl::vanilla_rnn_forward*

Public Types

enum class **kind**

Kinds of primitives supported by the library.

Values:

enumerator **undef**

Undefined primitive.

enumerator **reorder**

A reorder primitive.

enumerator **shuffle**

A shuffle primitive.

enumerator **concat**

A (out-of-place) tensor concatenation primitive.

enumerator **sum**

A summation primitive.

enumerator **convolution**

A convolution primitive.

enumerator **deconvolution**

A deconvolution primitive.

enumerator **eltwise**

An element-wise primitive.

enumerator **softmax**

A softmax primitive.

enumerator **pooling**

A pooling primitive.

enumerator **prelu**

A PReLU primitive.

enumerator **lrn**

An LRN primitive.

enumerator **batch_normalization**

A batch normalization primitive.

enumerator **layer_normalization**

A layer normalization primitive.

enumerator **inner_product**

An inner product primitive.

enumerator **rnn**

An RNN primitive.

enumerator **binary**

A binary primitive.

enumerator **matmul**

A matmul (matrix multiplication) primitive.

enumerator **resampling**

A resampling primitive.

Public Functions

primitive()

Default constructor. Constructs an empty object.

primitive(const *primitive_desc_base* &pd)

Constructs a primitive from a primitive descriptor.

Parameters
pd – Primitive descriptor.

inline *kind* **get_kind**() const

Returns the kind of the primitive.

Returns
The primitive kind.

void **execute**(const *stream* &astream, const std::unordered_map<int, *memory*> &args) const

Executes computations specified by the primitive in a specified stream.

Arguments are passed via an arguments map containing <index, memory object> pairs. The index must be one of the DNNL_ARG_* values such as DNNL_ARG_SRC, and the memory must have a memory descriptor matching the one returned by *dnnl::primitive_desc_base::query_md*(query::exec_arg_md, index) unless using dynamic shapes (see *DNNL_RUNTIME_DIM_VAL*).

Parameters

- **astream** – Stream object. The stream must belong to the same engine as the primitive.
- **args** – Arguments map.

primitive &**operator=**(const *primitive* &rhs)

Assignment operator.

cl::sycl::event **dnnl::sycl_interop::execute**(const *primitive* &aprimitive, const *stream* &astream, const std::unordered_map<int, *memory*> &args, const std::vector<cl::sycl::event> &dependencies = {})

Executes computations using a specified primitive object in a specified stream.

Arguments are passed via an arguments map containing <index, memory object> pairs. The index must be one of the DNNL_ARG_* values such as DNNL_ARG_SRC, and the memory must have a memory descriptor matching the one returned by *dnnl::primitive_desc_base::query_md*(query::exec_arg_md, index) unless using dynamic shapes (see *DNNL_RUNTIME_DIM_VAL*).

Parameters

- **aprimitive** – Primitive to be executed.
- **astream** – Stream object. The stream must belong to the same engine as the primitive.
- **args** – Arguments map.
- **dependencies** – Vector of SYCL events that the execution depends on.

Returns

SYCL event object for the specified primitive execution.

Base Class for Primitives Descriptors

There is a common base class for primitive descriptors.

struct **primitive_desc_base**

Base class for all primitive descriptors.

Subclassed by *dnnl::concat::primitive_desc*, *dnnl::primitive_desc*, *dnnl::reorder::primitive_desc*, *dnnl::sum::primitive_desc*

Public Functions

primitive_desc_base()

Default constructor. Produces an empty object.

engine **get_engine**() const

Returns the engine of the primitive descriptor.

Returns

The engine of the primitive descriptor.

const char ***impl_info_str**() const

Returns implementation name.

Returns

The implementation name.

memory::*dim* **query_s64**(query what) const

Returns a *memory::dim* value (same as int64_t).

Parameters

what – The value to query.

Returns

The result of the query.

memory::*dims* **get_strides**() const

Returns strides.

Returns
Strides.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a strides parameter.

memory::*dims* **get_dilations**() const

Returns dilations.

Returns
Dilations.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a dilations parameter.

memory::*dims* **get_padding_l**() const

Returns a left padding.

Returns
A left padding.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a left padding parameter.

memory::*dims* **get_padding_r**() const

Returns a right padding.

Returns
A right padding.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a right padding parameter.

float **get_epsilon**() const

Returns an epsilon.

Returns
An epsilon.

Returns
Zero if the primitive does not have an epsilon parameter.

template<typename **T** = unsigned>
T **get_flags**() const

Returns flags.

Template Parameters
T – Flags enumeration type.

Returns
Flags.

Returns
Zero if the primitive does not have a flags parameter.

dnnl::*algorithm* **get_algorithm**() const

Returns an algorithm kind.

Returns
An algorithm kind.

Returns
dnnl::algorithm::undef if the primitive does not have an algorithm parameter.

float **get_alpha**() const

Returns an alpha.

Returns
An alpha.

Returns
Zero if the primitive does not have an alpha parameter.

float **get_beta**() const

Returns a beta.

Returns
A beta.

Returns
Zero if the primitive does not have a beta parameter.

int **get_axis**() const

Returns an axis.

Returns
An axis.

Returns
A negative number if the primitive does not have an axis parameter.

memory::*dim* **get_local_size**() const

Returns an LRN local size parameter.

Returns
An LRN local size parameter.

Returns
Zero if the primitive does not have an LRN local size parameter.

float **get_k**() const

Returns an LRN K parameter.

Returns
An LRN K parameter.

Returns
Zero if the primitive does not have an LRN K parameter.

float **get_p**() const

Returns a reduction P parameter.

Returns
A reduction P parameter.

Returns
Zero if the primitive does not have a reduction P parameter.

std::vector<float> **get_factors**() const

Returns a resampling factors parameters.

Returns
A vector of factors.

Returns
An empty vector if the primitive does not have a resampling factors parameter.

dnnl::*algorithm* **get_cell_kind**() const

Returns an RNN cell kind parameter.

Returns
An RNN cell kind parameter.

Returns
dnnl::algorithm::undef if the primitive does not have an RNN cell kind parameter.

dnnl::*rnn_direction* **get_direction**() const

Returns an RNN direction parameter.

Returns
An RNN direction parameter.

Returns
dnnl::rnn_direction::undef if the primitive does not have an RNN direction parameter.

dnnl::*algorithm* **get_activation_kind**() const

Returns an RNN activation kind parameter.

Returns
An RNN activation kind parameter.

Returns
dnnl::algorithm::undef if the primitive does not have an RNN activation kind parameter.

memory::*dims* **get_kernel**() const

Returns a pooling kernel parameter.

Returns
A pooling kernel parameter.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a pooling kernel parameter.

memory::*dim* **get_group_size**() const

Returns a shuffle group size parameter.

Returns
A shuffle group size parameter.

Returns
Zero if the primitive does not have a shuffle group size parameter.

dnnl::*prop_kind* **get_prop_kind**() const

Returns a propagation kind.

Returns
A propagation kind.

Returns
dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

memory::*desc* **query_md**(query what, int idx = 0) const

Returns a memory descriptor.

Note: There are also convenience methods *dnnl::primitive_desc_base::src_desc()*, *dnnl::primitive_desc_base::dst_desc()*, and others.

Parameters

- **what** – The kind of parameter to query; can be dnnl::query::src_md, dnnl::query::dst_md, etc.
- **idx** – Index of the parameter. For example, convolution bias can be queried with what = dnnl::query::weights_md and idx = 1.

Returns

The requested memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a parameter of the specified kind or index.

memory::*desc* **src_desc**(int idx) const

Returns a source memory descriptor.

Parameters

idx – Source index.

Returns

Source memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a source parameter with index `pdx`.

memory::*desc* **dst_desc**(int idx) const

Returns a destination memory descriptor.

Parameters

idx – Destination index.

Returns

Destination memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a destination parameter with index `pdx`.

memory::*desc* **bias_desc**() const

Returns the bias memory descriptor.

Returns

The bias memory descriptor.

Returns

A zero memory descriptor of the primitive does not have a bias parameter.

memory::*desc* **weights_desc**(int idx) const

Returns a weights memory descriptor.

Parameters
idx – Weights index.

Returns
Weights memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a weights parameter with index pdx.

memory::*desc* **diff_src_desc**(int idx) const

Returns a diff source memory descriptor.

Parameters
idx – Diff source index.

Returns
Diff source memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a diff source parameter with index pdx.

memory::*desc* **diff_dst_desc**(int idx) const

Returns a diff destination memory descriptor.

Parameters
idx – Diff destination index.

Returns
Diff destination memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a diff destination parameter with index pdx.

memory::*desc* **diff_weights_desc**(int idx) const

Returns a diff weights memory descriptor.

Parameters
idx – Diff weights index.

Returns
Diff weights memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a diff weights parameter with index pdx.

memory::*desc* **src_desc**() const

Returns a source memory descriptor.

Returns
Source memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **dst_desc**() const

Returns a destination memory descriptor.

Returns
Destination memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a destination parameter.

memory::*desc* **weights_desc**() const

Returns a weights memory descriptor.

Returns
Weights memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a weights parameter.

memory::*desc* **diff_src_desc**() const

Returns a diff source memory descriptor.

Returns
Diff source memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a diff source memory with.

memory::*desc* **diff_dst_desc**() const

Returns a diff destination memory descriptor.

Returns
Diff destination memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a diff destination parameter.

memory::*desc* **diff_weights_desc**() const

Returns a diff weights memory descriptor.

Returns
Diff weights memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a diff weights parameter.

memory::*desc* **workspace_desc**() const

Returns the workspace memory descriptor.

Returns
Workspace memory descriptor.

Returns
A zero memory descriptor if the primitive does not require workspace parameter.

memory::*desc* **scratchpad_desc**() const

Returns the scratchpad memory descriptor.

Returns
scratchpad memory descriptor.

Returns
A zero memory descriptor if the primitive does not require scratchpad parameter.

engine **scratchpad_engine**() const

Returns the engine on which the scratchpad memory is located.

Returns
The engine on which the scratchpad memory is located.

primitive_attr **get_primitive_attr**() const

Returns the primitive attributes.

Returns
The primitive attributes.

dnnl::*primitive*::*kind* **get_kind**() const

Returns the kind of the primitive descriptor.

Returns
The kind of the primitive descriptor.

It is further derived from to provide base class for all primitives.

struct **primitive_desc** : public dnnl::*primitive_desc_base*

A base class for descriptors of all primitives that have an operation descriptor and that support iteration over multiple implementations.

Subclassed by *dnnl::batch_normalization_backward::primitive_desc*, *dnnl::batch_normalization_forward::primitive_desc*, *dnnl::binary::primitive_desc*, *dnnl::convolution_backward_data::primitive_desc*, *dnnl::convolution_backward_weights::primitive_desc*, *dnnl::convolution_forward::primitive_desc*, *dnnl::deconvolution_backward_data::primitive_desc*, *dnnl::deconvolution_backward_weights::primitive_desc*, *dnnl::deconvolution_forward::primitive_desc*, *dnnl::eltwise_backward::primitive_desc*, *dnnl::eltwise_forward::primitive_desc*, *dnnl::inner_product_backward_data::primitive_desc*, *dnnl::inner_product_backward_weights::primitive_desc*, *dnnl::inner_product_forward::primitive_desc*, *dnnl::layer_normalization_backward::primitive_desc*, *dnnl::layer_normalization_forward::primitive_desc*, *dnnl::lrn_backward::primitive_desc*, *dnnl::lrn_forward::primitive_desc*, *dnnl::matmul::primitive_desc*, *dnnl::pooling_backward::primitive_desc*, *dnnl::pooling_forward::primitive_desc*, *dnnl::prelu_backward::primitive_desc*, *dnnl::prelu_forward::primitive_desc*, *dnnl::reduction::primitive_desc*, *dnnl::resampling_backward::primitive_desc*, *dnnl::resampling_forward::primitive_desc*, *dnnl::rnn_primitive_desc_base*, *dnnl::shuffle_backward::primitive_desc*, *dnnl::shuffle_forward::primitive_desc*, *dnnl::softmax_backward::primitive_desc*, *dnnl::softmax_forward::primitive_desc*

Public Functions

primitive_desc()

Default constructor. Produces an empty object.

bool **next_impl**()

Advances the primitive descriptor iterator to the next implementation.

Returns
`true` on success, and `false` if the last implementation reached, in which case primitive descriptor is not modified.

The *dnnl::reorder*, *dnnl::sum* and *dnnl::concat* primitives also subclass *dnnl::primitive_desc* to implement their primitive descriptors.

RNN primitives further subclass the *dnnl::primitive_desc_base* to provide utility functions for frequently queried memory descriptors.

struct **rnn_primitive_desc_base** : public dnnl::*primitive_desc*

Base class for primitive descriptors for RNN primitives.

Subclassed by dnnl::augru_backward::primitive_desc, dnnl::augru_forward::primitive_desc, *dnnl::gru_backward::primitive_desc*, *dnnl::gru_forward::primitive_desc*, dnnl::lbr_augru_backward::primitive_desc,

dnnl::lbr_augru_forward::primitive_desc, *dnnl::lbr_gru_backward::primitive_desc*, *dnnl::lbr_gru_forward::primitive_desc*, *dnnl::lstm_backward::primitive_desc*, *dnnl::lstm_forward::primitive_desc*, *dnnl::vanilla_rnn_backward::primitive_desc*, *dnnl::vanilla_rnn_forward::primitive_desc*

Public Functions

rnn_primitive_desc_base()

Default constructor. Produces an empty object.

memory::*desc* **src_layer_desc**() const

Returns source layer memory descriptor.

Returns
Source layer memory descriptor.

memory::*desc* **src_iter_desc**() const

Returns source iteration memory descriptor.

Returns
Source iteration memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a source iteration parameter.

memory::*desc* **src_iter_c_desc**() const

Returns source recurrent cell state memory descriptor.

Returns
Source recurrent cell state memory descriptor.

memory::*desc* **weights_layer_desc**() const

Returns weights layer memory descriptor.

Returns
Weights layer memory descriptor.

memory::*desc* **weights_iter_desc**() const

Returns weights iteration memory descriptor.

Returns
Weights iteration memory descriptor.

memory::*desc* **weights_peephole_desc**() const

Returns weights peephole memory descriptor.

Returns
Weights peephole memory descriptor.

memory::*desc* **weights_projection_desc**() const

Returns weights projection memory descriptor.

Returns
Weights projection memory descriptor.

memory::*desc* **augru_attention_desc**() const

Returns AUGRU attention memory descriptor.

Returns
AUGRU attention memory descriptor.

memory::*desc* **bias_desc**() const

Returns bias memory descriptor.

Returns
Bias memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a bias parameter.

memory::*desc* **dst_layer_desc**() const

Returns destination layer memory descriptor.

Returns
Destination layer memory descriptor.

memory::*desc* **dst_iter_desc**() const

Returns destination iteration memory descriptor.

Returns
Destination iteration memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a destination iteration parameter.

memory::*desc* **dst_iter_c_desc**() const

Returns destination recurrent cell state memory descriptor.

Returns
Destination recurrent cell state memory descriptor.

memory::*desc* **diff_src_layer_desc**() const

Returns diff source layer memory descriptor.

Returns
Diff source layer memory descriptor.

memory::*desc* **diff_src_iter_desc**() const

Returns diff source iteration memory descriptor.

Returns
Diff source iteration memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a diff source iteration parameter.

memory::*desc* **diff_src_iter_c_desc**() const

Returns diff source recurrent cell state memory descriptor.

Returns
Diff source recurrent cell state memory descriptor.

memory::*desc* **diff_weights_layer_desc**() const

Returns diff weights layer memory descriptor.

Returns
Diff weights layer memory descriptor.

memory::*desc* **diff_weights_iter_desc**() const

Returns diff weights iteration memory descriptor.

Returns
Diff weights iteration memory descriptor.

memory::*desc* **diff_weights_peephole_desc**() const

Returns diff weights peephole memory descriptor.

Returns
Diff weights peephole memory descriptor.

memory::*desc* **diff_weights_projection_desc**() const

Returns diff weights projection memory descriptor.

Returns
Diff weights projection memory descriptor.

memory::*desc* **diff_augru_attention_desc**() const

Returns diff AUGRU attention memory descriptor.

Returns
Diff AUGRU attention memory descriptor.

memory::*desc* **diff_bias_desc**() const

Returns diff bias memory descriptor.

Returns
Diff bias memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a diff bias parameter.

memory::*desc* **diff_dst_layer_desc**() const

Returns diff destination layer memory descriptor.

Returns
Diff destination layer memory descriptor.

memory::*desc* **diff_dst_iter_desc**() const

Returns diff destination iteration memory descriptor.

Returns
Diff destination iteration memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a diff destination iteration parameter.

memory::*desc* **diff_dst_iter_c_desc**() const

Returns diff destination recurrent cell state memory descriptor.

Returns
Diff destination recurrent cell state memory descriptor.

Common Enumerations

enum class dnnl::**prop_kind**

Propagation kind.

Values:

enumerator **undef**

Undefined propagation kind.

enumerator **forward_training**

Forward data propagation (training mode). In this mode, primitives perform computations necessary for subsequent backward propagation.

enumerator **forward_inference**

Forward data propagation (inference mode). In this mode, primitives perform only computations that are necessary for inference and omit computations that are necessary only for backward propagation.

enumerator **forward_scoring**

Forward data propagation, alias for *dnnl::prop_kind::forward_inference*.

enumerator **forward**

Forward data propagation, alias for *dnnl::prop_kind::forward_training*.

enumerator **backward**

Backward propagation (with respect to all parameters).

enumerator **backward_data**

Backward data propagation.

enumerator **backward_weights**

Backward weights propagation.

enumerator **backward_bias**

Backward bias propagation.

enum class `dnnl::algorithm`

Kinds of algorithms.

Values:

enumerator **undef**

Undefined algorithm.

enumerator **convolution_auto**

Convolution algorithm that is chosen to be either direct or Winograd automatically

enumerator **convolution_direct**

Direct convolution.

enumerator **convolution_winograd**

Winograd convolution.

enumerator **deconvolution_direct**

Direct deconvolution.

enumerator **deconvolution_winograd**

Winograd deconvolution.

enumerator **eltwise_abs**

Elementwise: abs.

enumerator **eltwise_bounded_relu**

Elementwise: bounded_relu.

enumerator **eltwise_clip**

Elementwise: clip.

enumerator **eltwise_clip_use_dst_for_bwd**

Elementwise: clip (dst for backward)

enumerator **eltwise_elu**

Elementwise: exponential linear unit (ELU)

enumerator **eltwise_elu_use_dst_for_bwd**

Elementwise: exponential linear unit (ELU) (dst for backward)

enumerator **eltwise_exp**

Elementwise: exponent.

enumerator **eltwise_exp_use_dst_for_bwd**

Elementwise: exponent (dst for backward)

enumerator **eltwise_gelu**

Elementwise: gelu alias for *dnnl::algorithm::eltwise_gelu_tanh*

enumerator **eltwise_gelu_tanh**

Elementwise: tanh-based gelu.

enumerator **eltwise_gelu_erf**

Elementwise: erf-based gelu.

enumerator **eltwise_hardswish**

Elementwise: hardswish.

enumerator **eltwise_hardsigmoid**

Elementwise: hardsigmoid.

enumerator **eltwise_linear**

Elementwise: linear.

enumerator **eltwise_log**

Elementwise: natural logarithm.

enumerator **eltwise_logistic**

Elementwise: logistic.

enumerator **eltwise_logistic_use_dst_for_bwd**

Elementwise: logistic (dst for backward)

enumerator **eltwise_mish**

Elementwise: mish.

enumerator **eltwise_pow**

Elementwise: pow.

enumerator **eltwise_relu**

Elementwise: rectified linear unit (ReLU)

enumerator **eltwise_relu_use_dst_for_bwd**

Elementwise: rectified linear unit (ReLU) (dst for backward)

enumerator **eltwise_round**

Elementwise: round.

enumerator **eltwise_soft_relu**

Elementwise: soft_relu.

enumerator **eltwise_sqrt**

Elementwise: square root.

enumerator **eltwise_sqrt_use_dst_for_bwd**

Elementwise: square root (dst for backward)

enumerator **eltwise_square**

Elementwise: square.

enumerator **eltwise_swish**

Elementwise: swish ($x \cdot sigmoid(a \cdot x)$)

enumerator **eltwise_tanh**

Elementwise: hyperbolic tangent non-linearity (tanh)

enumerator **eltwise_tanh_use_dst_for_bwd**

Elementwise: hyperbolic tangent non-linearity (tanh) (dst for backward)

enumerator **lrn_across_channels**

Local response normalization (LRN) across multiple channels.

enumerator **lrn_within_channel**

LRN within a single channel.

enumerator **pooling_max**

Max pooling.

enumerator **pooling_avg**

Average pooling exclude padding, alias for *dnnl::algorithm::pooling_avg_include_padding*

enumerator **pooling_avg_include_padding**

Average pooling include padding.

enumerator **pooling_avg_exclude_padding**

Average pooling exclude padding.

enumerator **vanilla_rnn**

RNN cell.

enumerator **vanilla_lstm**

LSTM cell.

enumerator **vanilla_gru**

GRU cell.

enumerator **lbr_gru**

GRU cell with linear before reset. Differs from original GRU in how the new memory gate is calculated: $c_t = tanh(W_c * x_t + b_{c_x} + r_t * (U_c * h_{t-1} + b_{c_h}))$ LRB GRU expects 4 bias tensors on input: $[b_u, b_r, b_{c_x}, b_{c_h}]$

enumerator **binary_add**

Binary add.

enumerator **binary_mul**

Binary mul.

enumerator **binary_max**

Binary max.

enumerator **binary_min**

Binary min.

enumerator **binary_div**

Binary div.

enumerator **binary_sub**

Binary sub.

enumerator **binary_ge**

Binary greater than or equal.

enumerator **binary_gt**

Binary greater than.

enumerator **binary_le**

Binary less than or equal.

enumerator **binary_lt**

Binary less than.

enumerator **binary_eq**

Binary equal.

enumerator **binary_ne**

Binary not equal.

enumerator **resampling_nearest**

Nearest Neighbor resampling method.

enumerator **resampling_linear**

Linear (Bilinear, Trilinear) resampling method.

enumerator **reduction_max**

Reduction using max operation.

enumerator **reduction_min**

Reduction using min operation.

enumerator **reduction_sum**

Reduction using sum operation.

enumerator **reduction_mul**

Reduction using mul operation.

enumerator **reduction_mean**

Reduction using mean operation.

enumerator **reduction_norm_lp_max**

Reduction using norm_lp_max operation.

enumerator **reduction_norm_lp_sum**

Reduction using norm_lp_sum operation.

enumerator **reduction_norm_lp_power_p_max**

Reduction using norm_lp_power_p_max operation.

enumerator **reduction_norm_lp_power_p_sum**

Reduction using norm_lp_power_p_sum operation.

enumerator **softmax_accurate**

Softmax, numerically stable.

enumerator **softmax_log**

LogSoftmax, numerically stable.

Normalization Primitives Flags

enum class dnnl::**normalization_flags** : unsigned

Flags for normalization primitives (can be combined via '|')

Values:

enumerator **none**

Use no normalization flags. If specified, the library computes mean and variance on forward propagation for training and inference, outputs them on forward propagation for training, and computes the respective derivatives on backward propagation.

enumerator **use_global_stats**

Use global statistics. If specified, the library uses mean and variance provided by the user as an input on forward propagation and does not compute their derivatives on backward propagation. Otherwise, the library computes mean and variance on forward propagation for training and inference, outputs them on forward propagation for training, and computes the respective derivatives on backward propagation.

enumerator **use_scale**

Use scale and shift parameters. If specified, the user is expected to pass scale and shift as inputs on forward propagation. On backward propagation of type *dnnl::prop_kind::backward*, the library computes their derivatives. If not specified, the scale and shift parameters are not used by the library in any way.

enumerator **use_shift**

enumerator **fuse_norm_relu**

Fuse normalization with ReLU. On training, normalization will require the workspace to implement backward propagation. On inference, the workspace is not required and behavior is the same as when normalization is fused with ReLU using the post-ops API.

Execution argument indices

DNNL_ARG_SRC_0

Source argument #0.

DNNL_ARG_SRC

A special mnemonic for source argument for primitives that have a single source. An alias for *DNNL_ARG_SRC_0*.

DNNL_ARG_SRC_LAYER

A special mnemonic for RNN input vector. An alias for *DNNL_ARG_SRC_0*.

DNNL_ARG_FROM

A special mnemonic for reorder source argument. An alias for *DNNL_ARG_SRC_0*.

DNNL_ARG_SRC_1

Source argument #1.

DNNL_ARG_SRC_ITER

A special mnemonic for RNN input recurrent hidden state vector. An alias for *DNNL_ARG_SRC_1*.

DNNL_ARG_SRC_2

Source argument #2.

DNNL_ARG_SRC_ITER_C

A special mnemonic for RNN input recurrent cell state vector. An alias for *DNNL_ARG_SRC_2*.

DNNL_ARG_DST_0

Destination argument #0.

DNNL_ARG_DST

A special mnemonic for destination argument for primitives that have a single destination. An alias for *DNNL_ARG_DST_0*.

DNNL_ARG_TO

A special mnemonic for reorder destination argument. An alias for *DNNL_ARG_DST_0*.

DNNL_ARG_DST_LAYER

A special mnemonic for RNN output vector. An alias for *DNNL_ARG_DST_0*.

DNNL_ARG_DST_1

Destination argument #1.

DNNL_ARG_DST_ITER

A special mnemonic for RNN input recurrent hidden state vector. An alias for *DNNL_ARG_DST_1*.

DNNL_ARG_DST_2

Destination argument #2.

DNNL_ARG_DST_ITER_C

A special mnemonic for LSTM output recurrent cell state vector. An alias for *DNNL_ARG_DST_2*.

DNNL_ARG_WEIGHTS_0

Weights argument #0.

DNNL_ARG_WEIGHTS

A special mnemonic for primitives that have a single weights argument. Alias for *DNNL_ARG_WEIGHTS_0*.

DNNL_ARG_SCALE

Scale values argument of normalization primitives.

DNNL_ARG_SHIFT

Shift values argument of normalization primitives.

DNNL_ARG_WEIGHTS_LAYER

A special mnemonic for RNN weights applied to the layer input. An alias for *DNNL_ARG_WEIGHTS_0*.

DNNL_ARG_WEIGHTS_1

Weights argument #1.

DNNL_ARG_WEIGHTS_ITER

A special mnemonic for RNN weights applied to the recurrent input. An alias for *DNNL_ARG_WEIGHTS_1*.

DNNL_ARG_BIAS

Bias tensor argument.

DNNL_ARG_MEAN

Mean values tensor argument.

DNNL_ARG_VARIANCE

Variance values tensor argument.

DNNL_ARG_WORKSPACE

Workspace tensor argument. Workspace is used to pass information from forward propagation to backward propagation computations.

DNNL_ARG_SCRATCHPAD

Scratchpad (temporary storage) tensor argument.

DNNL_ARG_DIFF_SRC_0

Gradient (diff) of the source argument #0.

DNNL_ARG_DIFF_SRC

A special mnemonic for primitives that have a single diff source argument. An alias for *DNNL_ARG_DIFF_SRC_0*.

DNNL_ARG_DIFF_SRC_LAYER

A special mnemonic for gradient (diff) of RNN input vector. An alias for *DNNL_ARG_DIFF_SRC_0*.

DNNL_ARG_DIFF_SRC_1

Gradient (diff) of the source argument #1.

DNNL_ARG_DIFF_SRC_ITER

A special mnemonic for gradient (diff) of RNN input recurrent hidden state vector. An alias for *DNNL_ARG_DIFF_SRC_1*.

DNNL_ARG_DIFF_SRC_2

Gradient (diff) of the source argument #2.

DNNL_ARG_DIFF_SRC_ITER_C

A special mnemonic for gradient (diff) of RNN input recurrent cell state vector. An alias for *DNNL_ARG_DIFF_SRC_1*.

DNNL_ARG_DIFF_DST_0

Gradient (diff) of the destination argument #0.

DNNL_ARG_DIFF_DST

A special mnemonic for primitives that have a single diff destination argument. An alias for *DNNL_ARG_DIFF_DST_0*.

DNNL_ARG_DIFF_DST_LAYER

A special mnemonic for gradient (diff) of RNN output vector. An alias for *DNNL_ARG_DIFF_DST_0*.

DNNL_ARG_DIFF_DST_1

Gradient (diff) of the destination argument #1.

DNNL_ARG_DIFF_DST_ITER

A special mnemonic for gradient (diff) of RNN input recurrent hidden state vector. An alias for *DNNL_ARG_DIFF_DST_1*.

DNNL_ARG_DIFF_DST_2

Gradient (diff) of the destination argument #2.

DNNL_ARG_DIFF_DST_ITER_C

A special mnemonic for gradient (diff) of RNN input recurrent cell state vector. An alias for *DNNL_ARG_DIFF_DST_2*.

DNNL_ARG_DIFF_WEIGHTS_0

Gradient (diff) of the weights argument #0.

DNNL_ARG_DIFF_WEIGHTS

A special mnemonic for primitives that have a single diff weights argument. Alias for *DNNL_ARG_DIFF_WEIGHTS_0*.

DNNL_ARG_DIFF_SCALE

Scale gradient argument of normalization primitives.

DNNL_ARG_DIFF_SHIFT

Shift gradient argument of normalization primitives.

DNNL_ARG_DIFF_WEIGHTS_LAYER

A special mnemonic for diff of RNN weights applied to the layer input. An alias for *DNNL_ARG_DIFF_WEIGHTS_0*.

DNNL_ARG_DIFF_WEIGHTS_1

Gradient (diff) of the weights argument #1.

DNNL_ARG_DIFF_WEIGHTS_ITER

A special mnemonic for diff of RNN weights applied to the recurrent input. An alias for *DNNL_ARG_DIFF_WEIGHTS_1*.

DNNL_ARG_DIFF_BIAS

Gradient (diff) of the bias tensor argument.

DNNL_ARG_MULTIPLE_SRC

Starting index for source arguments for primitives that take a variable number of source arguments.

DNNL_ARG_MULTIPLE_DST

Starting index for destination arguments for primitives that produce a variable number of destination arguments.

DNNL_ARG_ATTR_SCALES

Scaling factors provided at execution time.

DNNL_ARG_ATTR_ZERO_POINTS

Zero points provided at execution time.

DNNL_RUNTIME_DIM_VAL

A wildcard value for dimensions that are unknown at a primitive creation time.

DNNL_RUNTIME_SIZE_VAL

A `size_t` counterpart of the *DNNL_RUNTIME_DIM_VAL*. For instance, this value is returned by *dnnl::memory::desc::get_size()* if either of the dimensions or strides equal to *DNNL_RUNTIME_DIM_VAL*.

DNNL_RUNTIME_F32_VAL

A wildcard value for floating point values that are unknown at a primitive creation time.

DNNL_RUNTIME_S32_VAL

A wildcard value for int32_t values that are unknown at a primitive creation time.

4.5.2 Attributes

The parameters passed to create a primitive descriptor specify the basic problem description: the operation kind, the propagation kind, the input and output tensors descriptors (e.g. strides if applicable...), as well as the engine where the primitive will be executed.

Attributes specify some extra properties of the primitive. Users must create them before use and must set required specifics using the corresponding setters. The attributes are copied during primitive descriptor creation, so users can change or destroy attributes right after that.

If not modified, attributes can stay empty, which is equivalent to the default attributes. Primitive descriptors' constructors have empty attributes as default parameters, so, unless required, users can simply omit them.

Attributes can also contain *post-ops*, which are computations executed after the primitive.

Post-ops

Post-ops are operations that are appended after a primitive. They are implemented using the *Attributes* mechanism. If there are multiple post-ops, they are executed in the order they have been appended as follow:

$$\text{dst} = po[n](po[n-1](...(po[0](OP()))))$$

Note: Post-ops does not preserve intermediate data during computation. This typically makes them suitable for inference only.

The post-ops are represented by *dnnl::post_ops* which is copied once it is attached to the attributes using *dnnl::primitive_attr::set_post_ops()* function. The attributes then need to be passed to a primitive descriptor creation function to take effect. Below is a simple sketch:

```
dnnl::post_ops po; // default empty post-ops
assert(po.len() == 0); // no post-ops attached

po.append_SOMETHING(params); // append some particular post-op
po.append_SOMETHING_ELSE(other_params); // append one more post-op

// (!) Note that the order in which post-ops are appended matters!
assert(po.len() == 2);

dnnl::primitive_attr attr; // default attributes
attr.set_post_ops(po); // attach the post-ops to the attr
// any changes to po after this point don't affect the value stored in attr

primitive::primitive_desc op_pd(params, attr); // create a pd with the attr
```

Note: Different primitives may have different post-ops support. Moreover, the support might also depend on the actual implementation of a primitive. So robust code should be able to handle errors accordingly. See the *Attribute Related Error Handling*.

Note: Post-ops do not change memory format of the operation destination memory object.

The post-op objects can be inspected using the *dnnl::post_ops::kind()* function that takes an index of the post-op to inspect (that must be less than the value returned by *dnnl::post_ops::len()*), and returns its kind.

Supported Post-ops

Eltwise Post-op

The eltwise post-op is appended using *dnnl::post_ops::append_eltwise()* function. The *dnnl::post_ops::kind()* returns *dnnl::primitive::kind::eltwise* for such a post-op.

The eltwise post-op replaces:

$$\text{dst}[:] = \text{Op}(...)$$

with

$$\text{dst}[:] = scale \cdot \text{eltwise}(\text{Op}(...))$$

The intermediate result of the $\text{Op}(...)$ is not preserved.

The $scale$ factor is supported in *int8* inference only. For all other cases the scale must be *1.0* (default value). The scale parameter is set to 1.0 by default, and can be set using the *dnnl::primitive_attr::set_scales_mask()* attribute for the argument `DNNL_ARG_ATTR_MULTIPLE_POST_OP`.

Sum Post-op

The sum post-op accumulates the result of a primitive with the existing data and is appended using *dnnl::post_ops::append_sum()* function. The *dnnl::post_ops::kind()* returns *dnnl::primitive::kind::sum* for such a post-op.

Prior to accumulating the result, the existing value is multiplied by scale. The $scale$ factor is supported in *int8* inference only and should be used only when the result and the existing data have different magnitudes. For all other cases the scale must be *1.0* (default value). The scale parameter is set to 1.0 by default, and can be set using the *dnnl::primitive_attr::set_scales_mask()* attribute for the argument `DNNL_ARG_ATTR_MULTIPLE_POST_OP`.

Additionally, the sum post-op can reinterpret the destination values as a different data type of the same size. This may be used to, for example, reinterpret 8-bit signed data as unsigned or vice versa (which requires that values fall within a common range to work).

The sum post-op replaces

$$\text{dst}[:] = \text{Op}(...)$$

with

$$\text{dst}[:] = scale \cdot as_data_type(\text{dst}[:]) + \text{Op}(...)$$

Binary post-ops

The binary post-op replaces: .. math:

```
\dst[:] = \operatorname{Op}(...)
```

with

$$\text{dst}[:] = \text{binary}(\text{Op}(...), scale[:] \cdot Source\_1[:])$$

The binary post-op supports the same algorithms and broadcast semantic as the *binary primitive*.

Furthermore, the binary post-op scale parameter is set to 1.0 by default, and can be set using the *dnnl::primitive_attr::set_scales_mask()* attribute for the argument DNNL_ARG_ATTR_MULTIPLE_POST_OP | *DNNL_ARG_SRC_1*. For example:

```
primitive_attr attr;
post_ops p_ops;
p_ops.append_binary(algorithm::binary_add, summand_md);

attr.set_post_ops(p_ops);
attr.set_scales_mask(DNNL_ARG_ATTR_MULTIPLE_POST_OP(0) | DNNL_ARG_SRC_1,
        /* mask */ 0);
```

Examples of Chained Post-ops

Post-ops can be chained together by appending one after another. Note that the order matters: the post-ops are executed in the order they have been appended.

Sum -> ReLU

This pattern is pretty common for the CNN topologies of the ResNet family.

```
dnnl::post_ops po;
po.append_sum();
po.append_eltwise(
        /* algorithm = */ dnnl::algorithm::eltwise_relu,
        /* neg slope = */ 0.f,
        /* unused for ReLU */ 0.f);

dnnl::primitive_attr attr;
attr.set_post_ops(po);

convolution_forward::primitive_desc(conv_d, attr, engine);
```

This will lead to the following computations:

$$\text{dst}[:] = \text{ReLU}(\text{dst}[:] + \text{conv}(\text{src}[:], \text{weights}[:])$$

API

struct **post_ops**

Post-ops.

Post-ops are computations executed after the main primitive computations and are attached to the primitive via primitive attributes.

Public Functions

post_ops()

Constructs an empty sequence of post-ops.

int **len**() const

Returns the number of post-ops entries.

primitive::*kind* **kind**(int index) const

Returns the primitive kind of post-op at entry with a certain index.

Parameters
index – Index of the post-op to return the kind for.

Returns
Primitive kind of the post-op at the specified index.

void **append_sum**(*memory*::*data_type* data_type = *memory*::*data_type*::*undef*)

Appends an accumulation (sum) post-op. Prior to accumulating the result, the previous value would be multiplied by a scaling factor `scale` provided as execution argument.

The kind of this post-op is *dnnl::primitive::kind::sum*.

This feature may improve performance for cases like residual learning blocks, where the result of convolution is accumulated to the previously computed activations. The parameter `scale` may be used for the integer-based computations when the result and previous activations have different logical scaling factors.

In the simplest case when the accumulation is the only post-op, the computations would be `dst[:] := scale * dst[:] + op(...)` instead of `dst[:] := op(...)`.

If `data_type` is specified, the original dst tensor will be reinterpreted as a tensor with the provided data type. Because it is a reinterpretation, data_type and dst data type should have the same size. As a result, computations would be `dst[:] <- scale * as_data_type(dst[:]) + op(...)` instead of `dst[:] <- op(...)`.

Note: This post-op executes in-place and does not change the destination layout.

Parameters
data_type – Data type.

void **get_params_sum**(int index, float &scale) const

Returns the parameters of an accumulation (sum) post-op.

Parameters

- **index** – Index of the sum post-op.
- **scale** – Scaling factor of the sum post-op.

void **get_params_sum**(int index, float &scale, *memory*::*data_type* &data_type) const

Returns the parameters of an accumulation (sum) post-op.

Parameters

- **index** – Index of the sum post-op.
- **scale** – Scaling factor of the sum post-op.
- **data_type** – Data type of the sum post-op.

void **append_eltwise**(*algorithm* aalgorithm, float alpha, float beta)

Appends an elementwise post-op.

The kind of this post-op is *dnnl::primitive::kind::eltwise*.

In the simplest case when the elementwise is the only post-op, the computations would be `dst[:] := scale * eltwise_op (op(...))` instead of `dst[:] <- op(...)`, where eltwise_op is configured with the given parameters.

Parameters

- **aalgorithm** – Elementwise algorithm.
- **alpha** – Alpha parameter for the elementwise algorithm.
- **beta** – Beta parameter for the elementwise algorithm.

void **get_params_eltwise**(int index, *algorithm* &aalgorithm, float &alpha, float &beta) const

Returns parameters of an elementwise post-up.

Parameters

- **index** – Index of the post-op.
- **aalgorithm** – Output elementwise algorithm kind.
- **alpha** – Output alpha parameter for the elementwise algorithm.
- **beta** – Output beta parameter for the elementwise algorithm.

void **append_binary**(*algorithm* aalgorithm, const *memory*::*desc* &src1_desc)

Appends a binary post-op.

The kind of this post operation is *dnnl::primitive::kind::binary*.

In the simplest case when the binary is the only post operation, the computations would be:

```
dst[:] <- binary_op (dst[:], another_input[:])
```

where binary_op is configured with the given parameters. binary_op supports broadcast semantics for a second operand.

Parameters

- **aalgorithm** – Binary algorithm for the post-op.
- **src1_desc** – Memory descriptor of a second operand.

void **get_params_binary**(int index, *algorithm* &aalgorithm, *memory*::*desc* &src1_desc) const

Returns the parameters of a binary post-op.

Parameters

- **index** – Index of the binary post-op.

- **aalgorithm** – Output binary algorithm kind.
- **src1_desc** – Output memory descriptor of a second operand.

Scratchpad Mode

Some primitives might require a temporary buffer while performing their computations. For instance, the operations that do not have enough independent work to utilize all cores on a system might use parallelization over the reduction dimension (the K dimension in the GEMM notation). In this case different threads compute partial results in private temporary buffers, and then the private results are added to produce the final result. Another example is using matrix multiplication (GEMM) to implement convolution. Before calling GEMM, the source activations need to be transformed using the `im2col` operation. The transformation result is written to a temporary buffer that is then used as an input for the GEMM.

In both of these examples, the temporary buffer is no longer required once the primitive computation is completed. oneDNN refers to such kind of a memory buffer as a *scratchpad*.

Both types of implementation might need extra space for the reduction in case there are too few independent tasks. The amount of memory required by the `im2col` transformation is proportional to the size of the source image multiplied by the weights spatial size. The size of a buffer for reduction is proportional to the tensor size to be reduced (e.g., `diff_weights` in the case of backward by weights) multiplied by the number of threads in the reduction groups (the upper bound is the total number of threads).

By contrast, some other primitives might require very little extra space. For instance, one of the implementation of the *dnnl::sum* primitive requires temporary space only to store the pointers to data for each and every input array (that is, the size of the scratchpad is `n * sizeof(void *)`, where `n` is the number of summands).

oneDNN supports two modes for handling scratchpads:

enum class `dnnl::scratchpad_mode`

Scratchpad mode.

Values:

enumerator **library**

The library manages the scratchpad allocation. There may be multiple implementation-specific policies that can be configured via mechanisms that fall outside of the scope of this specification.

enumerator **user**

The user manages the scratchpad allocation by querying and providing the scratchpad memory to primitives. This mode is thread-safe as long as the scratchpad buffers are not used concurrently by two primitive executions.

The scratchpad mode is controlled though the *dnnl::primitive_attr::set_scratchpad_mode()* primitive attributes.

If the user provides scratchpad memory to a primitive, this memory must be created using the same engine that the primitive uses.

All primitives support both scratchpad modes.

Note: Primitives are not thread-safe by default. The only way to make the primitive execution fully thread-safe is to use the *dnnl::scratchpad_mode::user* mode and not pass the same scratchpad memory to two primitives that are executed concurrently.

Examples

Library Manages Scratchpad

As mentioned above, this is a default behavior. We only want to highlight how a user can query the amount of memory consumed by a primitive due to a scratchpad.

```
// Use default attr, hence the library allocates scratchpad
dnnl::primitive::primitive_desc op_pd(params, /* other arguments */);

// Print how much memory would be hold by a primitive due to scratchpad
std::cout << "primitive will use "
          << op_pd.query_s64(dnnl::query::memory_consumption_s64)
          << " bytes" << std::endl;

// In this case scratchpad is internal, hence user visible scratchpad memory
// descriptor should be empty:
auto zero_md = dnnl::memory::desc();
```

User Manages Scratchpad

```
// Create an empty (default) attributes
dnnl::primitive_attr attr;

// Default scratchpad mode is `library`:
assert(attr.get_scratchpad_mode() == dnnl::scratchpad_mode::library);

// Set scratchpad mode to `user`
attr.set_scratchpad_mode(dnnl::scratchpad_mode::user);

// Create a primitive descriptor with custom attributes
dnnl::primitive::primitive_desc op_pd(op_d, attr, engine);

// Query the scratchpad memory descriptor
dnnl::memory::desc scratchpad_md = op_pd.scratchpad_desc();

// Note, that a primitive doesn't consume memory in this configuration:
assert(op_pd.query_s64(dnnl::query::memory_consumption_s64) == 0);

// Create a primitive
dnnl::primitive prim(op_pd);

// ... more code ..

// Create a scratchpad memory
// NOTE: if scratchpad is not required for a particular primitive the
//       scratchpad_md.get_size() will return 0. It is fine to have
//       scratchpad_ptr == nullptr in this case.
void *scratchpad_ptr = user_memory_manager::allocate(scratchpad_md.get_size());
// NOTE: engine here must much the engine of the primitive
dnnl::memory scratchpad(scratchpad_md, engine, scratchpad_ptr);
```

(continues on next page)

(continued from previous page)

```
// Pass a scratchpad memory to a primitive
prim.execute(stream, { /* other arguments */,
        {DNNL_ARG_SCRATCHPAD, scratchpad}});
```

Quantization

Primitives may support reduced precision computations which require quantization. This process is explained in more details in the *Quantization Model* section.

Quantization Attributes (scales and zero-points)

oneDNN provides *dnnl::primitive_attr::set_scales_mask()* and *dnnl::primitive_attr::set_zero_points_mask()* for setting the quantization parameter for a given argument of a primitive.

The primitives may not support passing quantization parameters if source (and weights) tensors are not of the int8 data type. In other words, convolution operating on the single precision floating point data type may not scale and/or shift its inputs and outputs.

Broadcast semantic for quantization parameters is handled through masks that are explicitly passed to the *dnnl::primitive_attr::set_scales_mask()* and *dnnl::primitive_attr::set_zero_points_mask()* methods. For example, if the primitive destination is a $D_0 \times ... \times D_{n-1}$ tensor and we want to have a scale per d_i dimension (where $0 \leq d_i < n$), then $mask = \sum_{d_i} 2^{d_i}$ and the number of scales should be `scales.size()` $= \prod_{d_i} D_{d_i}$. The mask should be set in attributes during primitive creation, and the actual scales and zero-points are passed as argument to the primitive execution function.

The quantization parameters are applied in the single precision floating point data type (*dnnl::memory::data_type::f32*). Before it is stored, the result is converted to the destination data type with saturation if required. The rounding happens according to the current hardware setting.

When using *Post-ops*, the same *dnnl::primitive_attr::set_scales_mask()* and *dnnl::primitive_attr::set_zero_points_mask()* are used to pass quantization parameters to a given post-ops arguments.

Example 1: weights quantization with per-output-channel scaling

```
// weights dimensions
const int OC, IC, KH, KW;

// original f32 weights in plain format
dnnl::memory::desc wei_plain_f32_md(
        {OC, IC, KH, KW},                 // dims
        dnnl::memory::data_type::f32,     // the data originally in f32
        dnnl::memory::format_tag::hwigo   // the plain memory format
        );

// the scaling factors for quantized weights
// An unique scale for each output-channel.
std::vector<float> wei_scales(OC) = { /* values */ };
dnnl::memory();
```

(continues on next page)

(continued from previous page)

```
// int8 convolution primitive descriptor
dnnl::convolution_forward::primitive_desc conv_pd(/* see the next example */);

// query the convolution weights memory descriptor
dnnl::memory::desc wei_conv_s8_md = conv_pd.weights_desc();

// prepare the attributes for the reorder
dnnl::primitive_attr attr;
const int quantization_mask = 0
    | (1 << 0);  // scale per  OC dimension, which is the dim #0
attr.set_scales_mask(DNNL_ARG_DST, quantization_mask);

// create reorder that would perform:
//   wei_s8(oc, ic, kh, kw) <- wei_f32(oc, ic, kh, kw) / scale(oc)
// including the data format conversion.
auto wei_reorder_pd = dnnl::reorder::primitive_desc(
        wei_plain_f32_md, engine, // source
        wei_conv_s8_md, engine, // destination,
        attr);
auto wei_reorder = dnnl::reorder(wei_reorder_pd);
```

Example 2: convolution with groups, with per-output-channel quantization

This example is complementary to the previous example (which should ideally be the first one). Let's say we want to create an int8 convolution with per-output channel scaling.

```
const float src_scale; // src_f32[:] = src_scale * src_s8[:]
const float dst_scale; // dst_f32[:] = dst_scale * dst_s8[:]

// the scaling factors for quantized weights (as declared above)
// An unique scale for each group and output-channel.
std::vector<float> wei_scales(OC) = {...};


// Src, weights, and dst memory descriptors for convolution,
// with memory format tag == any to allow a convolution implementation
// to chose the appropriate memory format

dnnl::memory::desc src_conv_s8_any_md(
        {BATCH, IC, IH, IW},          // dims
        dnnl::memory::data_type::s8,  // the data originally in s8
        dnnl::memory::format_tag::any // let convolution to choose
        );

dnnl::memory::desc wei_conv_s8_any_md(
        {OC, IC, KH, KW},             // dims
        dnnl::memory::data_type::s8,  // the data originally in s8
        dnnl::memory::format_tag::any // let convolution to choose
        );
```

(continues on next page)

(continued from previous page)

```
dnnl::memory::desc dst_conv_s8_any_md(...);  // ditto

// prepare the attributes for the convolution
dnnl::primitive_attr attr;
const int data_mask = 0; // scale and zero-point per tensor for source and destination
const int wei_mask = 0
    | (1 << 1); // scale per OC dimension, which is the dim #0 on weights tensor:
                // (   OC, IC, KH, KW)
                //      0   1   2   3

attr.set_scales_mask(DNNL_ARG_SRC, data_mask);
attr.set_zero_points_mask(DNNL_ARG_SRC, data_mask);

attr.set_scales_mask(DNNL_ARG_WEIGHTS, wei_mask);

attr.set_scales_mask(DNNL_ARG_DST, data_mask);
attr.set_zero_points_mask(DNNL_ARG_DST, data_mask);

// create a convolution primitive descriptor
auto conv_pd = dnnl::convolution_forward::primitive_desc(
        dnnl::prop_kind::forward_inference,
        dnnl::algorithm::convolution_direct,
        src_conv_s8_any_md,                     // what's important is that
        wei_conv_s8_any_md,                     // we specified that we want
        dst_conv_s8_any_md,                     // computations in s8
        strides, padding_l, padding_r,
        dnnl::padding_kind::zero
        attr);   // the attributes describe the quantization flow
```

Implicit downconversions and floating-point math mode

oneDNN provides *dnnl::primitive_attr::set_fpmath_mode()* to allow implicit downconversions from fp32 to lower accuracy datatypes during primitive execution. For some applications, it allows to speedup computations without noticeable impact on accuracy.

The *dnnl::primitive_attr::set_fpmath_mode()* primitive attribute specifies which implicit down-conversions are allowed for that given primitive. Only down-conversions from f32 to narrower data-types (f16, bf16, or tf32) are currently allowed. Furthermore these down-conversions are allowed only during computation, and do not affect the storage datatype (which must remain f32).

The *dnnl::primitive_attr::set_fpmath_mode()* primitive attribute can take 3 types of values:

- the *strict* mode disables any down-conversion (default).
- the *any* mode allows all conversions from f32 to a smaller floating-point datatype (f16, bf16, or tf32).
- a specific datatype (f16, bf16, or tf32) which specifically allows down-conversion only from f32 to a datatype at least as accurate as the specified data-type (at least same number of exponent and mantissa bits).

The default value for this attribute shall be *strict*. However, it is allowed to expose global functions or environment variables to change this default value.

This attribute is ignored if a primitive computation data-type is integral.

Attribute Related Error Handling

Since the attributes are created separately from the corresponding primitive descriptor, consistency checks are delayed. Users can successfully set attributes in whatever configuration they want. However, when they try to create a primitive descriptor with the attributes they set, it might happen that there is no primitive implementation that supports such a configuration. In this case the library will throw the *dnnl::error* exception.

API

struct **primitive_attr**

Primitive attributes.

Public Functions

primitive_attr()

Constructs default (empty) primitive attributes.

scratchpad_mode **get_scratchpad_mode**() const

Returns the scratchpad mode.

void **set_scratchpad_mode**(*scratchpad_mode* mode)

Sets scratchpad mode.

Parameters
mode – Specified scratchpad mode.

fpmath_mode **get_fpmath_mode**() const

Returns the fpmath mode.

void **set_fpmath_mode**(fpmath_mode mode)

Sets fpmath mode.

Parameters
mode – Specified fpmath mode.

int **get_scales_mask**(int arg) const

Returns scaling factors correspondence mask for a given memory argument.

Parameters
arg – Parameter argument index as passed to the *primitive::execute()* call.

void **set_scales_mask**(int arg, int mask)

Sets scaling factors correspondance mask for a given memory argument.

See also:

dnnl::primitive_attr::set_scales_mask

Note: The order of dimensions does not depend on how elements are laid out in memory. For example:

- for a 2D CNN activations tensor the order is always (n, c)
- for a 4D CNN activations tensor the order is always (n, c, h, w)
- for a 5D CNN weights tensor the order is always

Parameters

- **arg** – Parameter argument index as passed to the *primitive::execute()* call.
- **mask** – Scaling factors correspondence mask that defines the correspondence between the `arg` tensor dimensions and the scales vector. Setting the i-th bit indicates that a dedicated scaling factor is used for each index along that dimension. Set the mask to 0 to use a common scaling factor for the whole tensor. The scales must be passed at execution time as an argument with index *DNNL_ARG_ATTR_SCALES*.

void **set_zero_points_mask**(int arg, int mask)

Sets zero points for primitive operations for a given memory argument.

See also:

dnnl::primitive_attr::set_output_scales

Parameters

- **arg** – Parameter argument index as passed to the *primitive::execute()* call.
- **mask** – Zero point correspondence mask that defines the correspondence between the tensor dimensions and the `zero_points` vector. The set i-th bit indicates that a dedicated zero point is used for each index along that dimension. Set the mask to 0 to use a common zero point for the whole output tensor. The zero points must be passed at execution time as an argument with index *DNNL_ARG_ATTR_ZERO_POINTS*.

const *post_ops* **get_post_ops**() const

Returns post-ops previously set via *set_post_ops()*.

Returns
Post-ops.

void **set_post_ops**(const *post_ops* ops)

Sets post-ops.

Note: There is no way to check whether the post-ops would be supported by the target primitive. Any error will be reported by the respective primitive descriptor constructor.

Parameters
ops – Post-ops object to copy post-ops from.

void **set_rnn_data_qparams**(float scale, float shift)

Sets quantization scale and shift parameters for RNN data tensors.

For performance reasons, the low-precision configuration of the RNN primitives expect input activations to have the unsigned 8-bit integer data type. The scale and shift parameters are used to quantize floating-point data to unsigned integer and must be passed to the RNN primitive using attributes.

The quantization formula is `scale * (data + shift)`.

Example usage:

```
// RNN parameters
int l = 2, t = 2, mb = 32, sic = 32, slc = 32, dic = 32, dlc = 32;
// Activations quantization parameters
float scale = 2.0f, shift = 0.5f;

primitive_attr attr;

// Set scale and shift for int8 quantization of activation
attr.set_rnn_data_qparams(scale, shift);

// Create and configure rnn op_desc
vanilla_rnn_forward::desc rnn_d(/* arguments */);
vanilla_rnn_forward::primitive_desc rnn_d(rnn_d, attr, engine);
```

Note: Quantization scale and shift are common for src_layer, src_iter, dst_iter, and dst_layer.

Parameters

- **scale** – The value to scale the data by.
- **shift** – The value to shift the data by.

void **set_rnn_weights_qparams**(int mask, const std::vector<float> &scales)

Sets quantization scaling factors for RNN weights tensors. The low-precision configuration of the RNN primitives expect input weights to use the signed 8-bit integer data type. The scaling factors are used to quantize floating-point data to signed integer and must be passed to RNN primitives using attributes.

Note: The dimension order is always native and does not depend on the actual layout used. For example, five-dimensional weights always have (l, d, i, g, o) logical dimension ordering.

Note: Quantization scales are common for weights_layer and weights_iteration

Parameters

- **mask** – Scaling factors correspondence mask that defines the correspondence between the output tensor dimensions and the `scales` vector. The set i-th bit indicates that a dedicated scaling factor should be used each index along that dimension. Set the mask to 0 to use a common scaling factor for the whole output tensor.
- **scales** – Constant vector of output scaling factors. The following equality must hold: $scales.size() = \prod_{d \in mask} weights.dims[d]$. Violations can only be detected when the attributes are used to create a primitive descriptor.

4.5.3 Batch Normalization

The batch normalization primitive performs a forward or backward batch normalization operation on tensors with number of dimensions equal to 2 or more. Variable names follow the standard *Conventions*.

The batch normalization operation is defined by the following formulas. We show formulas only for 2D spatial data which are straightforward to generalize to cases of higher and lower dimensions.

The different flavors of the primitive are controlled by the `flags` parameter that is passed to the primitive descriptor initialization function like *dnnl::batch_normalization_forward::primitive_desc*. Multiple flags can be combined using the bitwise OR operator (`|`).

Forward

$$\text{dst}(n,c,h,w) = \gamma(c) \cdot \frac{\text{src}(n,c,h,w) - \mu(c)}{\sqrt{\sigma^2(c) + \varepsilon}} + \beta(c),$$

where

- $\gamma(c)$ and $\beta(c)$ are optional scale and shift for a channel (controlled using the *use_scale* and *use_shift* flags),
- $\mu(c)$ and $\sigma^2(c)$ are mean and variance for a channel (controlled using the *use_global_stats* flag), and
- ε is a constant to improve numerical stability.

Mean and variance are computed at runtime or provided by a user. When mean and variance are computed at runtime, the following formulas are used:

- $\mu(c) = \frac{1}{NHW} \sum_{nhw} \text{src}(n,c,h,w)$,
- $\sigma^2(c) = \frac{1}{NHW} \sum_{nhw} (\text{src}(n,c,h,w) - \mu(c))^2$.

The $\gamma(c)$ and $\beta(c)$ tensors are considered learnable.

In the training mode, the primitive also optionally supports fusion with ReLU activation with zero negative slope applied to the result (see *fuse_norm_relu* flag).

Note: The batch normalization primitive computes population mean and variance and not the sample or unbiased versions that are typically used to compute running mean and variance. * Using the mean and variance computed by the batch normalization primitive, running mean and variance $\hat{\mu}_i$ and $\hat{\sigma}_i^2$ where i is iteration number, can be computed as:

$$\hat{\mu}_{i+1} = \alpha \cdot \hat{\mu}_i + (1-\alpha) \cdot \mu,$$
$$\hat{\sigma}_{i+1}^2 = \alpha \cdot \hat{\sigma}_i^2 + (1-\alpha) \cdot \sigma^2.$$

Difference Between Forward Training and Forward Inference

- If mean and variance are computed at runtime (i.e., *use_global_stats* is not set), they become outputs for the propagation kind *forward_training* (because they would be required during the backward propagation) and are not exposed for the propagation kind *forward_inference*.
- If batch normalization is created with ReLU fusion (i.e., *fuse_norm_relu* is set), for the propagation kind *forward_training* the primitive would produce a `workspace` memory as one extra output. This memory is required to compute the backward propagation. When the primitive is executed with propagation kind *forward_inference*, the workspace is not produced. Behavior would be the same as creating a batch normalization primitive with ReLU as a post-op (see section below).

Backward

The backward propagation computes $\text{diff\_src}(n, c, h, w)$, $\text{diff\_}\gamma(c)^*$, and $\text{diff\_}\beta(c)^*$ based on $\text{diff\_dst}(n, c, h, w)$, $\text{src}(n, c, h, w)$, $\mu(c)$, $\sigma^2(c)$, $\gamma(c)^*$, and $\beta(c)^*$.

The tensors marked with an asterisk are used only when the primitive is configured to use $\gamma(c)$ and $\beta(c)$ (i.e., *use_scale* and *use_shift* are set).

Execution Arguments

Depending on the flags and propagation kind, the batch normalization primitive requires different inputs and outputs. For clarity, a summary is shown below.

	forward_infer	*forward_training*	*backward*	*backward_data*
none	*In*: src; *Out*: dst	*In*: src; *Out*: dst, μ, σ^2	*In*: diff_dst, src, μ, σ^2; *Out*: diff_src	Same as for *backward*
use_global_st	*In*: src, μ, σ^2; *Out*: dst	*In*: src, μ, σ^2; *Out*: dst	*In*: diff_dst, src, μ, σ^2; *Out*: diff_src	Same as for *backward*
use_scale	*In*: src, γ; *Out*: dst	*In*: src, γ; *Out*: dst, μ, σ^2	*In*: diff_dst, src, μ, σ^2, γ; *Out*: diff_src, diff_γ	Not supported
use_shift	*In*: src, β; *Out*: dst	*In*: src, β; *Out*: dst, μ, σ^2	*In*: diff_dst, src, μ, σ^2, β; *Out*: diff_src, diff_β	Not supported
use_scale | *use_shift*	*In*: src, γ, β; *Out*: dst	*In*: src, γ, β; *Out*: dst, μ, σ^2	*In*: diff_dst, src, μ, σ^2, γ, β; *Out*: diff_src, diff_γ, diff_β	Not supported
use_global_st | *use_scale* | *use_shift*	*In*: src, μ, σ^2, γ, β; *Out*: dst	*In*: src, μ, σ^2, γ, β; *Out*: dst	*In*: diff_dst, src, μ, σ^2, γ, β; *Out*: diff_src, diff_γ, diff_β	Not supported
`flags` | *fuse_norm_rel*	*In*: same as with `flags`; *Out*: same as with `flags`	*In*: same as with `flags`; *Out*: same as with `flags`, workspace	*In*: same as with `flags`, workspace; *Out*: same as with `flags`	Same as for *backward* if `flags` do not contain *use_scale* or *use_shift*; not supported otherwise

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	*DNNL_ARG_SRC*
γ	*DNNL_ARG_SCALE*
β	*DNNL_ARG_SHIFT*
mean (μ)	*DNNL_ARG_MEAN*
variance (σ)	*DNNL_ARG_VARIANCE*
dst	*DNNL_ARG_DST*
workspace	*DNNL_ARG_WORKSPACE*
diff_dst	*DNNL_ARG_DIFF_DST*
diff_src	*DNNL_ARG_DIFF_SRC*
diff_γ	*DNNL_ARG_DIFF_SCALE*
diff_β	*DNNL_ARG_DIFF_SHIFT*

Operation Details

1. For forward propagation, the mean and variance might be either computed at runtime (in which case they are outputs of the primitive) or provided by a user (in which case they are inputs). In the latter case, a user must set the *use_global_stats* flag. For the backward propagation, the mean and variance are always input parameters.
2. The memory format and data type for `src` and `dst` are assumed to be the same, and in the API they are typically referred to as `data` (e.g., see `data_desc` in *dnnl::batch_normalization_forward::primitive_desc*). The same is true for `diff_src` and `diff_dst`. The corresponding memory descriptors are referred to as `diff_data_desc`.
3. Both forward and backward propagation support in-place operations, meaning that src can be used as input and output for forward propagation, and diff_dst can be used as input and output for backward propagation. In case of an in-place operation, the original data will be overwritten. Note, however, that backward propagation requires original src, hence the corresponding forward propagation should not be performed in-place.
4. As mentioned above, the batch normalization primitive can be fused with ReLU activation even in the training mode. In this case, on the forward propagation the primitive has one additional output, `workspace`, that should be passed during the backward propagation.

Data Types Support

The operation supports the following combinations of data types.

Note: Here we abbreviate data types names for readability. For example, *dnnl::memory::data_type::f32* is abbreviated to *f32*.

Propagation	Source / Destination	Mean / Variance / Scale / Shift
forward / backward	*f32*, *bf16*	*f32*
forward	*f16*	*f32*
forward	*s8*	*f32*

Data Representation

Source, Destination, and Their Gradients

Like other CNN primitives, the batch normalization primitive expects data to be $N \times C \times SP_n \times \cdots \times SP_0$ tensor.

The batch normalization primitive is optimized for the following memory formats:

Spatial	Logical tensor	Implementations optimized for memory formats
0D	NC	`nc` (`ab`)
1D	NCW	`ncw` (`abc`), `nwc` (`acb`), *optimized*
2D	NCHW	`nchw` (`abcd`), `nhwc` (`acdb`), *optimized*
3D	NCDHW	`ncdhw` (`abcde`), `ndhwc` (`acdeb`), *optimized*

Here *optimized* means the format chosen by the preceding compute-intensive primitive.

Statistics Tensors

The mean (μ) and variance (σ^2) are separate 1D tensors of size C.

The format of the corresponding memory object must be `x` (`a`).

If used, the scale (γ) and shift (β) are combined in a single 2D tensor of shape $2 \times C$.

The format of the corresponding memory object must be `nc` (`ab`).

Post-ops and Attributes

Propagation	Type	Operation	Description
forward	post-op	eltwise	Applies an eltwise operation to the output.

Note: Using ReLU as a post-op does not produce additional output in the `workspace` that is required to compute backward propagation correctly. Hence, one should use the *`fuse_norm_relu`* flag for training.

API

struct **`batch_normalization_forward`** : public dnnl::*primitive*

Batch normalization forward propagation primitive.

Public Functions

batch_normalization_forward()

Default constructor. Produces an empty object.

batch_normalization_forward(const *primitive_desc* &pd)

Constructs a batch normalization forward propagation primitive.

Parameters
pd – Primitive descriptor for a batch normalization forward propagation primitive.

struct **primitive_desc** : public dnnl::*primitive_desc*

Primitive descriptor for a batch normalization forward propagation primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, const *memory*::*desc* &src_desc, const *memory*::*desc* &dst_desc, float epsilon, *normalization_flags* flags, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a batch normalization forward propagation primitive.

Note: In-place operation is supported: the dst can refer to the same memory as the src.

Parameters
- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training* and *dnnl::prop_kind::forward_inference*.
- **src_desc** – Source memory descriptor.
- **dst_desc** – Destination memory descriptor.
- **epsilon** – Batch normalization epsilon parameter.
- **flags** – Batch normalization flags (*dnnl::normalization_flags*).
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() const

Returns a source memory descriptor.
Returns
Source memory descriptor.
Returns
A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **dst_desc**() const

Returns a destination memory descriptor.
Returns
Destination memory descriptor.
Returns
A zero memory descriptor if the primitive does not have a destination parameter.

memory::*desc* **weights_desc**() const

Returns a weights memory descriptor.

Returns

Weights memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a weights parameter.

memory::*desc* **workspace_desc**() const

Returns the workspace memory descriptor.

Returns

Workspace memory descriptor.

Returns

A zero memory descriptor if the primitive does not require workspace parameter.

memory::*desc* **mean_desc**() const

Returns memory descriptor for mean.

Returns

Memory descriptor for mean.

memory::*desc* **variance_desc**() const

Returns memory descriptor for variance.

Returns

Memory descriptor for variance.

dnnl::*prop_kind* **get_prop_kind**() const

Returns a propagation kind.

Returns

A propagation kind.

Returns

dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

float **get_epsilon**() const

Returns an epsilon.

Returns

An epsilon.

Returns

Zero if the primitive does not have an epsilon parameter.

normalization_flags **get_flags**() const

Returns normalization flags.

Returns

Normalization flags.

struct **batch_normalization_backward** : public dnnl::*primitive*

Batch normalization backward propagation primitive.

Public Functions

batch_normalization_backward()

Default constructor. Produces an empty object.

batch_normalization_backward(const *primitive_desc* &pd)

Constructs a batch normalization backward propagation primitive.

Parameters
pd – Primitive descriptor for a batch normalization backward propagation primitive.

struct **primitive_desc** : public dnnl::*primitive_desc*

Primitive descriptor for a batch normalization backward propagation primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, const *memory*::*desc* &diff_src_desc, const *memory*::*desc* &diff_dst_desc, const *memory*::*desc* &src_desc, float epsilon, *normalization_flags* flags, const *batch_normalization_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a batch normalization backward propagation primitive.

Parameters
- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::backward_data* and *dnnl::prop_kind::backward* (diffs for all parameters are computed in this case).
- **diff_src_desc** – Diff source memory descriptor.
- **diff_dst_desc** – Diff destination memory descriptor.
- **src_desc** – Source memory descriptor.
- **epsilon** – Batch normalization epsilon parameter.
- **flags** – Batch normalization flags (*dnnl::normalization_flags*).
- **hint_fwd_pd** – Primitive descriptor for a batch normalization forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() const

Returns a source memory descriptor.

Returns
Source memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **weights_desc**() const

Returns a weights memory descriptor.

Returns
Weights memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a weights parameter.

memory::*desc* **dst_desc**() const

Returns a destination memory descriptor.

Returns
Destination memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a destination parameter.

memory::*desc* **diff_src_desc**() const

Returns a diff source memory descriptor.

Returns
Diff source memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a diff source memory with.

memory::*desc* **diff_dst_desc**() const

Returns a diff destination memory descriptor.

Returns
Diff destination memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a diff destination parameter.

memory::*desc* **diff_weights_desc**()

Returns a diff weights memory descriptor.

Returns
Diff weights memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a diff weights parameter.

memory::*desc* **mean_desc**() const

Returns memory descriptor for mean.

Returns
Memory descriptor for mean.

memory::*desc* **variance_desc**() const

Returns memory descriptor for variance.

Returns
Memory descriptor for variance.

memory::*desc* **workspace_desc**() const

Returns the workspace memory descriptor.

Returns
Workspace memory descriptor.

Returns
A zero memory descriptor if the primitive does not require workspace parameter.

dnnl::*prop_kind* **get_prop_kind**() const

Returns a propagation kind.

Returns
A propagation kind.

Returns
dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

float **get_epsilon**() const

Returns an epsilon.

Returns
An epsilon.

Returns
Zero if the primitive does not have an epsilon parameter.

normalization_flags **get_flags**() const

Returns normalization flags.

Returns
Normalization flags.

4.5.4 Binary

The binary primitive computes a result of a binary elementwise operation between tensors source 0 and source 1.

$$\mathrm{dst}(\overline{x}) = \mathrm{src}_0(\overline{x}) \; op \; \mathrm{src}_1(\overline{x}),$$

where $\overline{x} = (x_0, \ldots, x_n)$ and op is an operator like addition, multiplication, maximum or minimum. Variable names follow the standard *Conventions*.

Forward and Backward

The binary primitive does not have a notion of forward or backward propagations.

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src_0	*DNNL_ARG_SRC_0*
src_1	*DNNL_ARG_SRC_1*
dst	*DNNL_ARG_DST*

Operation Details

- The binary primitive requires all source and destination tensors to have the same number of dimensions.
- The binary primitive supports implicit broadcast semantics for source 1. It means that if some dimension has value of one, this value will be used to compute an operation with each point of source 0 for this dimension.
- The dst memory format can be either specified explicitly or by *dnnl::memory::format_tag::any* (recommended), in which case the primitive will derive the most appropriate memory format based on the format of the source 0 tensor.
- Destination memory descriptor should completely match source 0 memory descriptor.
- The binary primitive supports in-place operations, meaning that source 0 tensor may be used as the destination, in which case its data will be overwritten.

Post-ops and Attributes

The following attributes should be supported:

Type	Operation	Description	Restrictions
Attribute	*Scales*	Sets scale(s) for the corresponding tensor(s)	
post-op	*Eltwise*	Applies an elementwise operation to the result	
post-op	*Binary*	Applies a binary operation to the result	
post-op	*Sum*	Adds the operation result to the destination tensor instead of overwriting it	

Data Types Support

The source and destination tensors may have *dnnl::memory::data_type::f32*, *dnnl::memory::data_type::bf16*, *dnnl::memory::data_type::s8* or *dnnl::memory::data_type::u8* data types.

Data Representation

The binary primitive works with arbitrary data tensors. There is no special meaning associated with any of tensors dimensions.

API

struct **binary** : public dnnl::*primitive*

Elementwise binary operator primitive.

Public Functions

binary()

Default constructor. Produces an empty object.

binary(const *primitive_desc* &pd)

Constructs an elementwise binary operation primitive.

Parameters

pd – Primitive descriptor for an elementwise binary operation primitive.

struct **primitive_desc** : public dnnl::*primitive_desc*

Primitive descriptor for an elementwise binary operator primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *algorithm* aalgorithm, const *memory*::*desc* &src0, const *memory*::*desc* &src1, const *memory*::*desc* &dst, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for an elementwise binary operator primitive.

Parameters

- **aengine** – Engine to use.
- **aalgorithm** – Elementwise binary algorithm.
- **src0** – Memory descriptor for source tensor #0.
- **src1** – Memory descriptor for source tensor #1.
- **dst** – Memory descriptor for destination tensor.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**(int idx = 0) const

Returns a source memory descriptor.

Parameters

idx – Source index.

Returns

Source memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a source parameter with index `pdx`.

memory::*desc* **src0_desc**() const

Returns the memory descriptor for source #0.

memory::*desc* **src1_desc**() const

Returns the memory descriptor for source #1.

memory::*desc* **dst_desc**() const

Returns a destination memory descriptor.

Returns

Destination memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a destination parameter.

algorithm **get_algorithm**() const

Returns an algorithm kind.

Returns

An algorithm kind.

Returns

dnnl::algorithm::undef if the primitive does not have an algorithm parameter.

4.5.5 Concat

A primitive to concatenate data by arbitrary dimension.

The concat primitive concatenates N tensors over `concat_dimension` (here denoted as C), and is defined as

$$\mathrm{dst}(\overline{ou}, c, \overline{in}) = \mathrm{src}_i(\overline{ou}, c', \overline{in}),$$

where

- $c = C_1 + \ldots + C_{i-1} + c'$,
- $\overline{ou}$ is the outermost indices (to the left from concat axis),
- $\overline{in}$ is the innermost indices (to the right from concat axis), and

Variable names follow the standard *Conventions*.

Forward and Backward

The concat primitive does not have a notion of forward or backward propagations. The backward propagation for the concatenation operation is simply an identity operation.

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	*DNNL_ARG_MULTIPLE_SRC*
dst	*DNNL_ARG_DST*

Operation Details

1. The dst memory format can be either specified by a user or derived by the primitive. The recommended way is to allow the primitive to choose the most appropriate format.
2. The concat primitive requires all source and destination tensors to have the same shape except for the `concat_dimension`. The destination dimension for the `concat_dimension` must be equal to the sum of the `concat_dimension` dimensions of the sources (i.e. $C = \sum_i C_i$). Implicit broadcasting is not supported.

Data Types Support

The concat primitive supports arbitrary data types for source and destination tensors. However, it is required that all source tensors are of the same data type (but not necessarily matching the data type of the destination tensor).

Data Representation

The concat primitive does not assign any special meaning associated with any logical dimensions.

Post-ops and Attributes

The concat primitive does not support any post-ops or attributes.

API

struct **concat** : public dnnl::*primitive*

Tensor concatenation (concat) primitive.

Public Functions

concat()

Default constructor. Produces an empty object.

concat(const *primitive_desc* &pd)

Constructs a concatenation primitive.

Parameters
pd – Primitive descriptor for concatenation primitive.

struct **primitive_desc** : public dnnl::*primitive_desc_base*

Primitive descriptor for a concat primitive.

Public Functions

primitive_desc()

Default constructor. Produces an empty object.

primitive_desc(const *memory*::*desc* &dst, int concat_dimension, const std::vector<*memory*::*desc*> &srcs, const *engine* &aengine, const *primitive_attr* &attr = *primitive_attr*())

Constructs a primitive descriptor for an out-of-place concatenation primitive.

Parameters
- **dst** – Destination memory descriptor.
- **concat_dimension** – Source tensors will be concatenated over dimension with this index. Note that order of dimensions does not depend on memory format.
- **srcs** – Vector of source memory descriptors.
- **aengine** – Engine to perform the operation on.
- **attr** – Primitive attributes to use (optional).

primitive_desc(int concat_dimension, const std::vector<*memory*::*desc*> &srcs, const *engine* &aengine, const *primitive_attr* &attr = *primitive_attr*())

Constructs a primitive descriptor for an out-of-place concatenation primitive.

This version derives the destination memory descriptor automatically.

Parameters
- **concat_dimension** – Source tensors will be concatenated over dimension with this index. Note that order of dimensions does not depend on memory format.

- **srcs** – Vector of source memory descriptors.
- **aengine** – Engine to perform the operation on.
- **attr** – Primitive attributes to use (optional).

memory::*desc* **src_desc**(int idx = 0) const

Returns a source memory descriptor.

Parameters

idx – Source index.

Returns

Source memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a source parameter with index pdx.

memory::*desc* **dst_desc**() const

Returns a destination memory descriptor.

Returns

Destination memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a destination parameter.

4.5.6 Convolution and Deconvolution

The convolution and deconvolution primitives compute forward, backward, or weight update for a batched convolution or deconvolution operations on 1D, 2D, or 3D spatial data with bias.

The operations are defined by the following formulas. We show formulas only for 2D spatial data which are straightforward to generalize to cases of higher and lower dimensions. Variable names follow the standard *Conventions*.

Forward

Let src, $\mathrm{weights}$ and dst be $N \times IC \times IH \times IW$, $OC \times IC \times KH \times KW$, and $N \times OC \times OH \times OW$ tensors respectively. Let bias be a 1D tensor with OC elements.

Furthermore, let the remaining convolution parameters be:

Parameter	Deptl	Heigh	Widtl	Comment
Padding: Front, top, and left	PD_L	PH_L	PW_L	In the API `padding_l` indicates the corresponding vector of paddings (`_l` in the name stands for **left**)
Padding: Back, bottom, and right	PD_R	PH_R	PW_R	In the API `padding_r` indicates the corresponding vector of paddings (`_r` in the name stands for **right**)
Stride	SD	SH	SW	Convolution without strides is defined by setting the stride parameters to 1
Dilation	DD	DH	DW	Non-dilated convolution is defined by setting the dilation parameters to 0

The following formulas show how oneDNN computes convolutions. They are broken down into several types to simplify the exposition, but in reality the convolution types can be combined.

To further simplify the formulas, we assume that $\mathrm{src}(n, ic, ih, iw) = 0$ if $ih < 0$, or $ih \geq IH$, or $iw < 0$, or $iw \geq IW$.

Regular Convolution

$$\begin{aligned}\mathrm{dst}(n, oc, oh, ow) &= \mathrm{bias}(oc) \\ &+ \sum_{ic=0}^{IC-1} \sum_{kh=0}^{KH-1} \sum_{kw=0}^{KW-1} \mathrm{src}(n, ic, oh', ow') \cdot \mathrm{weights}(oc, ic, kh, kw).\end{aligned}$$

Here:

- $oh' = oh \cdot SH + kh - PH_L$,
- $ow' = ow \cdot SW + kw - PW_L$,
- $OH = \left\lfloor \frac{IH - KH + PH_L + PH_R}{SH} \right\rfloor + 1$,
- $OW = \left\lfloor \frac{IW - KW + PW_L + PW_R}{SW} \right\rfloor + 1$.

Convolution with Groups

oneDNN adds a separate groups dimension to memory objects representing $\mathrm{weights}$ tensors and represents weights as $G \times OC_G \times IC_G \times KH \times KW$ 5D tensors for 2D convolutions with groups.

$$\begin{aligned}\mathrm{dst}(n, g \cdot OC_G + oc_g, oh, ow) &= \mathrm{bias}(g \cdot OC_G + oc_g) \\ &+ \sum_{ic_g=0}^{IC_G-1} \sum_{kh=0}^{KH-1} \sum_{kw=0}^{KW-1} \mathrm{src}(n, g \cdot IC_G + ic_g, oh', ow') \cdot \mathrm{weights}(g, oc_g, ic_g, kh, kw),\end{aligned}$$

where

- $IC_G = \frac{IC}{G}$,
- $OC_G = \frac{OC}{G}$, and
- $oc_g \in [0, OC_G)$.

The case when $OC_G = IC_G = 1$ is also known as *a depthwise convolution*.

Convolution with Dilation

$$\begin{aligned}\mathrm{dst}(n, oc, oh, ow) &= \mathrm{bias}(oc) + \\ &+ \sum_{ic=0}^{IC-1} \sum_{kh=0}^{KH-1} \sum_{kw=0}^{KW-1} \mathrm{src}(n, ic, oh'', ow'') \cdot \mathrm{weights}(oc, ic, kh, kw).\end{aligned}$$

Here:

- $oh'' = oh \cdot SH + kh \cdot (DH + 1) - PH_L$,
- $ow'' = ow \cdot SW + kw \cdot (DW + 1) - PW_L$,
- $OH = \left\lfloor \frac{IH - DKH + PH_L + PH_R}{SH} \right\rfloor + 1$, where $DKH = 1 + (KH - 1) \cdot (DH + 1)$, and
- $OW = \left\lfloor \frac{IW - DKW + PW_L + PW_R}{SW} \right\rfloor + 1$, where $DKW = 1 + (KW - 1) \cdot (DW + 1)$.

Deconvolution (Transposed Convolution)

Deconvolutions (also called fractionally-strided convolutions or transposed convolutions) can be defined by swapping the forward and backward passes of a convolution. One way to put it is to note that the weights define a convolution, but whether it is a direct convolution or a transposed convolution is determined by how the forward and backward passes are computed.

Difference Between Forward Training and Forward Inference

There is no difference between the *forward_training* and *forward_inference* propagation kinds.

Backward

The backward propagation computes diff_src based on diff_dst and weights.

The weights update computes diff_weights and diff_bias based on diff_dst and src.

Note: The *optimized* memory formats src and weights might be different on forward propagation, backward propagation, and weights update.

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	*DNNL_ARG_SRC*
weights	*DNNL_ARG_WEIGHTS*
bias	*DNNL_ARG_BIAS*
dst	*DNNL_ARG_DST*
diff_src	*DNNL_ARG_DIFF_SRC*
diff_weights	*DNNL_ARG_DIFF_WEIGHTS*
diff_bias	*DNNL_ARG_DIFF_BIAS*
diff_dst	*DNNL_ARG_DIFF_DST*

Operation Details

N/A

Data Types Support

Convolution primitive supports the following combination of data types for source, destination, and weights memory objects.

Note: Here we abbreviate data types names for readability. For example, *dnnl::memory::data_type::f32* is abbreviated to *f32*.

Propagation	Source	Weights	Destination	Bias
forward / backward	*f32*	*f32*	*f32*	*f32*
forward	*f16*	*f16*	*f16*	*f16*
forward	*u8*, *s8*	*s8*	*u8*, *s8*, *s32*, *f32*	*u8*, *s8*, *s32*, *f32*
forward	*bf16*	*bf16*	*f32*, *bf16*	*f32*, *bf16*
backward	*f32*, *bf16*	*bf16*	*bf16*	
weights update	*bf16*	*f32*, *bf16*	*bf16*	*f32*, *bf16*

Data Representation

Like other CNN primitives, the convolution primitive expects the following tensors:

Spatial	Source / Destination	Weights
1D	$N \times C \times W$	$[G\times]OC \times IC \times KW$
2D	$N \times C \times H \times W$	$[G\times]OC \times IC \times KH \times KW$
3D	$N \times C \times D \times H \times W$	$[G\times]OC \times IC \times KD \times KH \times KW$

Memory format of data and weights memory objects is critical for convolution primitive performance. In the oneDNN programming model, convolution is one of the few primitives that support the placeholder memory format tag *any* and can define data and weight memory objects format based on the primitive parameters. When using *any* it is necessary to first create a convolution primitive descriptor and then query it for the actual data and weight memory objects formats.

While convolution primitives can be created with memory formats specified explicitly, the performance is likely to be suboptimal.

The table below shows the combinations for which *plain* memory formats the convolution primitive is optimized for.

Spatial	Convolution Type	Data / Weights logical tensor	Implementation optimized for memory formats
1D, 2D, 3D		*any*	*optimized*
1D	f32, bf16	NCW / OIW, GOIW	*ncw* (*abc*) / *oiw* (*abc*), *goiw* (*abcd*)
1D	f32, bf16	NCW / OIW, GOIW	*nwc* (*acb*) / *wio* (*cba*), *wigo* (*dcab*)
1D	int8	NCW / OIW	*nwc* (*acb*) / *wio* (*cba*)
2D	f32, bf16	NCHW / OIHW, GOIHW	*nchw* (*abcd*) / *oihw* (*abcd*), *goihw* (*abcde*)
2D	f32, bf16	NCHW / OIHW, GOIHW	*nhwc* (*acdb*) / *hwio* (*cdba*), *hwigo* (*decab*)
2D	int8	NCHW / OIHW, GOIHW	*nhwc* (*acdb*) / *hwio* (*cdba*), *hwigo* (*decab*)
3D	f32, bf16	NCDHW / OIDHW, GOIDHW	*ncdhw* (*abcde*) / *oidhw* (*abcde*), *goidhw* (*abcdef*)
3D	f32, bf16	NCDHW / OIDHW, GOIDHW	*ndhwc* (*acdeb*) / *dhwio* (*cdeba*), *dhwigo* (*defcab*)
3D	int8	NCDHW / OIDHW	*ndhwc* (*acdeb*) / *dhwio* (*cdeba*)

Post-ops and Attributes

Post-ops and attributes enable you to modify the behavior of the convolution primitive by applying quantization parameters to the result of the primitive and by chaining certain operations after the primitive. The following attributes and post-ops are supported:

Type	Operation	Description	Restrictions
Attribute	*Scales*	Sets scale(s) for the corresponding tensor(s)	Int8 computations only
Attribute	*Zero points*	Sets zero point(s) for the corresponding tensors	Int8 computations only
post-op	*Eltwise*	Applies an elementwise operation to the result	
post-op	*Binary*	Applies a binary operation to the result	
post-op	*Sum*	Adds the operation result to the destination tensor instead of overwriting it	

The primitive supports dynamic quantization via run-time scales. That means a user could configure the scales and zero-point attributes at the primitive descriptor creation stage. The user must then provide the scales and zero-points as an additional input memory objects with argument *DNNL_ARG_ATTR_SCALES* and *DNNL_ARG_ATTR_ZERO_POINTS* during the execution stage (more details are provided in the *Quantization* section).

Note: The library does not prevent using post-ops in training, but note that not all post-ops are feasible for training usage. For instance, using ReLU with non-zero negative slope parameter as a post-op would not produce an additional

output `workspace` that is required to compute backward propagation correctly. Hence, in this particular case one should use separate convolution and eltwise primitives for training.

The following post-ops chaining should be supported by the library:

Type of convolutions	Post-ops sequence supported
f32 and bf16 convolution	eltwise, sum, sum -> eltwise
int8 convolution	eltwise, sum, sum -> eltwise, eltwise -> sum

The operations during attributes and post-ops applying are done in single precision floating point data type. The conversion to the actual destination data type happens just before the actual storing.

Example 1

Consider the following pseudo code:

```
attribute attr;
attr.set_post_ops({
        { sum={scale=beta} },
        { eltwise={scale=gamma, type=tanh, alpha=ignore, beta=ignored }
    });

convolution_forward(src, weights, dst, attr)
```

The would lead to the following:

$$\mathrm{dst}(\overline{x}) = \gamma \cdot \tanh\left(\alpha \cdot conv(\mathrm{src}, \mathrm{weights}) + \beta \cdot \mathrm{dst}(\overline{x})\right)$$

Example 2

The following pseudo code:

```
attribute attr;
attr.set_output_scale(alpha);
attr.set_post_ops({
        { eltwise={scale=gamma, type=relu, alpha=eta, beta=ignored }
        { sum={scale=beta} },
    });

convolution_forward(src, weights, dst, attr)
```

That would lead to the following:

$$\mathrm{dst}(\overline{x}) = \beta \cdot \mathrm{dst}(\overline{x}) + \gamma \cdot ReLU\left(\alpha \cdot conv(\mathrm{src}, \mathrm{weights}), \eta\right)$$

Algorithms

oneDNN implementations may implement convolution primitives using several different algorithms which can be chosen by the user.

- *Direct* (`dnnl::algorithm::convolution_direct`). The convolution operation is computed directly using SIMD instructions. This also includes implicit GEMM formulations which notably may require workspace.
- *Winograd* (`dnnl::algorithm::convolution_winograd`). This algorithm reduces computational complexity of convolution at the expense of accuracy loss and additional memory operations. The implementation is based on the Fast Algorithms for Convolutional Neural Networks by A. Lavin and S. Gray. The Winograd algorithm often results in the best performance, but it is applicable only to particular shapes. Moreover, Winograd only supports int8 and f32 data types.
- *Auto* (`dnnl::algorithm::convolution_auto`). In this case the library should automatically select the *best* algorithm based on the heuristics that take into account tensor shapes and the number of logical processors available.

API

struct **convolution_forward** : public dnnl::*primitive*

Convolution forward propagation primitive.

Public Functions

convolution_forward()

Default constructor. Produces an empty object.

convolution_forward(const *primitive_desc* &pd)

Constructs a convolution forward propagation primitive.

Parameters
pd – Primitive descriptor for a convolution forward propagation primitive.

struct **primitive_desc** : public dnnl::*primitive_desc*

Primitive descriptor for a convolution forward propagation primitive.

Public Functions

primitive_desc()

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *algorithm* aalgorithm, const *memory*::*desc* &src_desc, const *memory*::*desc* &weights_desc, const *memory*::*desc* &bias_desc, const *memory*::*desc* &dst_desc, const *memory*::*dims* &strides, const *memory*::*dims* &padding_l, const *memory*::*dims* &padding_r, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a convolution forward propagation primitive with bias.

Arrays `strides`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note: All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- **aalgorithm** – Convolution algorithm. Possible values are *dnnl::algorithm::convolution_direct*, *dnnl::algorithm::convolution_winograd*, and *dnnl::algorithm::convolution_auto*.
- **src_desc** – Source memory descriptor.
- **weights_desc** – Weights memory descriptor.
- **bias_desc** – Bias memory descriptor. Passing zero memory descriptor disables the bias term.
- **dst_desc** – Destination memory descriptor.
- **strides** – Strides for each spatial dimension.
- **padding_l** – Vector of padding values for low indices for each spatial dimension `([[front,] top,] left)`.
- **padding_r** – Vector of padding values for high indices for each spatial dimension `([[back,] bottom,] right)`.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *algorithm* aalgorithm, const *memory*::*desc* &src_desc, const *memory*::*desc* &weights_desc, const *memory*::*desc* &dst_desc, const *memory*::*dims* &strides, const *memory*::*dims* &padding_l, const *memory*::*dims* &padding_r, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a convolution forward propagation primitive without bias.

Arrays `strides`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note: All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- **aalgorithm** – Convolution algorithm. Possible values are *dnnl::algorithm::convolution_direct*, *dnnl::algorithm::convolution_winograd*, and *dnnl::algorithm::convolution_auto*.
- **src_desc** – Source memory descriptor.
- **weights_desc** – Weights memory descriptor.
- **dst_desc** – Destination memory descriptor.
- **strides** – Strides for each spatial dimension.

- **padding_l** – Vector of padding values for low indices for each spatial dimension ([[front,] top,] left).
- **padding_r** – Vector of padding values for high indices for each spatial dimension ([[back,] bottom,] right).
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *algorithm* aalgorithm, const *memory*::*desc* &src_desc, const *memory*::*desc* &weights_desc, const *memory*::*desc* &bias_desc, const *memory*::*desc* &dst_desc, const *memory*::*dims* &strides, const *memory*::*dims* &dilates, const *memory*::*dims* &padding_l, const *memory*::*dims* &padding_r, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a convolution forward propagation primitive with bias.

Arrays strides, dilates, padding_l, and padding_r contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note: All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of format_tag.

Parameters

- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- **aalgorithm** – Convolution algorithm. Possible values are *dnnl::algorithm::convolution_direct*, *dnnl::algorithm::convolution_winograd*, and *dnnl::algorithm::convolution_auto*.
- **src_desc** – Source memory descriptor.
- **weights_desc** – Weights memory descriptor.
- **bias_desc** – Bias memory descriptor. Passing zero memory descriptor disables the bias term.
- **dst_desc** – Destination memory descriptor.
- **strides** – Strides for each spatial dimension.
- **dilates** – Dilations for each spatial dimension. A zero value means no dilation in the corresponding dimension.
- **padding_l** – Vector of padding values for low indices for each spatial dimension ([[front,] top,] left).
- **padding_r** – Vector of padding values for high indices for each spatial dimension ([[back,] bottom,] right).
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *algorithm* aalgorithm, const *memory*::*desc* &src_desc, const *memory*::*desc* &weights_desc, const *memory*::*desc* &dst_desc, const *memory*::*dims* &strides, const *memory*::*dims* &dilates, const *memory*::*dims* &padding_l, const *memory*::*dims* &padding_r, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a convolution forward propagation primitive without bias.

Arrays `strides`, `dilates`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note: All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- **aalgorithm** – Convolution algorithm. Possible values are *dnnl::algorithm::convolution_direct*, *dnnl::algorithm::convolution_winograd*, and *dnnl::algorithm::convolution_auto*.
- **src_desc** – Source memory descriptor.
- **weights_desc** – Weights memory descriptor.
- **dst_desc** – Destination memory descriptor.
- **strides** – Strides for each spatial dimension.
- **dilates** – Dilations for each spatial dimension. A zero value means no dilation in the corresponding dimension.
- **padding_l** – Vector of padding values for low indices for each spatial dimension `([[front,] top,] left)`.
- **padding_r** – Vector of padding values for high indices for each spatial dimension `([[back,] bottom,] right)`.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() const

Returns a source memory descriptor.

Returns

Source memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **weights_desc**() const

Returns a weights memory descriptor.

Returns

Weights memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a weights parameter.

memory::*desc* **dst_desc**() const

Returns a destination memory descriptor.

Returns

Destination memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a destination parameter.

memory::*desc* **bias_desc**() const

Returns the bias memory descriptor.

Returns
The bias memory descriptor.

Returns
A zero memory descriptor of the primitive does not have a bias parameter.

algorithm **get_algorithm**() const

Returns an algorithm kind.

Returns
An algorithm kind.

Returns
dnnl::algorithm::undef if the primitive does not have an algorithm parameter.

prop_kind **get_prop_kind**() const

Returns a propagation kind.

Returns
A propagation kind.

Returns
dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

memory::*dims* **get_strides**() const

Returns strides.

Returns
Strides.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a strides parameter.

memory::*dims* **get_dilations**() const

Returns dilations.

Returns
Dilations.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a dilations parameter.

memory::*dims* **get_padding_l**() const

Returns a left padding.

Returns
A left padding.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a left padding parameter.

memory::*dims* **get_padding_r**() const

Returns a right padding.

Returns
A right padding.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a right padding parameter.

struct **convolution_backward_data** : public dnnl::*primitive*

Convolution backward propagation primitive.

Public Functions

convolution_backward_data()

Default constructor. Produces an empty object.

convolution_backward_data(const *primitive_desc* &pd)

Constructs a convolution backward propagation primitive.

Parameters
pd – Primitive descriptor for a convolution backward propagation primitive.

struct **primitive_desc** : public dnnl::*primitive_desc*

Primitive descriptor for a convolution backward propagation primitive.

Public Functions

primitive_desc()

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *algorithm* aalgorithm, const *memory*::*desc* &diff_src_desc, const *memory*::*desc* &weights_desc, const *memory*::*desc* &diff_dst_desc, const *memory*::*dims* &strides, const *memory*::*dims* &padding_l, const *memory*::*dims* &padding_r, const *convolution_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a convolution backward propagation primitive.

Arrays `strides`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note: All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters
- **aengine** – Engine to use.
- **aalgorithm** – Convolution algorithm. Possible values are *dnnl::algorithm::convolution_direct*, *dnnl::algorithm::convolution_winograd*, and *dnnl::algorithm::convolution_auto*.
- **diff_src_desc** – Diff source memory descriptor.
- **weights_desc** – Weights memory descriptor.
- **diff_dst_desc** – Diff destination memory descriptor.
- **strides** – Strides for each spatial dimension.
- **padding_l** – Vector of padding values for low indices for each spatial dimension `([[front,] top,] left)`.
- **padding_r** – Vector of padding values for high indices for each spatial dimension `([[back,] bottom,] right)`.
- **hint_fwd_pd** – Primitive descriptor for a convolution forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.

- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, *algorithm* aalgorithm, const *memory*::*desc* &diff_src_desc, const *memory*::*desc* &weights_desc, const *memory*::*desc* &diff_dst_desc, const *memory*::*dims* &strides, const *memory*::*dims* &dilates, const *memory*::*dims* &padding_l, const *memory*::*dims* &padding_r, const *convolution_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a convolution backward propagation primitive.

Arrays `strides`, `dilates`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note: All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- **aengine** – Engine to use.
- **aalgorithm** – Convolution algorithm. Possible values are *dnnl::algorithm::convolution_direct*, *dnnl::algorithm::convolution_winograd*, and *dnnl::algorithm::convolution_auto*.
- **diff_src_desc** – Diff source memory descriptor.
- **weights_desc** – Weights memory descriptor.
- **diff_dst_desc** – Diff destination memory descriptor.
- **strides** – Strides for each spatial dimension.
- **dilates** – Dilations for each spatial dimension. A zero value means no dilation in the corresponding dimension.
- **padding_l** – Vector of padding values for low indices for each spatial dimension `([[front,] top,] left)`.
- **padding_r** – Vector of padding values for high indices for each spatial dimension `([[back,] bottom,] right)`.
- **hint_fwd_pd** – Primitive descriptor for a convolution forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **diff_src_desc**() const

Returns a diff source memory descriptor.

Returns

Diff source memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a diff source memory with.

memory::*desc* **weights_desc**() const

Returns a weights memory descriptor.

Returns

Weights memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a weights parameter.

memory::*desc* **diff_dst_desc**() const

Returns a diff destination memory descriptor.

Returns
Diff destination memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a diff destination parameter.

algorithm **get_algorithm**() const

Returns an algorithm kind.

Returns
An algorithm kind.

Returns
dnnl::algorithm::undef if the primitive does not have an algorithm parameter.

prop_kind **get_prop_kind**() const

Returns a propagation kind.

Returns
A propagation kind.

Returns
dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

memory::*dims* **get_strides**() const

Returns strides.

Returns
Strides.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a strides parameter.

memory::*dims* **get_dilations**() const

Returns dilations.

Returns
Dilations.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a dilations parameter.

memory::*dims* **get_padding_l**() const

Returns a left padding.

Returns
A left padding.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a left padding parameter.

memory::*dims* **get_padding_r**() const

Returns a right padding.

Returns
A right padding.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a right padding parameter.

struct **convolution_backward_weights** : public dnnl::*primitive*

Convolution weights gradient primitive.

Public Functions

convolution_backward_weights()

Default constructor. Produces an empty object.

convolution_backward_weights(const *primitive_desc* &pd)

Constructs a convolution weights gradient primitive.

Parameters

pd – Primitive descriptor for a convolution weights gradient primitive.

struct **primitive_desc** : public dnnl::*primitive_desc*

Primitive descriptor for a convolution weights gradient primitive.

Public Functions

primitive_desc()

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *algorithm* aalgorithm, const *memory*::*desc* &src_desc, const *memory*::*desc* &diff_weights_desc, const *memory*::*desc* &diff_bias_desc, const *memory*::*desc* &diff_dst_desc, const *memory*::*dims* &strides, const *memory*::*dims* &padding_l, const *memory*::*dims* &padding_r, const *convolution_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a convolution weights gradient primitive with bias.

Arrays `strides`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note: All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- **aengine** – Engine to use.
- **aalgorithm** – Convolution algorithm. Possible values are *dnnl::algorithm::convolution_direct*, *dnnl::algorithm::convolution_winograd*, and *dnnl::algorithm::convolution_auto*.
- **src_desc** – Source memory descriptor.
- **diff_weights_desc** – Diff weights memory descriptor.
- **diff_bias_desc** – Diff bias memory descriptor. Passing zero memory descriptor disables the bias term.
- **diff_dst_desc** – Diff destination memory descriptor.
- **strides** – Strides for each spatial dimension.
- **padding_l** – Vector of padding values for low indices for each spatial dimension `([[front,] top,] left)`.
- **padding_r** – Vector of padding values for high indices for each spatial dimension `([[back,] bottom,] right)`.
- **hint_fwd_pd** – Primitive descriptor for a convolution forward propagation primitive. It is used as a hint for deciding which memory format to use.

- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, *algorithm* aalgorithm, const *memory*::*desc* &src_desc, const *memory*::*desc* &diff_weights_desc, const *memory*::*desc* &diff_dst_desc, const *memory*::*dims* &strides, const *memory*::*dims* &padding_l, const *memory*::*dims* &padding_r, const *convolution_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a convolution weights gradient primitive without bias.

Arrays `strides`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note: All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- **aengine** – Engine to use.
- **aalgorithm** – Convolution algorithm. Possible values are *dnnl::algorithm::convolution_direct*, *dnnl::algorithm::convolution_winograd*, and *dnnl::algorithm::convolution_auto*.
- **src_desc** – Source memory descriptor.
- **diff_weights_desc** – Diff weights memory descriptor.
- **diff_dst_desc** – Diff destination memory descriptor.
- **strides** – Strides for each spatial dimension.
- **padding_l** – Vector of padding values for low indices for each spatial dimension `([[front,] top,] left)`.
- **padding_r** – Vector of padding values for high indices for each spatial dimension `([[back,] bottom,] right)`.
- **hint_fwd_pd** – Primitive descriptor for a convolution forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, *algorithm* aalgorithm, const *memory*::*desc* &src_desc, const *memory*::*desc* &diff_weights_desc, const *memory*::*desc* &diff_bias_desc, const *memory*::*desc* &diff_dst_desc, const *memory*::*dims* &strides, const *memory*::*dims* &dilates, const *memory*::*dims* &padding_l, const *memory*::*dims* &padding_r, const *convolution_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a convolution weights gradient primitive with bias.

Arrays `strides`, `dilates`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note: All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- **aengine** – Engine to use.
- **aalgorithm** – Convolution algorithm. Possible values are *dnnl::algorithm::convolution_direct*, *dnnl::algorithm::convolution_winograd*, and *dnnl::algorithm::convolution_auto*.
- **src_desc** – Source memory descriptor.
- **diff_weights_desc** – Diff weights memory descriptor.
- **diff_bias_desc** – Diff bias memory descriptor. Passing zero memory descriptor disables the bias term.
- **diff_dst_desc** – Diff destination memory descriptor.
- **strides** – Strides for each spatial dimension.
- **dilates** – Dilations for each spatial dimension. A zero value means no dilation in the corresponding dimension.
- **padding_l** – Vector of padding values for low indices for each spatial dimension `([[front,] top,] left)`.
- **padding_r** – Vector of padding values for high indices for each spatial dimension `([[back,] bottom,] right)`.
- **hint_fwd_pd** – Primitive descriptor for a convolution forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, *algorithm* aalgorithm, const *memory*::*desc* &src_desc, const *memory*::*desc* &diff_weights_desc, const *memory*::*desc* &diff_dst_desc, const *memory*::*dims* &strides, const *memory*::*dims* &dilates, const *memory*::*dims* &padding_l, const *memory*::*dims* &padding_r, const *convolution_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a convolution weights gradient primitive without bias.

Arrays `strides`, `dilates`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note: All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- **aengine** – Engine to use.
- **aalgorithm** – Convolution algorithm. Possible values are *dnnl::algorithm::convolution_direct*, *dnnl::algorithm::convolution_winograd*, and *dnnl::algorithm::convolution_auto*.
- **src_desc** – Source memory descriptor.
- **diff_weights_desc** – Diff weights memory descriptor.
- **diff_dst_desc** – Diff destination memory descriptor.
- **strides** – Strides for each spatial dimension.

- **dilates** – Dilations for each spatial dimension. A zero value means no dilation in the corresponding dimension.
- **padding_l** – Vector of padding values for low indices for each spatial dimension `([[front,] top,] left)`.
- **padding_r** – Vector of padding values for high indices for each spatial dimension `([[back,] bottom,] right)`.
- **hint_fwd_pd** – Primitive descriptor for a convolution forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() const

Returns a source memory descriptor.

Returns
Source memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **diff_weights_desc**() const

Returns a diff weights memory descriptor.

Returns
Diff weights memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a diff weights parameter.

memory::*desc* **diff_dst_desc**() const

Returns a diff destination memory descriptor.

Returns
Diff destination memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a diff destination parameter.

memory::*desc* **diff_bias_desc**() const

Returns the diff bias memory descriptor.

Returns
The diff bias memory descriptor.

Returns
A zero memory descriptor of the primitive does not have a diff bias parameter.

algorithm **get_algorithm**() const

Returns an algorithm kind.

Returns
An algorithm kind.

Returns
dnnl::algorithm::undef if the primitive does not have an algorithm parameter.

prop_kind **get_prop_kind**() const

Returns a propagation kind.

Returns
A propagation kind.

Returns
dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

memory::*dims* **get_strides**() const

Returns strides.

Returns
Strides.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a strides parameter.

memory::*dims* **get_dilations**() const

Returns dilations.

Returns
Dilations.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a dilations parameter.

memory::*dims* **get_padding_l**() const

Returns a left padding.

Returns
A left padding.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a left padding parameter.

memory::*dims* **get_padding_r**() const

Returns a right padding.

Returns
A right padding.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a right padding parameter.

struct **deconvolution_forward** : public dnnl::*primitive*

Deconvolution forward propagation primitive.

Public Functions

deconvolution_forward()

Default constructor. Produces an empty object.

deconvolution_forward(const *primitive_desc* &pd)

Constructs a deconvolution forward propagation primitive.

Parameters
pd – Primitive descriptor for a deconvolution forward propagation primitive.

struct **primitive_desc** : public dnnl::*primitive_desc*

Primitive descriptor for a deconvolution forward propagation primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *algorithm* aalgorithm, const *memory*::*desc* &src_desc, const *memory*::*desc* &weights_desc, const *memory*::*desc* &bias_desc, const *memory*::*desc* &dst_desc, const *memory*::*dims* &strides, const *memory*::*dims* &padding_l, const *memory*::*dims* &padding_r, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a deconvolution forward propagation primitive with bias.

Arrays `strides`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note: All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- **aalgorithm** – Deconvolution algorithm: *dnnl::algorithm::deconvolution_direct*, and *dnnl::algorithm::deconvolution_winograd*.
- **src_desc** – Source memory descriptor.
- **weights_desc** – Weights memory descriptor.
- **bias_desc** – Bias memory descriptor. Passing zero memory descriptor disables the bias term.
- **dst_desc** – Destination memory descriptor.
- **strides** – Vector of strides for spatial dimension.
- **padding_l** – Vector of padding values for low indices for each spatial dimension `([[front,] top,] left)`.
- **padding_r** – Vector of padding values for high indices for each spatial dimension `([[back,] bottom,] right)`.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *algorithm* aalgorithm, const *memory*::*desc* &src_desc, const *memory*::*desc* &weights_desc, const *memory*::*desc* &dst_desc, const *memory*::*dims* &strides, const *memory*::*dims* &padding_l, const *memory*::*dims* &padding_r, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a deconvolution forward propagation primitive without bias.

Arrays `strides`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note: All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- **aalgorithm** – Deconvolution algorithm: *dnnl::algorithm::deconvolution_direct*, and *dnnl::algorithm::deconvolution_winograd*.
- **src_desc** – Source memory descriptor.
- **weights_desc** – Weights memory descriptor.
- **dst_desc** – Destination memory descriptor.
- **strides** – Vector of strides for spatial dimension.
- **padding_l** – Vector of padding values for low indices for each spatial dimension `([[front,] top,] left)`.
- **padding_r** – Vector of padding values for high indices for each spatial dimension `([[back,] bottom,] right)`.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *algorithm* aalgorithm, const *memory*::*desc* &src_desc, const *memory*::*desc* &weights_desc, const *memory*::*desc* &bias_desc, const *memory*::*desc* &dst_desc, const *memory*::*dims* &strides, const *memory*::*dims* &dilates, const *memory*::*dims* &padding_l, const *memory*::*dims* &padding_r, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a deconvolution forward propagation primitive with bias.

Arrays `strides`, `dilates`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note: All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- **aalgorithm** – Deconvolution algorithm: *dnnl::algorithm::deconvolution_direct*, and *dnnl::algorithm::deconvolution_winograd*.
- **src_desc** – Source memory descriptor.
- **weights_desc** – Weights memory descriptor.
- **bias_desc** – Bias memory descriptor. Passing zero memory descriptor disables the bias term.
- **dst_desc** – Destination memory descriptor.
- **strides** – Vector of strides for spatial dimension.
- **dilates** – Dilations for each spatial dimension. A zero value means no dilation in the corresponding dimension.

- **padding_l** – Vector of padding values for low indices for each spatial dimension ([[front,] top,] left).
- **padding_r** – Vector of padding values for high indices for each spatial dimension ([[back,] bottom,] right).
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *algorithm* aalgorithm, const *memory*::*desc* &src_desc, const *memory*::*desc* &weights_desc, const *memory*::*desc* &dst_desc, const *memory*::*dims* &strides, const *memory*::*dims* &dilates, const *memory*::*dims* &padding_l, const *memory*::*dims* &padding_r, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a deconvolution forward propagation primitive without bias.

Arrays strides, dilates, padding_l, and padding_r contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note: All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of format_tag.

Parameters

- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- **aalgorithm** – Deconvolution algorithm: *dnnl::algorithm::deconvolution_direct*, and *dnnl::algorithm::deconvolution_winograd*.
- **src_desc** – Source memory descriptor.
- **weights_desc** – Weights memory descriptor.
- **dst_desc** – Destination memory descriptor.
- **strides** – Vector of strides for spatial dimension.
- **dilates** – Dilations for each spatial dimension. A zero value means no dilation in the corresponding dimension.
- **padding_l** – Vector of padding values for low indices for each spatial dimension ([[front,] top,] left).
- **padding_r** – Vector of padding values for high indices for each spatial dimension ([[back,] bottom,] right).
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() const

Returns a source memory descriptor.

Returns

Source memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **weights_desc**() const

Returns a weights memory descriptor.

Returns
Weights memory descriptor.
Returns
A zero memory descriptor if the primitive does not have a weights parameter.

memory::*desc* **dst_desc**() const

Returns a destination memory descriptor.
Returns
Destination memory descriptor.
Returns
A zero memory descriptor if the primitive does not have a destination parameter.

memory::*desc* **bias_desc**() const

Returns the bias memory descriptor.
Returns
The bias memory descriptor.
Returns
A zero memory descriptor of the primitive does not have a bias parameter.

algorithm **get_algorithm**() const

Returns an algorithm kind.
Returns
An algorithm kind.
Returns
dnnl::algorithm::undef if the primitive does not have an algorithm parameter.

prop_kind **get_prop_kind**() const

Returns a propagation kind.
Returns
A propagation kind.
Returns
dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

memory::*dims* **get_strides**() const

Returns strides.
Returns
Strides.
Returns
An empty *dnnl::memory::dims* if the primitive does not have a strides parameter.

memory::*dims* **get_dilations**() const

Returns dilations.
Returns
Dilations.
Returns
An empty *dnnl::memory::dims* if the primitive does not have a dilations parameter.

memory::*dims* **get_padding_l**() const

Returns a left padding.
Returns
A left padding.
Returns
An empty *dnnl::memory::dims* if the primitive does not have a left padding parameter.

memory::*dims* **get_padding_r**() const

Returns a right padding.

Returns
A right padding.
Returns
An empty *dnnl::memory::dims* if the primitive does not have a right padding parameter.

struct **deconvolution_backward_data** : public dnnl::*primitive*

Deconvolution backward propagation primitive.

Public Functions

deconvolution_backward_data()

Default constructor. Produces an empty object.

deconvolution_backward_data(const *primitive_desc* &pd)

Constructs a deconvolution backward propagation primitive.

Parameters
pd – Primitive descriptor for a deconvolution backward propagation primitive.

struct **primitive_desc** : public dnnl::*primitive_desc*

Primitive descriptor for a deconvolution backward propagation primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *algorithm* aalgorithm, const *memory*::*desc* &diff_src_desc, const *memory*::*desc* &weights_desc, const *memory*::*desc* &diff_dst_desc, const *memory*::*dims* &strides, const *memory*::*dims* &padding_l, const *memory*::*dims* &padding_r, const *deconvolution_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a deconvolution backward propagation primitive.

Arrays `strides`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note: All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters
- **aengine** – Engine to use.
- **aalgorithm** – Deconvolution algorithm (*dnnl::algorithm::convolution_direct*, *dnnl::algorithm::convolution_winograd*).
- **diff_src_desc** – Diff source memory descriptor.
- **weights_desc** – Weights memory descriptor.
- **diff_dst_desc** – Diff destination memory descriptor.
- **strides** – Strides for each spatial dimension.

- **padding_l** – Vector of padding values for low indices for each spatial dimension (`[[front,] top,] left`).
- **padding_r** – Vector of padding values for high indices for each spatial dimension (`[[back,] bottom,] right`).
- **hint_fwd_pd** – Primitive descriptor for a deconvolution forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, *algorithm* aalgorithm, const *memory*::*desc* &diff_src_desc, const *memory*::*desc* &weights_desc, const *memory*::*desc* &diff_dst_desc, const *memory*::*dims* &strides, const *memory*::*dims* &dilates, const *memory*::*dims* &padding_l, const *memory*::*dims* &padding_r, const *deconvolution_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a deconvolution backward propagation primitive.

Arrays `strides`, `dilates`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note: All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- **aengine** – Engine to use.
- **aalgorithm** – Deconvolution algorithm (*dnnl::algorithm::convolution_direct*, *dnnl::algorithm::convolution_winograd*).
- **diff_src_desc** – Diff source memory descriptor.
- **weights_desc** – Weights memory descriptor.
- **diff_dst_desc** – Diff destination memory descriptor.
- **strides** – Strides for each spatial dimension.
- **dilates** – Dilations for each spatial dimension. A zero value means no dilation in the corresponding dimension.
- **padding_l** – Vector of padding values for low indices for each spatial dimension (`[[front,] top,] left`).
- **padding_r** – Vector of padding values for high indices for each spatial dimension (`[[back,] bottom,] right`).
- **hint_fwd_pd** – Primitive descriptor for a deconvolution forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **diff_src_desc**() const

Returns a diff source memory descriptor.

Returns

Diff source memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a diff source memory with.

memory::*desc* **weights_desc**() const

Returns a weights memory descriptor.

Returns
Weights memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a weights parameter.

memory::*desc* **diff_dst_desc**() const

Returns a diff destination memory descriptor.

Returns
Diff destination memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a diff destination parameter.

algorithm **get_algorithm**() const

Returns an algorithm kind.

Returns
An algorithm kind.

Returns
dnnl::algorithm::undef if the primitive does not have an algorithm parameter.

prop_kind **get_prop_kind**() const

Returns a propagation kind.

Returns
A propagation kind.

Returns
dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

memory::*dims* **get_strides**() const

Returns strides.

Returns
Strides.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a strides parameter.

memory::*dims* **get_dilations**() const

Returns dilations.

Returns
Dilations.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a dilations parameter.

memory::*dims* **get_padding_l**() const

Returns a left padding.

Returns
A left padding.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a left padding parameter.

memory::*dims* **get_padding_r**() const

Returns a right padding.

Returns
A right padding.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a right padding parameter.

struct **deconvolution_backward_weights** : public dnnl::*primitive*

Deconvolution weights gradient primitive.

Public Functions

deconvolution_backward_weights()

Default constructor. Produces an empty object.

deconvolution_backward_weights(const *primitive_desc* &pd)

Constructs a deconvolution weights gradient primitive.

Parameters
pd – Primitive descriptor for a deconvolution weights gradient primitive.

struct **primitive_desc** : public dnnl::*primitive_desc*

Primitive descriptor for a deconvolution weights gradient primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *algorithm* aalgorithm, const *memory*::*desc* &src_desc, const *memory*::*desc* &diff_weights_desc, const *memory*::*desc* &diff_bias_desc, const *memory*::*desc* &diff_dst_desc, const *memory*::*dims* &strides, const *memory*::*dims* &padding_l, const *memory*::*dims* &padding_r, const *deconvolution_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a deconvolution weights gradient primitive with bias.

Arrays `strides`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note: All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters
- **aengine** – Engine to use.
- **aalgorithm** – Deconvolution algorithm. Possible values are *dnnl::algorithm::deconvolution_direct*, and *dnnl::algorithm::deconvolution_winograd*.
- **src_desc** – Source memory descriptor.
- **diff_weights_desc** – Diff weights memory descriptor.
- **diff_bias_desc** – Diff bias memory descriptor. Passing zero memory descriptor disables the bias term.
- **diff_dst_desc** – Diff destination memory descriptor.
- **strides** – Strides for each spatial dimension.

- **padding_l** – Vector of padding values for low indices for each spatial dimension (`[[front,] top,] left`).
- **padding_r** – Vector of padding values for high indices for each spatial dimension (`[[back,] bottom,] right`).
- **hint_fwd_pd** – Primitive descriptor for a deconvolution forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, *algorithm* aalgorithm, const *memory*::*desc* &src_desc, const *memory*::*desc* &diff_weights_desc, const *memory*::*desc* &diff_dst_desc, const *memory*::*dims* &strides, const *memory*::*dims* &padding_l, const *memory*::*dims* &padding_r, const *deconvolution_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a deconvolution weights gradient primitive without bias.

Arrays `strides`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note: All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- **aengine** – Engine to use.
- **aalgorithm** – Deconvolution algorithm. Possible values are *dnnl::algorithm::deconvolution_direct*, and *dnnl::algorithm::deconvolution_winograd*.
- **src_desc** – Source memory descriptor.
- **diff_weights_desc** – Diff weights memory descriptor.
- **diff_dst_desc** – Diff destination memory descriptor.
- **strides** – Strides for each spatial dimension.
- **padding_l** – Vector of padding values for low indices for each spatial dimension (`[[front,] top,] left`).
- **padding_r** – Vector of padding values for high indices for each spatial dimension (`[[back,] bottom,] right`).
- **hint_fwd_pd** – Primitive descriptor for a deconvolution forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, *algorithm* aalgorithm, const *memory*::*desc* &src_desc, const *memory*::*desc* &diff_weights_desc, const *memory*::*desc* &diff_bias_desc, const *memory*::*desc* &diff_dst_desc, const *memory*::*dims* &strides, const *memory*::*dims* &dilates, const *memory*::*dims* &padding_l, const *memory*::*dims* &padding_r, const *deconvolution_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a deconvolution weights gradient primitive with bias.

Arrays `strides`, `dilates`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note: All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- **aengine** – Engine to use.
- **aalgorithm** – Deconvolution algorithm. Possible values are *dnnl::algorithm::deconvolution_direct*, and *dnnl::algorithm::deconvolution_winograd*.
- **src_desc** – Source memory descriptor.
- **diff_weights_desc** – Diff weights memory descriptor.
- **diff_bias_desc** – Diff bias memory descriptor. Passing zero memory descriptor disables the bias term.
- **diff_dst_desc** – Diff destination memory descriptor.
- **strides** – Strides for each spatial dimension.
- **dilates** – Dilations for each spatial dimension. A zero value means no dilation in the corresponding dimension.
- **padding_l** – Vector of padding values for low indices for each spatial dimension `([[front,] top,] left)`.
- **padding_r** – Vector of padding values for high indices for each spatial dimension `([[back,] bottom,] right)`.
- **hint_fwd_pd** – Primitive descriptor for a deconvolution forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, *algorithm* aalgorithm, const *memory*::*desc* &src_desc, const *memory*::*desc* &diff_weights_desc, const *memory*::*desc* &diff_dst_desc, const *memory*::*dims* &strides, const *memory*::*dims* &dilates, const *memory*::*dims* &padding_l, const *memory*::*dims* &padding_r, const *deconvolution_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a deconvolution weights gradient primitive without bias.

Arrays `strides`, `dilates`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note: All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- **aengine** – Engine to use.
- **aalgorithm** – Deconvolution algorithm. Possible values are *dnnl::algorithm::deconvolution_direct*, and *dnnl::algorithm::deconvolution_winograd*.
- **src_desc** – Source memory descriptor.
- **diff_weights_desc** – Diff weights memory descriptor.

- **diff_dst_desc** – Diff destination memory descriptor.
- **strides** – Strides for each spatial dimension.
- **dilates** – Dilations for each spatial dimension. A zero value means no dilation in the corresponding dimension.
- **padding_l** – Vector of padding values for low indices for each spatial dimension `([[front,] top,] left)`.
- **padding_r** – Vector of padding values for high indices for each spatial dimension `([[back,] bottom,] right)`.
- **hint_fwd_pd** – Primitive descriptor for a deconvolution forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() const

Returns a source memory descriptor.

Returns

Source memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **diff_weights_desc**() const

Returns a diff weights memory descriptor.

Returns

Diff weights memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a diff weights parameter.

memory::*desc* **diff_dst_desc**() const

Returns a diff destination memory descriptor.

Returns

Diff destination memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a diff destination parameter.

memory::*desc* **diff_bias_desc**() const

Returns the diff bias memory descriptor.

Returns

The diff bias memory descriptor.

Returns

A zero memory descriptor of the primitive does not have a diff bias parameter.

algorithm **get_algorithm**() const

Returns an algorithm kind.

Returns

An algorithm kind.

Returns

dnnl::algorithm::undef if the primitive does not have an algorithm parameter.

prop_kind **get_prop_kind**() const

Returns a propagation kind.

Returns

A propagation kind.

Returns

dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

memory::*dims* **get_strides**() const

Returns strides.

Returns
Strides.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a strides parameter.

memory::*dims* **get_dilations**() const

Returns dilations.

Returns
Dilations.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a dilations parameter.

memory::*dims* **get_padding_l**() const

Returns a left padding.

Returns
A left padding.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a left padding parameter.

memory::*dims* **get_padding_r**() const

Returns a right padding.

Returns
A right padding.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a right padding parameter.

4.5.7 Elementwise

The elementwise primitive applies an operation to every element of the tensor. Variable names follow the standard *Conventions*.

$$\mathrm{dst}(\overline{x}) = Operation(\mathrm{src}(\overline{x})),$$

for $\overline{x} = (x_0, \ldots, x_n)$.

Forward

The following forward operations are supported. Here s and d denote src and dst, tensor values respectively.

Elementwise algorithm	Forward formula
eltwise_abs	$d = \begin{cases} s & \text{if } s > 0 \\ -s & \text{if } s \leq 0 \end{cases}$
eltwise_clip, *eltwise_clip_use_dst_for_bwd*	$d = \begin{cases} \beta & \text{if } s > \beta \geq \alpha \\ s & \text{if } \alpha < s \leq \beta \\ \alpha & \text{if } s \leq \alpha \end{cases}$
eltwise_elu, *eltwise_elu_use_dst_for_bwd*	$d = \begin{cases} s & \text{if } s > 0 \\ \alpha(e^s - 1) & \text{if } s \leq 0 \end{cases}$
eltwise_exp, *eltwise_exp_use_dst_for_bwd*	$d = e^s$
eltwise_gelu_erf	$d = 0.5s(1 + erf[\frac{s}{\sqrt{2}}])$
eltwise_gelu_tanh	$d = 0.5s(1 + tanh[\sqrt{\frac{2}{\pi}}(s + 0.044715s^3)])$
eltwise_hardsigmoid	$d = \max(0, \min(1, \alpha s + \beta))$
eltwise_hardswish	$d = s \cdot hardsigmoid(s)$
eltwise_linear	$d = \alpha s + \beta$
eltwise_log	$d = \log_e s$
eltwise_logistic, *eltwise_logistic_use_dst_for_bwd*	$d = \frac{1}{1+e^{-s}}$
eltwise_mish	$d = s \cdot \tanh(\log_e(1 + e^s))$
eltwise_pow	$d = \alpha s^\beta$
eltwise_relu, *eltwise_relu_use_dst_for_bwd*	$d = \begin{cases} s & \text{if } s > 0 \\ \alpha s & \text{if } s \leq 0 \end{cases}$
eltwise_round	$d = round(s)$
eltwise_soft_relu	$d = \frac{1}{\alpha}\log_e(1 + e^{\alpha s})$
eltwise_sqrt, *eltwise_sqrt_use_dst_for_bwd*	$d = \sqrt{s}$
eltwise_square	$d = s^2$
eltwise_swish	$d = \frac{s}{1+e^{-\alpha s}}$
eltwise_tanh, *eltwise_tanh_use_dst_for_bwd*	$d = \tanh s$

Backward

The backward propagation computes $\mathrm{diff\_src}(\bar{s})$, based on $\mathrm{diff\_dst}(\bar{s})$ and $\mathrm{src}(\bar{s})$. However, some operations support a computation using $\mathrm{dst}(\bar{s})$ memory produced during forward propagation. Refer to the table above for a list of operations supporting destination as input memory and the corresponding formulas.

The following backward operations are supported. Here s, d, ds and dd denote src, dst, $\mathrm{diff\_src}$, and a $\mathrm{diff\_dst}$ tensor values respectively.

Elementwise algorithm	Backward formula
eltwise_abs	$ds = \begin{cases} dd & \text{if } s > 0 \\ -dd & \text{if } s < 0 \\ 0 & \text{if } s = 0 \end{cases}$
eltwise_clip	$ds = \begin{cases} dd & \text{if } \alpha < s < \beta \\ 0 & \text{otherwise} \end{cases}$
eltwise_clip_u	$ds = \begin{cases} dd & \text{if } \alpha < d < \beta \\ 0 & \text{otherwise} \end{cases}$
eltwise_elu	$ds = \begin{cases} dd & \text{if } s > 0 \\ dd \cdot \alpha e^s & \text{if } s \leq 0 \end{cases}$
eltwise_elu_us	$ds = \begin{cases} dd & \text{if } d > 0 \\ dd \cdot (d + \alpha) & \text{if } d \leq 0 \end{cases}$ only if $\alpha \geq 0$
eltwise_exp	$ds = dd \cdot e^s$
eltwise_exp_us	$ds = dd \cdot d$
eltwise_gelu_e	$ds = dd \cdot \left(0.5 + 0.5 \operatorname{erf}\left(\frac{s}{\sqrt{2}}\right) + \frac{s}{\sqrt{2\pi}} e^{-0.5 s^2}\right)$
eltwise_gelu_t	$ds = dd \cdot 0.5(1 + \tanh[\sqrt{\frac{2}{\pi}}(s + 0.044715 s^3)]) \cdot (1 + \sqrt{\frac{2}{\pi}}(s + 0.134145 s^3) \cdot (1 - \tanh[\sqrt{\frac{2}{\pi}}(s + 0.044715 s^3)]))$
eltwise_hardsi	$ds = \begin{cases} dd \cdot \alpha & \text{if } 0 < \alpha s + \beta < 1 \\ 0 & \text{otherwise} \end{cases}$
eltwise_hardsw	$ds = \begin{cases} dd \cdot (2\alpha + \beta) & \text{if } 0 < \alpha s + \beta < 1 \\ dd & \text{if } \alpha \cdot s + \beta \geq 1 \\ 0 & \text{otherwise} \end{cases}$
eltwise_linear	$ds = \alpha \cdot dd$
eltwise_log	$ds = \frac{dd}{s}$
eltwise_logist	$ds = \frac{dd}{1+e^{-s}} \cdot (1 - \frac{1}{1+e^{-s}})$
eltwise_mish	$ds = dd \cdot \frac{e^s \cdot \omega}{\delta^2}$ with $\omega = e^{3s} + 4 \cdot e^{2s} + e^s \cdot (4 \cdot s + 6) + 4 \cdot (s + 1)$ and $\delta = e^{2s} + 2 \cdot e^s + 2$
eltwise_logist	$ds = dd \cdot d \cdot (1 - d)$
eltwise_pow	$ds = dd \cdot \alpha \beta s^{\beta - 1}$
eltwise_relu	$ds = \begin{cases} dd & \text{if } s > 0 \\ \alpha \cdot dd & \text{if } s \leq 0 \end{cases}$
eltwise_relu_u	$ds = \begin{cases} dd & \text{if } d > 0 \\ \alpha \cdot dd & \text{if } d \leq 0 \end{cases}$ only if $alpha \geq 0$
eltwise_soft_r	$ds = \frac{dd}{1+e^{-\alpha s}}$
eltwise_sqrt	$ds = \frac{dd}{2\sqrt{s}}$
eltwise_sqrt_u	$ds = \frac{dd}{2d}$
eltwise_square	$ds = dd \cdot 2s$
eltwise_swish	$ds = \frac{dd}{1+e^{-\alpha s}}(1 + \alpha s(1 - \frac{1}{1+e^{-\alpha s}}))$
eltwise_tanh	$ds = dd \cdot (1 - \tanh^2 s)$
eltwise_tanh_u	$ds = dd \cdot (1 - d^2)$

Difference Between Forward Training and Forward Inference

There is no difference between the #dnnl_forward_training and #dnnl_forward_inference propagation kinds.

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	*DNNL_ARG_SRC*
dst	*DNNL_ARG_DST*
diff_src	*DNNL_ARG_DIFF_SRC*
diff_dst	*DNNL_ARG_DIFF_DST*

Operation Details

1. The *dnnl::eltwise_forward::primitive_desc* and *dnnl::eltwise_backward::primitive_desc* constructors take both parameters α, and β. These parameters are ignored if they are unused by the algorithm.
2. The memory format and data type for src and dst are assumed to be the same. The same holds for $\mathrm{diff\_src}$ and $\mathrm{diff\_dst}$.
3. Both forward and backward propagation support in-place operations, meaning that src can be used as input and output for forward propagation, and $\mathrm{diff\_dst}$ can be used as input and output for backward propagation. In case of an in-place operation, the original data will be overwritten. Note, however, that some algorithms for backward propagation require original src, hence the corresponding forward propagation should not be performed in-place for those algorithms. Algorithms that use dst for backward propagation can be safely done in-place.
4. For some operations it might be beneficial to compute backward propagation based on $\mathrm{dst}(\overline{s})$, rather than on $\mathrm{src}(\overline{s})$, for improved performance.

Note: For operations supporting destination memory as input, dst can be used instead of src when backward propagation is computed. This enables several performance optimizations (see the tips below).

Data Type Support

The eltwise primitive should support the following combinations of data types.

Note: Here we abbreviate data types names for readability. For example, *dnnl::memory::data_type::f32* is abbreviated to *f32*.

Propagation	Source / Destination	Intermediate data type
forward / backward	*f32*, *bf16*	*f32*
forward	*f16*	*f16*
forward	*s32* / *s8* / *u8*	*f32*

Here the intermediate data type means that the values coming in are first converted to the intermediate data type, then the operation is applied, and finally the result is converted to the output data type.

Data Representation

The eltwise primitive works with arbitrary data tensors. There is no special meaning associated with any logical dimensions.

Post-ops and Attributes

Type	Operation	Description	Restrictions
Post-op	*Binary*	Applies a binary operation to the result	

API

struct **eltwise_forward** : public dnnl::*primitive*

Elementwise unary operation forward propagation primitive.

Public Functions

eltwise_forward()

Default constructor. Produces an empty object.

eltwise_forward(const *primitive_desc* &pd)

Constructs an eltwise forward propagation primitive.

Parameters

pd – Primitive descriptor for an eltwise forward propagation primitive.

struct **primitive_desc** : public dnnl::*primitive_desc*

Primitive descriptor for an elementwise forward propagation primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *algorithm* aalgorithm, const *memory*::*desc* &src_desc, const *memory*::*desc* &dst_desc, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for an elementwise forward propagation primitive.

Parameters

- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- **aalgorithm** – Elementwise algorithm kind.
- **src_desc** – Source memory descriptor.

- **dst_desc** – Destination memory descriptor.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *algorithm* aalgorithm, const *memory*::*desc* &src_desc, const *memory*::*desc* &dst_desc, float alpha, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for an elementwise forward propagation primitive with an alpha parameter.

Parameters

- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- **aalgorithm** – Elementwise algorithm kind.
- **src_desc** – Source memory descriptor.
- **dst_desc** – Destination memory descriptor.
- **alpha** – The alpha parameter for the elementwise operation. Specific meaning depends on the algorithm.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *algorithm* aalgorithm, const *memory*::*desc* &src_desc, const *memory*::*desc* &dst_desc, float alpha, float beta, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for an elementwise forward propagation primitive with an alpha and beta parameters.

Parameters

- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- **aalgorithm** – Elementwise algorithm kind.
- **src_desc** – Source memory descriptor.
- **dst_desc** – Destination memory descriptor.
- **alpha** – The alpha parameter for the elementwise operation. Specific meaning depends on the algorithm.
- **beta** – The beta parameter for the elementwise operation. Specific meaning depends on the algorithm.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() const

Returns a source memory descriptor.

Returns

Source memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **dst_desc**() const

Returns a destination memory descriptor.

Returns
Destination memory descriptor.
Returns
A zero memory descriptor if the primitive does not have a destination parameter.

dnnl::*algorithm* **get_algorithm**() const

Returns an algorithm kind.
Returns
An algorithm kind.
Returns
dnnl::algorithm::undef if the primitive does not have an algorithm parameter.

dnnl::*prop_kind* **get_prop_kind**() const

Returns a propagation kind.
Returns
A propagation kind.
Returns
dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

float **get_alpha**() const

Returns an alpha.
Returns
An alpha.
Returns
Zero if the primitive does not have an alpha parameter.

float **get_beta**() const

Returns a beta.
Returns
A beta.
Returns
Zero if the primitive does not have a beta parameter.

struct **eltwise_backward** : public dnnl::*primitive*

Elementwise unary operation backward propagation primitive.

See also:

eltwise_forward

Public Functions

eltwise_backward()

Default constructor. Produces an empty object.

eltwise_backward(const *primitive_desc* &pd)

Constructs an eltwise backward propagation primitive.

Parameters
pd – Primitive descriptor for an eltwise backward propagation primitive.

struct **primitive_desc** : public dnnl::*primitive_desc*

Primitive descriptor for eltwise backward propagation.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *algorithm* aalgorithm, const *memory*::*desc* &diff_src_desc, const *memory*::*desc* &diff_dst_desc, const *memory*::*desc* &data_desc, const *eltwise_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for an elementwise backward propagation primitive with an alpha parameter.

Parameters

- **aengine** – Engine to use.
- **aalgorithm** – Elementwise algorithm kind.
- **diff_src_desc** – Diff source memory descriptor.
- **diff_dst_desc** – Diff destination memory descriptor.
- **data_desc** – Destination memory descriptor if one of the "use_dst_for_bwd" algorithms are used (such as *dnnl::algorithm::eltwise_relu_use_dst_for_bwd*), source memory descriptor otherwise.
- **hint_fwd_pd** – Primitive descriptor for an elementwise forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, *algorithm* aalgorithm, const *memory*::*desc* &diff_src_desc, const *memory*::*desc* &diff_dst_desc, const *memory*::*desc* &data_desc, float alpha, const *eltwise_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for an elementwise backward propagation primitive with an alpha parameter.

Parameters

- **aengine** – Engine to use.
- **aalgorithm** – Elementwise algorithm kind.
- **diff_src_desc** – Diff source memory descriptor.
- **diff_dst_desc** – Diff destination memory descriptor.
- **data_desc** – Destination memory descriptor if one of the "use_dst_for_bwd" algorithms are used (such as *dnnl::algorithm::eltwise_relu_use_dst_for_bwd*), source memory descriptor otherwise.
- **alpha** – The alpha parameter for the elementwise operation. Specific meaning depends on the algorithm.
- **hint_fwd_pd** – Primitive descriptor for an elementwise forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, *algorithm* aalgorithm, const *memory*::*desc* &diff_src_desc, const *memory*::*desc* &diff_dst_desc, const *memory*::*desc* &data_desc, float alpha, float beta, const *eltwise_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for an elementwise backward propagation primitive with an alpha

and beta parameters.

Parameters

- **aengine** – Engine to use.
- **aalgorithm** – Elementwise algorithm kind.
- **diff_src_desc** – Diff source memory descriptor.
- **diff_dst_desc** – Diff destination memory descriptor.
- **data_desc** – Destination memory descriptor if one of the "use_dst_for_bwd" algorithms are used (such as *dnnl::algorithm::eltwise_relu_use_dst_for_bwd*), source memory descriptor otherwise.
- **alpha** – The alpha parameter for the elementwise operation. Specific meaning depends on the algorithm.
- **beta** – The beta parameter for the elementwise operation. Specific meaning depends on the algorithm.
- **hint_fwd_pd** – Primitive descriptor for an elementwise forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() const

Returns a source memory descriptor.

Returns

Source memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **diff_src_desc**() const

Returns a diff source memory descriptor.

Returns

Diff source memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a diff source memory with.

memory::*desc* **diff_dst_desc**() const

Returns a diff destination memory descriptor.

Returns

Diff destination memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a diff destination parameter.

dnnl::*algorithm* **get_algorithm**() const

Returns an algorithm kind.

Returns

An algorithm kind.

Returns

dnnl::algorithm::undef if the primitive does not have an algorithm parameter.

dnnl::*prop_kind* **get_prop_kind**() const

Returns a propagation kind.

Returns

A propagation kind.

Returns

dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

float **get_alpha**() const

Returns an alpha.

Returns

An alpha.

Returns

Zero if the primitive does not have an alpha parameter.

float **get_beta**() const

Returns a beta.

Returns

A beta.

Returns

Zero if the primitive does not have a beta parameter.

4.5.8 Inner Product

The inner product primitive (sometimes called *fully connected layer*) treats each activation in the minibatch as a vector and computes its product with a weights 2D tensor producing a 2D tensor as an output.

Forward

Let src, weights, bias and dst be $N \times IC$, $OC \times IC$, OC, and $N \times OC$ tensors, respectively. Variable names follow the standard *Conventions*. Then:

$$\mathrm{dst}(n, oc) = \mathrm{bias}(oc) + \sum_{ic=0}^{IC-1} \mathrm{src}(n, ic) \cdot \mathrm{weights}(oc, ic)$$

In cases where the src and weights tensors have spatial dimensions, they are flattened to 2D. For example, if they are 4D $N \times IC' \times IH \times IW$ and $OC \times IC' \times KH \times KW$ tensors, then the formula above is applied with $IC = IC' \cdot IH \cdot IW$. In such cases, the src and weights tensors must have equal spatial dimensions (e.g. $KH = IH$ and $KW = IW$ for 4D tensors).

Difference Between Forward Training and Forward Inference

There is no difference between the *forward_training* and *forward_inference* propagation kinds.

Backward

The backward propagation computes diff_src based on diff_dst and weights.

The weights update computes diff_weights and diff_bias based on diff_dst and src.

Note: The *optimized* memory formats src and weights might be different on forward propagation, backward propagation, and weights update.

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	*DNNL_ARG_SRC*
weights	*DNNL_ARG_WEIGHTS*
bias	*DNNL_ARG_BIAS*
dst	*DNNL_ARG_DST*
diff_src	*DNNL_ARG_DIFF_SRC*
diff_weights	*DNNL_ARG_DIFF_WEIGHTS*
diff_bias	*DNNL_ARG_DIFF_BIAS*
diff_dst	*DNNL_ARG_DIFF_DST*

Operation Details

N/A

Data Types Support

Inner product primitive supports the following combination of data types for source, destination, weights, and bias.

Note: Here we abbreviate data types names for readability. For example, *dnnl::memory::data_type::f32* is abbreviated to *f32*.

Propagation	Source	Weights	Destination	Bias
forward / backward	*f32*	*f32*	*f32*	*f32*
forward	*f16*	*f16*	*f16*	*f16*
forward	*u8*, *s8*	*s8*	*u8*, *s8*, *s32*, *f32*	*u8*, *s8*, *s32*, *f32*
forward	*bf16*	*bf16*	*f32*, *bf16*	*f32*, *bf16*
backward	*f32*, *bf16*	*bf16*	*bf16*	
weights update	*bf16*	*f32*, *bf16*	*bf16*	*f32*, *bf16*

Data Representation

Like other CNN primitives, the inner product primitive expects the following tensors:

Spatial	Source	Destination	Weights
1D	$N \times C \times W$	$N \times C$	$OC \times IC \times KW$
2D	$N \times C \times H \times W$	$N \times C$	$OC \times IC \times KH \times KW$
3D	$N \times C \times D \times H \times W$	$N \times C$	$OC \times IC \times KD \times KH \times KW$

Memory format of data and weights memory objects is critical for inner product primitive performance. In the oneDNN programming model, inner product primitive is one of the few primitives that support the placeholder format *any* and can define data and weight memory objects formats based on the primitive parameters. When using *any* it is necessary

to first create an inner product primitive descriptor and then query it for the actual data and weight memory objects formats.

The table below shows the combinations for which **plain** memory formats the inner product primitive is optimized for. For the destination tensor (which is always $N \times C$) the memory format is always *nc* (*ab*).

Spatial	Source / Weights logical tensor	Implementation optimized for memory formats
0D	NC / OI	*nc* (*ab*) / *oi* (*ab*)
0D	NC / OI	*nc* (*ab*) / *io* (*ba*)
1D	NCW / OIW	*ncw* (*abc*) / *oiw* (*abc*)
1D	NCW / OIW	*nwc* (*acb*) / *wio* (*cba*)
2D	NCHW / OIHW	*nchw* (*abcd*) / *oihw* (*abcd*)
2D	NCHW / OIHW	*nhwc* (*acdb*) / *hwio* (*cdba*)
3D	NCDHW / OIDHW	*ncdhw* (*abcde*) / *oidhw* (*abcde*)
3D	NCDHW / OIDHW	*ndhwc* (*acdeb*) / *dhwio* (*cdeba*)

Post-ops and Attributes

The following post-ops should be supported by inner product primitives:

Type	Operation	Description	Restrictions
Attribute	*Scales*	Sets scale(s) for the corresponding tensor(s)	Int8 computations only
Attribute	*Zero points*	Sets zero point(s) for the corresponding tensors	Int8 computations only
Post-op	*Eltwise*	Applies an elementwise operation to the result	
Post-op	*Binary*	Applies a binary operation to the result	
Post-op	*Sum*	Adds the operation result to the destination tensor instead of overwriting it	

API

struct **inner_product_forward** : public dnnl::*primitive*

Inner product forward propagation primitive.

Public Functions

inner_product_forward()

Default constructor. Produces an empty object.

inner_product_forward(const *primitive_desc* &pd)

Constructs an inner product forward propagation primitive.

Parameters

pd – Primitive descriptor for an inner product forward propagation primitive.

struct **primitive_desc** : public dnnl::*primitive_desc*

Primitive descriptor for an inner product forward propagation primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, const *memory*::*desc* &src_desc, const *memory*::*desc* &weights_desc, const *memory*::*desc* &bias_desc, const *memory*::*desc* &dst_desc, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for an inner product forward propagation primitive with bias.

Note: All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- **src_desc** – Memory descriptor for src.
- **weights_desc** – Memory descriptor for weights.
- **bias_desc** – Memory descriptor for bias.
- **dst_desc** – Memory descriptor for dst.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, const *memory*::*desc* &src_desc, const *memory*::*desc* &weights_desc, const *memory*::*desc* &dst_desc, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for an inner product forward propagation primitive.

Note: All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- **src_desc** – Memory descriptor for src.
- **weights_desc** – Memory descriptor for weights.
- **dst_desc** – Memory descriptor for dst.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() const

Returns a source memory descriptor.

Returns

Source memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **weights_desc**() const

Returns a weights memory descriptor.

Returns
Weights memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a weights parameter.

memory::*desc* **dst_desc**() const

Returns a destination memory descriptor.

Returns
Destination memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a destination parameter.

memory::*desc* **bias_desc**() const

Returns the bias memory descriptor.

Returns
The bias memory descriptor.

Returns
A zero memory descriptor of the primitive does not have a bias parameter.

prop_kind **get_prop_kind**() const

Returns a propagation kind.

Returns
A propagation kind.

Returns
dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

struct **inner_product_backward_data** : public dnnl::*primitive*

Inner product backward propagation primitive.

Public Functions

inner_product_backward_data()

Default constructor. Produces an empty object.

inner_product_backward_data(const *primitive_desc* &pd)

Constructs an inner product backward propagation primitive.

Parameters
pd – Primitive descriptor for an inner product backward propagation primitive.

struct **primitive_desc** : public dnnl::*primitive_desc*

Primitive descriptor for an inner product backward propagation primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, const *memory*::*desc* &diff_src_desc, const *memory*::*desc* &weights_desc, const *memory*::*desc* &diff_dst_desc, const *inner_product_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for an inner product backward propagation primitive.

Note: All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- **aengine** – Engine to use.
- **diff_src_desc** – Memory descriptor for diff src.
- **weights_desc** – Memory descriptor for weights.
- **diff_dst_desc** – Memory descriptor for diff dst.
- **hint_fwd_pd** – Primitive descriptor for an inner product forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **diff_src_desc**() const

Returns a diff source memory descriptor.

Returns
Diff source memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a diff source memory with.

memory::*desc* **weights_desc**() const

Returns a weights memory descriptor.

Returns
Weights memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a weights parameter.

memory::*desc* **diff_dst_desc**() const

Returns a diff destination memory descriptor.

Returns
Diff destination memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a diff destination parameter.

prop_kind **get_prop_kind**() const

Returns a propagation kind.

Returns
A propagation kind.

Returns
dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

struct **inner_product_backward_weights** : public dnnl::*primitive*

Inner product weights gradient primitive.

Public Functions

inner_product_backward_weights()

Default constructor. Produces an empty object.

inner_product_backward_weights(const *primitive_desc* &pd)

Constructs an inner product weights gradient primitive.

Parameters
pd – Primitive descriptor for an inner product weights gradient primitive.

struct **primitive_desc** : public dnnl::*primitive_desc*

Primitive descriptor for an inner product weights gradient primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, const *memory*::*desc* &src_desc, const *memory*::*desc* &diff_weights_desc, const *memory*::*desc* &diff_bias_desc, const *memory*::*desc* &diff_dst_desc, const *inner_product_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for an inner product weights update primitive with bias.

Note: All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters
- **aengine** – Engine to use.
- **src_desc** – Memory descriptor for src.
- **diff_weights_desc** – Memory descriptor for diff weights.
- **diff_bias_desc** – Memory descriptor for diff bias.
- **diff_dst_desc** – Memory descriptor for diff dst.
- **hint_fwd_pd** – Primitive descriptor for an inner product forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, const *memory*::*desc* &src_desc, const *memory*::*desc* &diff_weights_desc, const *memory*::*desc* &diff_dst_desc, const *inner_product_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for an inner product weights update primitive.

Note: All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- **aengine** – Engine to use.
- **src_desc** – Memory descriptor for src.
- **diff_weights_desc** – Memory descriptor for diff weights.
- **diff_dst_desc** – Memory descriptor for diff dst.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **hint_fwd_pd** – Primitive descriptor for an inner product forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() const

Returns a source memory descriptor.

Returns
Source memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **diff_weights_desc**() const

Returns a diff weights memory descriptor.

Returns
Diff weights memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a diff weights parameter.

memory::*desc* **diff_dst_desc**() const

Returns a diff destination memory descriptor.

Returns
Diff destination memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a diff destination parameter.

memory::*desc* **diff_bias_desc**() const

Returns the diff bias memory descriptor.

Returns
The diff bias memory descriptor.

Returns
A zero memory descriptor of the primitive does not have a diff bias parameter.

prop_kind **get_prop_kind**() const

Returns a propagation kind.

Returns
A propagation kind.

Returns
dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

4.5.9 Layer normalization

The layer normalization primitive performs a forward or backward layer normalization operation on a 2-5D data tensor.

The layer normalization operation performs normalization over the last logical axis of the data tensor and is defined by the following formulas. We show formulas only for 3D data, which are straightforward to generalize to cases of higher dimensions. Variable names follow the standard *Conventions*.

Forward

$$\mathrm{dst}(t,n,c) = \gamma(c) \cdot \frac{\mathrm{src}(t,n,c) - \mu(t,n)}{\sqrt{\sigma^2(t,n) + \varepsilon}} + \beta(c),$$

where

- $\gamma(c), \beta(c)$ are optional scale and shift for a channel (see the `use_scale` and `use_shift` flag),
- $\mu(t,n), \sigma^2(t,n)$ are mean and variance (see `use_global_stats` flag), and
- ε is a constant to improve numerical stability.

Mean and variance are computed at runtime or provided by a user. When mean and variance are computed at runtime, the following formulas are used:

- $\mu(t,n) = \frac{1}{C}\sum_c \mathrm{src}(t,n,c)$,
- $\sigma^2(t,n) = \frac{1}{C}\sum_c (\mathrm{src}(t,n,c) - \mu(t,n))^2$.

The $\gamma(c)$ and $\beta(c)$ tensors are considered learnable.

Difference Between Forward Training and Forward Inference

If mean and variance are computed at runtime (i.e., `use_global_stats` is not set), they become outputs for the propagation kind `forward_training` (because they would be required during the backward propagation). Data layout for mean and variance must be specified during initialization of the layer normalization descriptor by passing the memory descriptor for statistics (e.g., by passing `stat_desc` in `dnnl::layer_normalization_forward::primtive_desc`). Mean and variance are not exposed for the propagation kind `forward_inference`.

Backward

The backward propagation computes $\mathrm{diff\_src}(t,n,c)$, $\mathrm{diff\_}\gamma(c)^*$, and $\mathrm{diff\_}\beta(c)^*$ based on $\mathrm{diff\_dst}(t,n,c)$, $src(t,n,c)$, $\mu(t,n)$, $\sigma^2(t,n)$, $\gamma(c)^*$, and $\beta(c)^*$.

The tensors marked with an asterisk are used only when the primitive is configured to use $\gamma(c)$, and $\beta(c)$ (i.e. `use_scale` and `use_shift` is set).

Execution Arguments

Depending on the flags and propagation kind, the layer normalization primitive requires different inputs and outputs. For clarity, a summary is shown below.

	forward_infer	*forward_train*	*backward*	*backward_data*
none	*In*: src *Out*: dst	*In*: src *Out*: dst, μ, σ^2	*In*: diff_dst, src, μ, σ^2 *Out*: diff_src	Same as for *backward*
use_global_stats	*In*: src, μ, σ^2 *Out*: dst	*In*: src, μ, σ^2 *Out*: dst	*In*: diff_dst, src, μ, σ^2 *Out*: diff_src	Same as for *backward*
use_scale	*In*: src, γ *Out*: dst	*In*: src, γ *Out*: dst, μ, σ^2	*In*: diff_dst, src, μ, σ^2, γ *Out*: diff_src, diff_γ	Not supported
use_shift	*In*: src, β *Out*: dst	*In*: src, β *Out*: dst, μ, σ^2	*In*: diff_dst, src, μ, σ^2, β *Out*: diff_src, diff_β	Not supported
use_scale | *use_shift*	*In*: src, γ, β *Out*: dst	*In*: src, γ, β *Out*: dst, μ, σ^2	*In*: diff_dst, src, μ, σ^2, γ, β *Out*: diff_src, diff_γ, diff_β	Not supported
use_global_stats | *use_scale* | *use_shift*	*In*: src, μ, σ^2, γ, β *Out*: dst	*In*: src, μ, σ^2, γ, β *Out*: dst	*In*: diff_dst, src, μ, σ^2, γ, β *Out*: diff_src, diff_γ, diff_β	Not supported

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	*DNNL_ARG_SRC*
γ, β	*DNNL_ARG_SCALE*
β	*DNNL_ARG_SHIFT*
mean (μ)	*DNNL_ARG_MEAN*
variance (σ)	*DNNL_ARG_VARIANCE*
dst	*DNNL_ARG_DST*
diff_dst	*DNNL_ARG_DIFF_DST*
diff_src	*DNNL_ARG_DIFF_SRC*
diff_γ	*DNNL_ARG_DIFF_SCALE*
diff_β	*DNNL_ARG_DIFF_SHIFT*

Operation Details

1. The different flavors of the primitive are partially controlled by the `flags` parameter that is passed to the primitive descriptor initialization function (e.g., `dnnl::layer_normalization_forward::primtive_desc`). Multiple flags can be combined using the bitwise OR operator (|).

2. For forward propagation, the mean and variance might be either computed at runtime (in which case they are outputs of the primitive) or provided by a user (in which case they are inputs). In the latter case, a user must set the *use_global_stats* flag. For the backward propagation, the mean and variance are always input parameters.

3. Both forward and backward propagation support in-place operations, meaning that src can be used as input and output for forward propagation, and diff_dst can be used as input and output for backward propagation. In case of an in-place operation, the original data will be overwritten. Note, however, that backward propagation requires original src, hence the corresponding forward propagation should not be performed in-place.

Data Types Support

The layer normalization supports the following combinations of data types.

Note: Here we abbreviate data types names for readability. For example, *dnnl::memory::data_type::f32* is abbreviated to *f32*.

Propagation	Source / Destination	Mean / Variance / Scale / Shift
forward / backward	*f32*	*f32*
forward	*f16*	*f32*

Data Representation

Mean and Variance

The mean (μ) and variance (σ^2) are separate tensors with number of dimensions equal to ($data\_ndims - 1$) and size ($data\_dim[0], data\_dim[1], ..., data\_dim[ndims - 2]$).

The corresponding memory object can have an arbitrary memory format. Unless mean and variance are computed at runtime and not exposed (i.e., propagation kind is *forward_inference* and *use_global_stats* is not set), the user should provide a memory descriptor for statistics when initializing the layer normalization descriptor. For best performance, it is advised to use the memory format that follows the data memory format; i.e., if the data format is *tnc*, the best performance can be expected for statistics with the *tn* format and suboptimal for statistics with the *nt* format.

Scale and Shift

If used, the scale (γ) and shift (β) are combined in a single 2D tensor of shape $2 \times C$.

The format of the corresponding memory object must be *nc* (*ab*).

Source, Destination, and Their Gradients

The layer normalization primitive works with an arbitrary data tensor; however, it was designed for RNN data tensors (i.e., *nc*, *tnc*, *ldnc*). Unlike CNN data tensors, RNN data tensors have a single feature dimension. Layer normalization performs normalization over the last logical dimension (feature dimension for RNN tensors) across non-feature dimensions.

The layer normalization primitive is optimized for the following memory formats:

Logical tensor	Implementations optimized for memory formats
NC	*nc* (*ab*)
TNC	*tnc* (*abc*), *ntc* (*bac*)
LDNC	*ldnc* (*abcd*)

API

struct **layer_normalization_forward** : public dnnl::*primitive*

Layer normalization forward propagation primitive.

Public Functions

layer_normalization_forward()

Default constructor. Produces an empty object.

layer_normalization_forward(const *primitive_desc* &pd)

Constructs a layer normalization forward propagation primitive.

Parameters
pd – Primitive descriptor for a layer normalization forward propagation primitive.

struct **primitive_desc** : public dnnl::*primitive_desc*

Primitive descriptor for a layer normalization forward propagation primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, const *memory*::*desc* &src_desc, const *memory*::*desc* &dst_desc, const *memory*::*desc* &stat_desc, float epsilon, *normalization_flags* flags, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a layer normalization forward propagation primitive.

Parameters
- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- **src_desc** – Source memory descriptor.
- **dst_desc** – Destination memory descriptor.
- **stat_desc** – Statistics memory descriptors.
- **epsilon** – Layer normalization epsilon parameter.
- **flags** – Layer normalization flags (*dnnl::normalization_flags*).
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, const *memory*::*desc* &src_desc, const *memory*::*desc* &dst_desc, float epsilon, *normalization_flags* flags, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a layer normalization forward propagation primitive.

Parameters
- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- **src_desc** – Source memory descriptor.

- **dst_desc** – Destination memory descriptor.
- **epsilon** – Layer normalization epsilon parameter.
- **flags** – Layer normalization flags (*dnnl::normalization_flags*).
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() const

Returns a source memory descriptor.

Returns
Source memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **dst_desc**() const

Returns a destination memory descriptor.

Returns
Destination memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a destination parameter.

memory::*desc* **weights_desc**() const

Returns a weights memory descriptor.

Returns
Weights memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a weights parameter.

memory::*desc* **workspace_desc**() const

Returns the workspace memory descriptor.

Returns
Workspace memory descriptor.

Returns
A zero memory descriptor if the primitive does not require workspace parameter.

memory::*desc* **mean_desc**() const

Returns memory descriptor for mean.

Returns
Memory descriptor for mean.

memory::*desc* **variance_desc**() const

Returns memory descriptor for variance.

Returns
Memory descriptor for variance.

dnnl::*prop_kind* **get_prop_kind**() const

Returns a propagation kind.

Returns
A propagation kind.

Returns
dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

float **get_epsilon**() const

Returns an epsilon.

Returns
An epsilon.
Returns
Zero if the primitive does not have an epsilon parameter.

normalization_flags **get_flags**() const

Returns normalization flags.
Returns
Normalization flags.

struct **layer_normalization_backward** : public dnnl::*primitive*

Layer normalization backward propagation primitive.

Public Functions

layer_normalization_backward()

Default constructor. Produces an empty object.

layer_normalization_backward(const *primitive_desc* &pd)

Constructs a layer normalization backward propagation primitive.

Parameters
pd – Primitive descriptor for a layer normalization backward propagation primitive.

struct **primitive_desc** : public dnnl::*primitive_desc*

Primitive descriptor for a layer normalization backward propagation primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, const *memory*::*desc* &diff_src_desc, const *memory*::*desc* &diff_dst_desc, const *memory*::*desc* &src_desc, const *memory*::*desc* &stat_desc, float epsilon, *normalization_flags* flags, const *layer_normalization_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a layer normalization backward propagation primitive.
Parameters
- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::backward_data* and *dnnl::prop_kind::backward* (diffs for all parameters are computed in this case).
- **diff_src_desc** – Diff source memory descriptor.
- **diff_dst_desc** – Diff destination memory descriptor.
- **src_desc** – Source memory descriptor.
- **stat_desc** – Statistics memory descriptors.
- **epsilon** – Layer normalization epsilon parameter.
- **flags** – Layer normalization flags (*dnnl::normalization_flags*).
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **hint_fwd_pd** – Primitive descriptor for a layer normalization forward propagation primitive. It is used as a hint for deciding which memory format to use.

- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, const *memory*::*desc* &diff_src_desc, const *memory*::*desc* &diff_dst_desc, const *memory*::*desc* &src_desc, float epsilon, *normalization_flags* flags, const *layer_normalization_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a layer normalization backward propagation primitive.

Parameters

- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::backward_data* and *dnnl::prop_kind::backward* (diffs for all parameters are computed in this case).
- **diff_src_desc** – Diff source memory descriptor.
- **diff_dst_desc** – Diff destination memory descriptor.
- **src_desc** – Source memory descriptor.
- **epsilon** – Layer normalization epsilon parameter.
- **flags** – Layer normalization flags (*dnnl::normalization_flags*).
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **hint_fwd_pd** – Primitive descriptor for a layer normalization forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() const

Returns a source memory descriptor.

Returns

Source memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **weights_desc**() const

Returns a weights memory descriptor.

Returns

Weights memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a weights parameter.

memory::*desc* **dst_desc**() const

Returns a destination memory descriptor.

Returns

Destination memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a destination parameter.

memory::*desc* **diff_src_desc**() const

Returns a diff source memory descriptor.

Returns

Diff source memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a diff source memory with.

memory::*desc* **diff_dst_desc**() const

Returns a diff destination memory descriptor.

Returns
Diff destination memory descriptor.
Returns
A zero memory descriptor if the primitive does not have a diff destination parameter.

memory::*desc* **diff_weights_desc**() const

Returns a diff weights memory descriptor.
Returns
Diff weights memory descriptor.
Returns
A zero memory descriptor if the primitive does not have a diff weights parameter.

memory::*desc* **mean_desc**() const

Returns memory descriptor for mean.
Returns
Memory descriptor for mean.

memory::*desc* **variance_desc**() const

Returns memory descriptor for variance.
Returns
Memory descriptor for variance.

memory::*desc* **workspace_desc**() const

Returns the workspace memory descriptor.
Returns
Workspace memory descriptor.
Returns
A zero memory descriptor if the primitive does not require workspace parameter.

dnnl::*prop_kind* **get_prop_kind**() const

Returns a propagation kind.
Returns
A propagation kind.
Returns
dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

float **get_epsilon**() const

Returns an epsilon.
Returns
An epsilon.
Returns
Zero if the primitive does not have an epsilon parameter.

normalization_flags **get_flags**() const

Returns normalization flags.
Returns
Normalization flags.

4.5.10 Local Response Normalization

The LRN primitive performs a forward or backward local response normalization operation defined by the following formulas. Variable names follow the standard *Conventions*.

Forward

LRN with algorithm *lrn_across_channels*:

$$\operatorname{dst}(n,c,h,w) = \left\{ k + \frac{\alpha}{n_l} \sum_{i=-(n_l-1)/2}^{(n_l+1)/2-1} (\operatorname{src}(n,c+i,h,w))^2 \right\}^{-\beta} \cdot \operatorname{src}(n,c,h,w),$$

LRN with algorithm *lrn_within_channel*:

$$\operatorname{dst}(n,c,h,w) = \left\{ k + \frac{\alpha}{n_l} \sum_{i=-(n_l-1)/2}^{(n_l+1)/2-1} \sum_{j=-(n_l-1)/2}^{(n_l+1)/2-1} (\operatorname{src}(n,c,h+i,w+j))^2 \right\}^{-\beta} \cdot \operatorname{src}(n,c,h,w),$$

where n_l is the `local_size`. Formulas are provided for 2D spatial data case.

Backward

The backward propagation computes $\operatorname{diff\_src}(n,c,h,w)$, based on $\operatorname{diff\_dst}(n,c,h,w)$ and $\operatorname{src}(n,c,h,w)$.

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	*DNNL_ARG_SRC*
dst	*DNNL_ARG_DST*
workspace	*DNNL_ARG_WORKSPACE*
diff_src	*DNNL_ARG_DIFF_SRC*
diff_dst	*DNNL_ARG_DIFF_DST*

Operation Details

1. During training, LRN might or might not require a workspace on forward and backward passes. The behavior is implementation specific. Optimized implementations typically require a workspace and use it to save some intermediate results from the forward pass that accelerate computations on the backward pass. To check whether a workspace is required, query the LRN primitive descriptor for the workspace. Success indicates that the workspace is required and its description will be returned.

Data Type Support

The LRN primitive supports the following combinations of data types.

Note: Here we abbreviate data types names for readability. For example, *dnnl::memory::data_type::f32* is abbreviated to *f32*.

Propagation	Source / Destination
forward / backward	*f32*, *bf16*
forward	*f16*

Data Representation

Source, Destination, and Their Gradients

Like most other primitives, the LRN primitive expects the following tensors:

Spatial	Source / Destination
0D	$N \times C$
1D	$N \times C \times W$
2D	$N \times C \times H \times W$
3D	$N \times C \times D \times H \times W$

The LRN primitive is optimized for the following memory formats:

Spatial	Logical tensor	Implementations optimized for memory formats
2D	NCHW	*nchw* (*abcd*), *nhwc* (*acdb*), *optimized*

Here *optimized* means the format chosen by the preceding compute-intensive primitive.

Post-ops and Attributes

The LRN primitive does not support any post-ops or attributes.

API

struct **lrn_forward** : public dnnl::*primitive*

Local response normalization (LRN) forward propagation primitive.

Public Functions

lrn_forward()

Default constructor. Produces an empty object.

lrn_forward(const *primitive_desc* &pd)

Constructs an LRN forward propagation primitive.

Parameters
pd – Primitive descriptor for an LRN forward propagation primitive.

struct **primitive_desc** : public dnnl::*primitive_desc*

Primitive descriptor for an LRN forward propagation primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *algorithm* aalgorithm, const *memory*::*desc* &src_desc, const *memory*::*desc* &dst_desc, *memory*::*dim* local_size, float alpha, float beta, float k, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for an LRN forward propagation primitive.

Parameters
- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- **aalgorithm** – LRN algorithm kind: either *dnnl::algorithm::lrn_across_channels*, or *dnnl::algorithm::lrn_within_channel*.
- **src_desc** – Source memory descriptor.
- **dst_desc** – Destination memory descriptor.
- **local_size** – Regularization local size.
- **alpha** – The alpha regularization parameter.
- **beta** – The beta regularization parameter.
- **k** – The k regularization parameter.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() const

Returns a source memory descriptor.

Returns
Source memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **dst_desc**() const

Returns a destination memory descriptor.

Returns
Destination memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a destination parameter.

memory::*desc* **workspace_desc**() const

Returns the workspace memory descriptor.

Returns

Workspace memory descriptor.

Returns

A zero memory descriptor if the primitive does not require workspace parameter.

algorithm **get_algorithm**() const

Returns an algorithm kind.

Returns

An algorithm kind.

Returns

dnnl::algorithm::undef if the primitive does not have an algorithm parameter.

prop_kind **get_prop_kind**() const

Returns a propagation kind.

Returns

A propagation kind.

Returns

dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

float **get_alpha**() const

Returns an alpha.

Returns

An alpha.

Returns

Zero if the primitive does not have an alpha parameter.

float **get_beta**() const

Returns a beta.

Returns

A beta.

Returns

Zero if the primitive does not have a beta parameter.

memory::*dim* **get_local_size**() const

Returns an LRN local size parameter.

Returns

An LRN local size parameter.

Returns

Zero if the primitive does not have an LRN local size parameter.

float **get_k**() const

Returns an LRN K parameter.

Returns

An LRN K parameter.

Returns

Zero if the primitive does not have an LRN K parameter.

struct **lrn_backward** : public dnnl::*primitive*

Local response normalization (LRN) backward propagation primitive.

Public Functions

lrn_backward()

Default constructor. Produces an empty object.

lrn_backward(const *primitive_desc* &pd)

Constructs an LRN backward propagation primitive.

Parameters
pd – Primitive descriptor for an LRN backward propagation primitive.

struct **primitive_desc** : public dnnl::*primitive_desc*

Primitive descriptor for an LRN backward propagation primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *algorithm* aalgorithm, const *memory*::*desc* &diff_src_desc, const *memory*::*desc* &diff_dst_desc, const *memory*::*desc* &src_desc, *memory*::*dim* local_size, float alpha, float beta, float k, const *lrn_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for an LRN backward propagation primitive.

Parameters
- **aengine** – Engine to use.
- **aalgorithm** – LRN algorithm kind: either *dnnl::algorithm::lrn_across_channels*, or *dnnl::algorithm::lrn_within_channel*.
- **diff_src_desc** – Diff source memory descriptor.
- **diff_dst_desc** – Diff destination memory descriptor.
- **src_desc** – Source memory descriptor.
- **local_size** – Regularization local size.
- **alpha** – The alpha regularization parameter.
- **beta** – The beta regularization parameter.
- **k** – The k regularization parameter.
- **hint_fwd_pd** – Primitive descriptor for an LRN forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **diff_src_desc**() const

Returns a source memory descriptor.

Returns
Source memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **diff_dst_desc**() const

Returns a diff destination memory descriptor.

Returns
Diff destination memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a diff destination parameter.

memory::*desc* **workspace_desc**() const

Returns the workspace memory descriptor.

Returns
Workspace memory descriptor.

Returns
A zero memory descriptor if the primitive does not require workspace parameter.

algorithm **get_algorithm**() const

Returns an algorithm kind.

Returns
An algorithm kind.

Returns
dnnl::algorithm::undef if the primitive does not have an algorithm parameter.

prop_kind **get_prop_kind**() const

Returns a propagation kind.

Returns
A propagation kind.

Returns
dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

float **get_alpha**() const

Returns an alpha.

Returns
An alpha.

Returns
Zero if the primitive does not have an alpha parameter.

float **get_beta**() const

Returns a beta.

Returns
A beta.

Returns
Zero if the primitive does not have a beta parameter.

memory::*dim* **get_local_size**() const

Returns an LRN local size parameter.

Returns
An LRN local size parameter.

Returns
Zero if the primitive does not have an LRN local size parameter.

float **get_k**() const

Returns an LRN K parameter.

Returns
An LRN K parameter.

Returns
Zero if the primitive does not have an LRN K parameter.

4.5.11 Matrix Multiplication

The matrix multiplication (MatMul) primitive computes the product of two 2D tensors with optional bias addition. Variable names follow the standard *Conventions*.

$$\mathrm{dst}(m,n) = \sum_{k=0}^{K-1} (\mathrm{src}(m,k) \cdot \mathrm{weights}(k,n)) + \mathrm{bias}(m,n)$$

The MatMul primitive also supports batching multiple independent matrix multiplication operations, in which case the tensors must be 3D:

$$\mathrm{dst}(mb,m,n) = \sum_{k=0}^{K-1} (\mathrm{src}(mb,m,k) \cdot \mathrm{weights}(mb,k,n)) + \mathrm{bias}(mb,m,n)$$

The bias tensor is optional and supports implicit broadcast semantics: any of its dimensions can be 1 and the same value would be used across the corresponding dimension. However, bias must have the same number of dimensions as the dst.

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	*DNNL_ARG_SRC*
weights	*DNNL_ARG_WEIGHTS*
bias	*DNNL_ARG_BIAS*
dst	*DNNL_ARG_DST*

Operation Details

The MatMul primitive supports input and output tensors with run-time specified shapes and memory formats. The run-time specified dimensions or strides are specified using the *DNNL_RUNTIME_DIM_VAL* wildcard value during the primitive initialization and creation stage. At the execution stage, the user must pass fully specified memory objects so that the primitive is able to perform the computations. Note that the less information about shapes or format is available at the creation stage, the less performant execution will be. In particular, if the shape is not known at creation stage, one cannot use the special format tag *any* to enable an implementation to choose the most appropriate memory format for the corresponding input or output shapes. On the other hand, run-time specified shapes enable users to create a primitive once and use it in different situations.

Data Types Support

The MatMul primitive supports the following combinations of data types for source, destination, weights, and bias tensors.

Note: Here we abbreviate data types names for readability. For example, *dnnl::memory::data_type::f32* is abbreviated to *f32*.

Source	Weights	Destination	Bias
f32	*f32*	*f32*	*f32*
f16	*f16*	*f16*	*f16*
bf16	*bf16*	*bf16*	*bf16*, *f32*
u8, *s8*	*s8*, *u8*	*u8*, *s8*, *s32*, *f32*	*u8*, *s8*, *s32*, *f32*

Data Representation

The MatMul primitive expects the following tensors:

Dims	Source	Weights	Destination	Bias (optional)
2D	$M \times K$	$K \times N$	$M \times N$	(M or 1) $\times$ (N or 1)
3D	$MB \times M \times K$	$MB \times K \times N$	$MB \times M \times N$	(MB or 1) $\times$ (M or 1) $\times$ (N or 1)

The MatMul primitive is generally optimized for the case in which memory objects use plain memory formats (with some restrictions; see the table below). However, it is recommended to use the placeholder memory format *any* if an input tensor is reused across multiple executions. In this case, the primitive will set the most appropriate memory format for the corresponding input tensor.

The table below shows the combinations of memory formats for which the MatMul primitive is optimized. The memory format of the destination tensor should always be *ab* for the 2D case and *abc* for the 3D one.

Dims	Logical tensors	MatMul is optimized for the following memory formats
2D	Source: $M \times K$, Weights: $K \times N$	Source: *ab* or *ba*, Weights: *ab* or *ba*
3D	Source: $MB \times M \times K$, Weights: $MB \times K \times N$	Source: *abc* or *acb*, Weights: *abc* or *acb*

Attributes and Post-ops

Attributes and post-ops enable modifying the behavior of the MatMul primitive. The following attributes and post-ops are supported:

Type	Operation	Description	Restrictions
Attribute	*Scales*	Sets scale(s) for the corresponding tensor(s)	
Attribute	*Zero points*	Sets zero point(s) for the corresponding tensors	Int8 computations only
Post-op	*Eltwise* | Applies an elementwise operation to the result		
Post-op	*Binary* | Applies a binary operation to the result		
Post-op	*Sum*	Adds the operation result to the destination tensor instead of overwriting it	

The primitive supports dynamic quantization via run-time scales. That means a user could configure the scales and zero-point attributes at the primitive descriptor creation stage. The user must then provide the scales and zero-points

as an additional input memory objects with argument *DNNL_ARG_ATTR_SCALES* and *DNNL_ARG_ATTR_ZERO_POINTS* during the execution stage (more details are provided in the *Quantization* section).

API

struct **matmul** : public dnnl::*primitive*

Matrix multiplication (matmul) primitive.

Public Functions

matmul()

Default constructor. Produces an empty object.

matmul(const *primitive_desc* &pd)

Constructs a matmul primitive.

Parameters
pd – Primitive descriptor for a matmul primitive.

struct **primitive_desc** : public dnnl::*primitive_desc*

Primitive descriptor for a matmul primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, const *memory*::*desc* &src_desc, const *memory*::*desc* &weights_desc, const *memory*::*desc* &dst_desc, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a matmul primitive without bias.

Parameters
- **aengine** – Engine to use.
- **src_desc** – Memory descriptor for source (matrix A).
- **weights_desc** – Memory descriptor for weights (matrix B).
- **dst_desc** – Memory descriptor for destination (matrix C).
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, const *memory*::*desc* &src_desc, const *memory*::*desc* &weights_desc, const *memory*::*desc* &bias_desc, const *memory*::*desc* &dst_desc, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a matmul primitive with bias.

Parameters
- **aengine** – Engine to use.
- **src_desc** – Memory descriptor for source (matrix A).
- **weights_desc** – Memory descriptor for weights (matrix B).
- **dst_desc** – Memory descriptor for destination (matrix C).
- **bias_desc** – Memory descriptor for bias.

- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() const

Returns a source memory descriptor.

Returns

Source memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **weights_desc**() const

Returns a weights memory descriptor.

Returns

Weights memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a weights parameter.

memory::*desc* **bias_desc**() const

Returns the bias memory descriptor.

Returns

The bias memory descriptor.

Returns

A zero memory descriptor of the primitive does not have a bias parameter.

memory::*desc* **dst_desc**() const

Returns a destination memory descriptor.

Returns

Destination memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a destination parameter.

4.5.12 Pooling

The pooling primitive performs forward or backward max or average pooling operation on 1D, 2D, or 3D spatial data.

The pooling operation is defined by the following formulas. We show formulas only for 2D spatial data which are straightforward to generalize to cases of higher and lower dimensions. Variable names follow the standard *Conventions*.

Forward

Max pooling:

$$\mathrm{dst}(n, c, oh, ow) = \max_{kh,kw} \left(\mathrm{src}(n, c, oh \cdot SH + kh - PH_L, ow \cdot SW + kw - PW_L)\right)$$

Average pooling:

$$\mathrm{dst}(n, c, oh, ow) = \frac{1}{DENOM} \sum_{kh,kw} \mathrm{src}(n, c, oh \cdot SH + kh - PH_L, ow \cdot SW + kw - PW_L)$$

Here output spatial dimensions are calculated similarly to how they are done for *Convolution and Deconvolution*.

Average pooling supports two algorithms:

- *pooling_avg_include_padding*, in which case $DENOM = KH \cdot KW$,

- *pooling_avg_exclude_padding*, in which case $DENOM$ equals to the size of overlap between an averaging window and images.

Difference Between Forward Training and Forward Inference

Max pooling requires a `workspace` for the *forward_training* propagation kind, and does not require it for *forward_inference* (see details below).

Backward

The backward propagation computes $\text{diff\_src}(n, c, h, w)$, based on $\text{diff\_dst}(n, c, h, w)$ and, in case of max pooling, `workspace`.

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	*DNNL_ARG_SRC*
dst	*DNNL_ARG_DST*
workspace	*DNNL_ARG_WORKSPACE*
diff_src	*DNNL_ARG_DIFF_SRC*
diff_dst	*DNNL_ARG_DIFF_DST*

Operation Details

1. During training, max pooling requires a workspace on forward (*forward_training*) and backward passes to save indices where a maximum was found. The workspace format is opaque, and the indices cannot be restored from it. However, one can use backward pooling to perform up-sampling (used in some detection topologies). The workspace can be created via *dnnl::pooling_forward::primitive_desc::workspace_desc()*.
2. A user can use memory format tag *any* for `dst` memory descriptor when creating pooling forward propagation. The library would derive the appropriate format from the `src` memory descriptor. However, the `src` itself must be defined. Similarly, a user can use memory format tag *any* for the `diff_src` memory descriptor when creating pooling backward propagation.

Data Type Support

The pooling primitive supports the following combinations of data types.

Note: Here we abbreviate data types names for readability. For example, *dnnl::memory::data_type::f32* is abbreviated to *f32*.

Propagation	Source / Destination	Accumulation data type (used for average pooling only)
forward / backward	*f32*, *bf16*	*f32*
forward	*f16*	*f16*
forward	*s8*, *u8*, *s32*	*s32*

Data Representation

Source, Destination, and Their Gradients

Like other CNN primitives, the pooling primitive expects data to be an $N \times C \times W$ tensor for the 1D spatial case, an $N \times C \times H \times W$ tensor for the 2D spatial case, and an $N \times C \times D \times H \times W$ tensor for the 3D spatial case.

The pooling primitive is optimized for the following memory formats:

Spatial	Logical tensor	Data type	Implementations optimized for memory formats
1D	NCW	f32	*ncw* (*abc*), *nwc* (*acb*), *optimized^*
1D	NCW	s32, s8, u8	*nwc* (*acb*), *optimized^*
2D	NCHW	f32	*nchw* (*abcd*), *nhwc* (*acdb*), *optimized^*
2D	NCHW	s32, s8, u8	*nhwc* (*acdb*), *optimized^*
3D	NCDHW	f32	*ncdhw* (*abcde*), *ndhwc* (*acdeb*), *optimized^*
3D	NCDHW	s32, s8, u8	*ndhwc* (*acdeb*), *optimized^*

Here *optimized^* means the format that comes out of any preceding compute-intensive primitive.

Post-ops and Attributes

The pooling primitive does not support any post-ops or attributes.

API

struct **pooling_forward** : public dnnl::*primitive*

Pooling forward propagation primitive.

Public Functions

pooling_forward()

Default constructor. Produces an empty object.

pooling_forward(const *primitive_desc* &pd)

Constructs a pooling forward propagation primitive.

Parameters

pd – Primitive descriptor for a pooling forward propagation primitive.

struct **primitive_desc** : public dnnl::*primitive_desc*

Primitive descriptor for a pooling forward propagation primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *algorithm* aalgorithm, const *memory*::*desc* &src_desc, const *memory*::*desc* &dst_desc, const *memory*::*dims* &strides, const *memory*::*dims* &kernel, const *memory*::*dims* &dilation, const *memory*::*dims* &padding_l, const *memory*::*dims* &padding_r, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for pooling forward propagation primitive.

Arrays `strides`, `kernel`, `dilation`, `padding_l` and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Parameters

- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- **aalgorithm** – Pooling algorithm kind: either *dnnl::algorithm::pooling_max*, *dnnl::algorithm::pooling_avg_include_padding*, or *dnnl::algorithm::pooling_avg_exclude_padding*.
- **src_desc** – Source memory descriptor.
- **dst_desc** – Destination memory descriptor.
- **strides** – Vector of strides for spatial dimension.
- **kernel** – Vector of kernel spatial dimensions.
- **dilation** – Array of dilations for spatial dimension.
- **padding_l** – Vector of padding values for low indices for each spatial dimension `([[front,] top,] left)`.
- **padding_r** – Vector of padding values for high indices for each spatial dimension `([[back,] bottom,] right)`.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() const

Returns a source memory descriptor.

Returns

Source memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **dst_desc**() const

Returns a destination memory descriptor.

Returns

Destination memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a destination parameter.

memory::*desc* **workspace_desc**() const

Returns the workspace memory descriptor.

Returns

Workspace memory descriptor.

Returns
A zero memory descriptor if the primitive does not require workspace parameter.

algorithm **get_algorithm**() const

Returns an algorithm kind.

Returns
An algorithm kind.

Returns
dnnl::algorithm::undef if the primitive does not have an algorithm parameter.

prop_kind **get_prop_kind**() const

Returns a propagation kind.

Returns
A propagation kind.

Returns
dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

memory::*dims* **get_strides**() const

Returns strides.

Returns
Strides.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a strides parameter.

memory::*dims* **get_kernel**() const

Returns a pooling kernel parameter.

Returns
A pooling kernel parameter.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a pooling kernel parameter.

memory::*dims* **get_dilations**() const

Returns dilations.

Returns
Dilations.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a dilations parameter.

memory::*dims* **get_padding_l**() const

Returns a left padding.

Returns
A left padding.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a left padding parameter.

memory::*dims* **get_padding_r**() const

Returns a right padding.

Returns
A right padding.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a right padding parameter.

struct **pooling_backward** : public dnnl::*primitive*

Pooling backward propagation primitive.

Public Functions

pooling_backward()

Default constructor. Produces an empty object.

pooling_backward(const *primitive_desc* &pd)

Constructs a pooling backward propagation primitive.

Parameters

pd – Primitive descriptor for a pooling backward propagation primitive.

struct **primitive_desc** : public dnnl::*primitive_desc*

Primitive descriptor for a pooling backward propagation primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *algorithm* aalgorithm, const *memory*::*desc* &diff_src_desc, const *memory*::*desc* &diff_dst_desc, const *memory*::*dims* &strides, const *memory*::*dims* &kernel, const *memory*::*dims* &dilation, const *memory*::*dims* &padding_l, const *memory*::*dims* &padding_r, const *pooling_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a pooling backward propagation primitive.

Arrays `strides`, `kernel`, `dilation`, `padding_l` and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Parameters

- **aengine** – Engine to use.
- **aalgorithm** – Pooling algorithm kind: either *dnnl::algorithm::pooling_max*, *dnnl::algorithm::pooling_avg_include_padding*, or *dnnl::algorithm::pooling_avg_exclude_padding*.
- **diff_src_desc** – Diff source memory descriptor.
- **diff_dst_desc** – Diff destination memory descriptor.
- **strides** – Vector of strides for spatial dimension.
- **kernel** – Vector of kernel spatial dimensions.
- **dilation** – Array of dilations for spatial dimension.
- **padding_l** – Vector of padding values for low indices for each spatial dimension `([[front,] top,] left)`.
- **padding_r** – Vector of padding values for high indices for each spatial dimension `([[back,] bottom,] right)`.
- **hint_fwd_pd** – Primitive descriptor for a pooling forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **diff_src_desc**() const

Returns a source memory descriptor.

Returns

Source memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **diff_dst_desc**() const

Returns a diff destination memory descriptor.

Returns
Diff destination memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a diff destination parameter.

memory::*desc* **workspace_desc**() const

Returns the workspace memory descriptor.

Returns
Workspace memory descriptor.

Returns
A zero memory descriptor if the primitive does not require workspace parameter.

algorithm **get_algorithm**() const

Returns an algorithm kind.

Returns
An algorithm kind.

Returns
dnnl::algorithm::undef if the primitive does not have an algorithm parameter.

prop_kind **get_prop_kind**() const

Returns a propagation kind.

Returns
A propagation kind.

Returns
dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

memory::*dims* **get_strides**() const

Returns strides.

Returns
Strides.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a strides parameter.

memory::*dims* **get_kernel**() const

Returns a pooling kernel parameter.

Returns
A pooling kernel parameter.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a pooling kernel parameter.

memory::*dims* **get_dilations**() const

Returns dilations.

Returns
Dilations.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a dilations parameter.

memory::*dims* **get_padding_l**() const

Returns a left padding.

Returns
A left padding.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a left padding parameter.

memory::*dims* **get_padding_r**() const

Returns a right padding.

Returns
A right padding.

Returns
An empty *dnnl::memory::dims* if the primitive does not have a right padding parameter.

4.5.13 Prelu

The prelu primitive (Leaky ReLU with trainable alpha parameter) performs forward or backward operation on data tensor. Weights (alpha) tensor supports broadcast-semantics.

Broadcast configuration is assumed based on src and weights dimensions.

Forward

The prelu operation is defined by the following formulas. We show formulas only for 2D spatial data which are straightforward to generalize to cases of higher and lower dimensions. Variable names follow the standard *Conventions*.

$$\text{dst}(n,c,h,w) = \begin{cases} \text{src}(n,c,h,w) & \text{if } \text{src}(n,c,h,w) > 0 \\ \text{src}(n,c,h,w) \cdot \text{weights}(n,c,h,w) & \text{if } \text{src}(n,c,h,w) \leq 0 \end{cases}$$

Difference Between Forward Training and Forward Inference

There is no difference between the *forward_training* and *forward_inference* propagation kinds.

Backward

The backward propagation computes diff_src and diff_weights. For no broadcast case, results are calculated using formula:

$$\text{diff\_src}(n,c,h,w) = \begin{cases} \text{diff\_dst}(n,c,h,w) & if\ \text{src}(n,c,h,w) > 0 \\ \text{diff\_dst}(n,c,h,w) \cdot \text{weights}(n,c,h,w) & if\ \text{src}(n,c,h,w) \leq 0 \end{cases}$$

$$\text{diff\_weights}(n,c,h,w) = \min(\text{src}(n,c,h,w), 0) \cdot \text{diff\_dst}(n,c,h,w)$$

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	*DNNL_ARG_SRC*
dst	*DNNL_ARG_DST*
weights	*DNNL_ARG_WEIGHTS*
diff_src	*DNNL_ARG_DIFF_SRC*
diff_dst	*DNNL_ARG_DIFF_DST*
diff_weights	*DNNL_ARG_DIFF_WEIGHTS*

Operation Details

1. All input and output tensors must have the same number of dimensions.
2. weights tensor dimensions must follow broadcast semantics. Each dimension can either be equal to the corresponding data dimension or equal to 1 to indicate a broadcasted dimension.

Post-ops and Attributes

The prelu primitive does not have to support any post-ops or attributes.

Data Types Support

The PReLU primitive supports the following combinations of data types:

Note: Here we abbreviate data types names for readability. For example, *dnnl::memory::data_type::f32* is abbreviated to *f32*.

Propagation	Source / Destination
forward / backward	*f32*, *s32*, *bf16*, *f16*, *s8*, *u8*

Data Representation

Source, Destination, and Their Gradients

The PReLU primitive works with arbitrary data tensors. There is no special meaning associated with any logical dimensions.

API

struct **prelu_forward** : public dnnl::*primitive*

PReLU forward propagation primitive.

Public Functions

prelu_forward() = default

Default constructor. Produces an empty object.

prelu_forward(const *primitive_desc* &pd)

Constructs a prelu forward propagation primitive.

Parameters
pd – Primitive descriptor for a prelu forward propagation primitive.

struct **primitive_desc** : public dnnl::*primitive_desc*

Primitive descriptor for a PReLU forward propagation primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, const *memory*::*desc* &src_desc, const *memory*::*desc* &weight_desc, const *memory*::*desc* &dst_desc, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a PReLU forward propagation primitive.

Parameters
- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- **src_desc** – Source memory descriptor.
- **weight_desc** – Alpha parameters memory descriptor.
- **dst_desc** – Destination memory descriptor.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() const

Returns a source memory descriptor.

Returns
Source memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **dst_desc**() const

Returns a destination memory descriptor.

Returns
Destination memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a destination parameter.

prop_kind **get_prop_kind**() const

Returns a propagation kind.

Returns
A propagation kind.

Returns
dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

struct **prelu_backward** : public dnnl::*primitive*

PReLU backward propagation primitive.

Public Functions

prelu_backward() = default

Default constructor. Produces an empty object.

prelu_backward(const *primitive_desc* &pd)

Constructs a prelu backward propagation primitive.

Parameters
pd – Primitive descriptor for a prelu backward propagation primitive.

struct **primitive_desc** : public dnnl::*primitive_desc*

Primitive descriptor for prelu backward propagation.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, const *memory*::*desc* &src_desc, const *memory*::*desc* &weight_desc, const *memory*::*desc* &diff_src_desc, const *memory*::*desc* &diff_weights_desc, const *memory*::*desc* &diff_dst_desc, const *prelu_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a descriptor for a PReLU backward propagation primitive.

Parameters
- **aengine** – Engine to use.
- **src_desc** – Source memory descriptor.
- **weight_desc** – Alpha parameters memory descriptor.
- **diff_src_desc** – Diff source memory descriptor.
- **diff_weights_desc** – Diff alpha parameters memory descriptor.
- **diff_dst_desc** – Diff destination memory descriptor.
- **hint_fwd_pd** – Primitive descriptor for a PReLU forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() const

Returns a source memory descriptor.

Returns
Source memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **diff_src_desc**() const

Returns a diff source memory descriptor.

Returns

Diff source memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a diff source memory with.

memory::*desc* **diff_dst_desc**() const

Returns a diff destination memory descriptor.

Returns

Diff destination memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a diff destination parameter.

prop_kind **get_prop_kind**() const

Returns a propagation kind.

Returns

A propagation kind.

Returns

dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

4.5.14 Reduction

The reduction primitive performs a reduction operation on one or multiple arbitrary dimensions, with respect to a specified algorithm. Variable names follow the standard *Conventions*.

$$\mathrm{dst}(f) = \underset{r \in R}{reduce\_op}\, \mathrm{src}(r),$$

where $reduce\_op$ can be max, min, sum, mul, mean, Lp-norm and Lp-norm-power-p, f is an index in an idle dimension and r is an index in a reduction dimension R.

The reduction algorithms are specified as follow.

Mean:

$$\mathrm{dst}(f) = \frac{\sum\limits_{r \in R} \mathrm{src}(r)}{\|R\|},$$

where $|R|$ is the size of a reduction dimension.

Lp-norm:

$$\mathrm{dst}(f) = \sqrt[p]{eps\_op(\sum_{r \in R} |src(r)|^p, eps)},$$

where eps_op can be max and sum.

Lp-norm-power-p:

$$\mathrm{dst}(f) = eps\_op(\sum_{r} |src(r)|^p, eps),$$

where eps_op can be max and sum.

Note:

- The reduction primitive requires the source and destination tensors to have the same number of dimensions.
- Dimensions which are reduced are of size 1 in the destination tensor.
- The reduction primitive does not have a notion of forward or backward propagations.

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	*DNNL_ARG_SRC*
dst	*DNNL_ARG_DST*

Operation Details

The dst memory format can be either specified explicitly or by using the special format tag *any* (recommended), in which case the primitive will derive the most appropriate memory format based on the format of the source tensor.

Data Types Support

The reduction primitive supports the following combinations of data types:

Propagation	Source / Destination
forward / backward	*f32*, *s32*, *bf16*, *f16*, *s8*, *u8*

Data Representation

The reduction primitive works with arbitrary data tensors. There is no special meaning associated with any of the dimensions of a tensor.

Attributes and Post-ops

Type	Operation	Description	Restrictions
Attribute	*Scales*	Sets scale(s) for the corresponding tensor(s)	Int8 computations only
Attribute	*Zero points*	Sets zero point(s) for the corresponding tensors	Int8 computations only
post-op	*Eltwise*	Applies an elementwise operation to the result	
post-op	*Binary*	Applies a binary operation to the result	

API

struct **reduction** : public dnnl::*primitive*

Reduction.

Public Functions

reduction() = default

Default constructor. Produces an empty object.

struct **primitive_desc** : public dnnl::*primitive_desc*

Primitive descriptor for a reduction primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *algorithm* aalgorithm, const *memory*::*desc* &src_desc, const *memory*::*desc* &dst_desc, float p, float eps, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a reduction primitive using algorithm specific parameters, source and destination memory descriptors.

Note: Destination memory descriptor may be initialized with *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- **aengine** – Engine to use.
- **aalgorithm** – reduction algorithm kind. Possible values: *algorithm::reduction_max*, *algorithm::reduction_min*, *algorithm::reduction_sum*, *algorithm::reduction_mul*, *algorithm::reduction_mean*, *algorithm::reduction_norm_lp_max*, *algorithm::reduction_norm_lp_sum*, *algorithm::reduction_norm_lp_power_p_max*, *algorithm::reduction_norm_lp_power_p_sum*.
- **p** – algorithm specific parameter.
- **eps** – algorithm specific parameter.
- **src_desc** – Source memory descriptor.
- **dst_desc** – Destination memory descriptor.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() const

Returns a source memory descriptor.

Returns

Source memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **dst_desc**() const

Returns a destination memory descriptor.

Returns
Destination memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a destination parameter.

float **get_p**() const

Returns a reduction P parameter.

Returns
A reduction P parameter.

Returns
Zero if the primitive does not have a reduction P parameter.

float **get_epsilon**() const

Returns an epsilon.

Returns
An epsilon.

Returns
Zero if the primitive does not have an epsilon parameter.

algorithm **get_algorithm**() const

Returns an algorithm kind.

Returns
An algorithm kind.

Returns
dnnl::algorithm::undef if the primitive does not have an algorithm parameter.

4.5.15 Reorder

A primitive to copy data between two memory objects. This primitive is typically used to change the way that the data is laid out in memory.

The reorder primitive copies data between different memory formats but does not change the tensor from mathematical perspective. Variable names follow the standard *Conventions*.

$$\mathrm{dst}(\overline{x}) = \mathrm{src}(\overline{x})$$

for $\overline{x} = (x_0, \ldots, x_n)$.

As described in *Introduction* in order to achieve the best performance some primitives (such as convolution) require special memory format which is typically referred to as an *optimized* memory format. The *optimized* memory format may match or may not match memory format that data is currently kept in. In this case a user can use reorder primitive to copy (reorder) the data between the memory formats.

Using the attributes and post-ops users can also use reorder primitive to quantize the data (and if necessary change the memory format simultaneously).

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	*DNNL_ARG_FROM*
dst	*DNNL_ARG_TO*

Operation Details

1. The reorder primitive requires the source and destination tensors to have the same shape. Implicit broadcasting is not supported.
2. While in most of the cases the reorder should be able to handle arbitrary source and destination memory formats and data types, it might happen than some combinations are not implemented. For instance:
   - Reorder implementations between weights in non-plain memory formats might be limited (but if encountered in real practice should be treated as a bug and reported to oneDNN team);
   - Weights in one Winograd format cannot be reordered to the weights of the other Winograd format;
   - Quantized weights for convolution with #dnnl_s8 source data type cannot be dequantized back to the #dnnl_f32 data type;
3. To alleviate the problem a user may rely on fact that the reorder from original plain memory format and user's data type to the *optimized* format with chosen data type should be always implemented.

Data Types Support

The reorder primitive supports arbitrary data types for the source and destination.

When converting the data from one data type to a smaller one saturation is used. For instance:

```
reorder(src={1024, data_type=f32}, dst={, data_type=s8})
// dst == {127}

reorder(src={-124, data_type=f32}, dst={, data_type=u8})
// dst == {0}
```

Data Representation

The reorder primitive works with arbitrary data tensors. There is no special meaning associated with any logical dimensions.

Post-ops and Attributes

The reorder primitive should support the following attributes and post-ops:

Type	Operation	Description	Restrictions
Attribute	*Scales*	Sets scale(s) for the corresponding tensor(s)	Int8 computations only
Attribute	*Zero points*	Sets zero point(s) for the corresponding tensors	Int8 computations only
post-op	*Sum*	Adds the operation result to the destination tensor instead of overwriting it	

For instance, the following pseudo-code

```
reorder(
        src = {dims={N, C, H, W}, data_type=dt_src, memory_format=fmt_src},
        dst = {dims={N, C, H, W}, data_type=dt_dst, memory_format=fmt_dst},
        attr ={
            output_scale=alpha,
            post-ops = { sum={scale=beta} },
        })
```

would lead to the following operation:

$$\mathrm{dst}(\overline{x}) = \alpha \cdot \mathrm{src}(\overline{x}) + \beta \cdot \mathrm{dst}(\overline{x})$$

Note: The intermediate operations are being done using single precision floating point data type.

API

struct **reorder** : public dnnl::*primitive*

Reorder primitive.

Public Functions

reorder()

Default constructor. Produces an empty object.

reorder(const *primitive_desc* &pd)

Constructs a reorder primitive.

Parameters

pd – Primitive descriptor for reorder primitive.

reorder(const *memory* &src, const *memory* &dst, const *primitive_attr* &attr = *primitive_attr*())

Constructs a reorder primitive that would reorder data between memory objects having the same memory descriptors as memory objects `src` and `dst`.

Parameters

- **src** – Source memory object.
- **dst** – Destination memory object.
- **attr** – Primitive attributes to use (optional).

void **execute**(const *stream* &astream, *memory* &src, *memory* &dst) const

Executes the reorder primitive.

Parameters

- **astream** – Stream object. The stream must belong to the same engine as the primitive.
- **src** – Source memory object.
- **dst** – Destination memory object.

struct **primitive_desc** : public dnnl::*primitive_desc_base*

Primitive descriptor for a reorder primitive.

Public Functions

primitive_desc()

Default constructor. Produces an empty object.

primitive_desc(const *engine* &src_engine, const *memory*::*desc* &src_md, const *engine* &dst_engine, const *memory*::*desc* &dst_md, const *primitive_attr* &attr = *primitive_attr*(), bool allow_empty = false)

Constructs a primitive descriptor for reorder primitive.

Note: If `allow_empty` is true, the constructor does not throw if a primitive descriptor cannot be created.

Parameters

- **src_engine** – Engine on which the source memory object will be located.
- **src_md** – Source memory descriptor.
- **dst_engine** – Engine on which the destination memory object will be located.
- **dst_md** – Destination memory descriptor.
- **attr** – Primitive attributes to use (optional).
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *memory* &src, const *memory* &dst, const *primitive_attr* &attr = *primitive_attr*(), bool allow_empty = false)

Constructs a primitive descriptor for reorder primitive.

Parameters

- **src** – Source memory object. It is used to obtain the source memory descriptor and engine.
- **dst** – Destination memory object. It is used to obtain the destination memory descriptor and engine.
- **attr** – Primitive attributes to use (optional).
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

engine **get_src_engine**() const

Returns the engine on which the source memory is allocated.

Returns

The engine on which the source memory is allocated.

engine **get_dst_engine**() const

Returns the engine on which the destination memory is allocated.

Returns

The engine on which the destination memory is allocated.

memory::*desc* **src_desc**() const

Returns a source memory descriptor.

Returns

Source memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **dst_desc**() const

Returns a destination memory descriptor.

Returns

Destination memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a destination parameter.

4.5.16 Resampling

The resampling primitive computes forward or backward resampling operation on 1D, 2D, or 3D spatial data. Resampling performs spatial scaling of original tensor using one of the supported interpolation algorithms:

- Nearest Neighbor
- Linear (or Bilinear for 2D spatial tensor, Trilinear for 3D spatial tensor).

Resampling operation is defined by the source tensor and scaling factors in each spatial dimension. Upsampling and downsampling are the alternative terms for resampling that are used when all scaling factors are greater (upsampling) or less (downsampling) than one.

The resampling operation is defined by the following formulas. We show formulas only for 2D spatial data which are straightforward to generalize to cases of higher and lower dimensions. Variable names follow the standard *Conventions*.

Let src and dst be $N \times C \times IH \times IW$ and $N \times C \times OH \times OW$ tensors respectively. Let $F_h = \frac{OH}{IH}$ and $F_w = \frac{OW}{IW}$ define scaling factors in each spatial dimension.

The following formulas show how oneDNN computes resampling for nearest neighbor and bilinear interpolation methods. To further simplify the formulas, we assume the following:

- $\mathrm{src}(n, ic, ih, iw) = 0$ if $ih < 0$ or $iw < 0$,
- $\mathrm{src}(n, ic, ih, iw) = \mathrm{src}(n, ic, IH - 1, iw)$ if $ih \geq IH$,
- $\mathrm{src}(n, ic, ih, iw) = \mathrm{src}(n, ic, ih, IW - 1)$ if $iw \geq IW$.

Forward

Nearest Neighbor Resampling

$$\mathrm{dst}(n, c, oh, ow) = \mathrm{src}(n, c, ih, iw)$$

where

- $ih = [\frac{oh+0.5}{F_h} - 0.5]$,
- $iw = [\frac{ow+0.5}{F_w} - 0.5]$.

Bilinear Resampling

$$\begin{aligned}\mathrm{dst}(n, c, oh, ow) = &\mathrm{src}(n, c, ih_0, iw_0) \cdot W_{ih} \cdot W_{iw} + \\ &\mathrm{src}(n, c, ih_1, iw_0) \cdot (1 - W_{ih}) \cdot W_{iw} + \\ &\mathrm{src}(n, c, ih_0, iw_1) \cdot W_{ih} \cdot (1 - W_{iw}) + \\ &\mathrm{src}(n, c, ih_1, iw_1) \cdot (1 - W_{ih}) \cdot (1 - W_{iw})\end{aligned}$$

where

- $ih_0 = \left\lfloor \frac{oh+0.5}{F_h} - 0.5 \right\rfloor$,
- $ih_1 = \left\lceil \frac{oh+0.5}{F_h} - 0.5 \right\rceil$,
- $iw_0 = \left\lfloor \frac{ow+0.5}{F_w} - 0.5 \right\rfloor$,
- $iw_1 = \left\lceil \frac{ow+0.5}{F_w} - 0.5 \right\rceil$,
- $W_{ih} = \frac{oh+0.5}{F_h} - 0.5 - ih_0$,
- $W_{iw} = \frac{ow+0.5}{F_w} - 0.5 - iw_0$.

Difference Between Forward Training and Forward Inference

There is no difference between the *forward_training* and *forward_inference* propagation kinds.

Backward

The backward propagation computes $\mathrm{diff\_src}$ based on $\mathrm{diff\_dst}$.

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	*DNNL_ARG_SRC*
dst	*DNNL_ARG_DST*
diff_src	*DNNL_ARG_DIFF_SRC*
diff_dst	*DNNL_ARG_DIFF_DST*

Operation Details

1. Resampling implementation supports data with arbitrary data tag (*nchw*, *nhwc*, etc.) but memory tags for `src` and `dst` are expected to be the same. Resampling primitive supports `dst` and `diff_src` memory tag *any* and can define destination format based on source format.
2. Resampling descriptor can be created by specifying the source and destination memory descriptors, only the source descriptor and floating point factors, or the source and destination memory descriptors and factors. In case when user does not provide the destination descriptor, the destination dimensions are deduced using the factors: $output\_spatial\_size = \left\lfloor \frac{input\_spatial\_size}{F} \right\rfloor$.

Note: Resampling algorithm uses factors as defined by the relation $F = \frac{output\_spatial\_size}{input\_spatial\_size}$ that do not necessarily equal to the ones passed by the user.

Data Types Support

Resampling primitive supports the following combination of data types for source and destination memory objects.

Note: Here we abbreviate data types names for readability. For example, *dnnl::memory::data_type::f32* is abbreviated to *f32*.

Propagation	Source / Destination
forward / backward	*f32*, *bf16*
forward	*f16*, *s8*, *u8*

Post-ops and Attributes

The resampling primitive does not support any post-ops or attributes.

API

struct **resampling_forward** : public dnnl::*primitive*

Resampling forward propagation.

Public Functions

resampling_forward()

Default constructor. Produces an empty object.

resampling_forward(const *primitive_desc* &pd)

Constructs a resampling forward propagation primitive.

Parameters
pd – Primitive descriptor for a resampling forward propagation primitive.

struct **primitive_desc** : public dnnl::*primitive_desc*

Primitive descriptor for a resampling forward propagation primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *algorithm* aalgorithm, const *memory*::*desc* &src_desc, const *memory*::*desc* &dst_desc, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a resampling forward propagation primitive using source and destination memory descriptors.

Note: Destination memory descriptor may be initialized with *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters
- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- **aalgorithm** – resampling algorithm kind: either *dnnl::algorithm::resampling_nearest*, or *dnnl::algorithm::resampling_linear*
- **src_desc** – Source memory descriptor.
- **dst_desc** – Destination memory descriptor.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *algorithm* aalgorithm, const std::vector<float> &factors, const *memory*::*desc* &src_desc, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a resampling forward propagation primitive using source memory descriptor and factors.

Parameters

- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- **aalgorithm** – resampling algorithm kind: either *dnnl::algorithm::resampling_nearest*, or *dnnl::algorithm::resampling_linear*
- **factors** – Vector of scaling factors for spatial dimension.
- **src_desc** – Source memory descriptor.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *algorithm* aalgorithm, const std::vector<float> &factors, const *memory*::*desc* &src_desc, const *memory*::*desc* &dst_desc, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a resampling forward propagation primitive.

Note: The destination memory descriptor may be initialized with *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- **aalgorithm** – resampling algorithm kind: either *dnnl::algorithm::resampling_nearest*, or *dnnl::algorithm::resampling_linear*
- **factors** – Vector of scaling factors for spatial dimension.
- **src_desc** – Source memory descriptor.
- **dst_desc** – Destination memory descriptor.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() const

Returns a source memory descriptor.

Returns

Source memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **dst_desc**() const

Returns a destination memory descriptor.

Returns

Destination memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a destination parameter.

struct **resampling_backward** : public dnnl::*primitive*

Resampling backward propagation primitive.

Public Functions

resampling_backward()

Default constructor. Produces an empty object.

resampling_backward(const *primitive_desc* &pd)

Constructs a resampling backward propagation primitive.

Parameters

pd – Primitive descriptor for a resampling backward propagation primitive.

struct **primitive_desc** : public dnnl::*primitive_desc*

Primitive descriptor for resampling backward propagation primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *algorithm* aalgorithm, const *memory*::*desc* &diff_src_desc, const *memory*::*desc* &diff_dst_desc, const *resampling_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a resampling backward propagation primitive using source and destination memory descriptors.

Parameters

- **aengine** – Engine to use.
- **aalgorithm** – resampling algorithm kind: either *dnnl::algorithm::resampling_nearest*, or *dnnl::algorithm::resampling_linear*
- **diff_src_desc** – Diff source memory descriptor.
- **diff_dst_desc** – Diff destination memory descriptor.
- **hint_fwd_pd** – Primitive descriptor for a resampling forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, *algorithm* aalgorithm, const std::vector<float> &factors, const *memory*::*desc* &diff_src_desc, const *memory*::*desc* &diff_dst_desc, const *resampling_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for resampling backward propagation primitive.

Parameters

- **aengine** – Engine to use.
- **aalgorithm** – resampling algorithm kind: either *dnnl::algorithm::resampling_nearest*, or *dnnl::algorithm::resampling_linear*
- **factors** – Vector of scaling factors for spatial dimension.
- **diff_src_desc** – Diff source memory descriptor.
- **diff_dst_desc** – Diff destination memory descriptor.
- **hint_fwd_pd** – Primitive descriptor for a resampling forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.

- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **diff_src_desc**() const

Returns a diff source memory descriptor.

Returns

Diff source memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a diff source memory with.

memory::*desc* **diff_dst_desc**() const

Returns a diff destination memory descriptor.

Returns

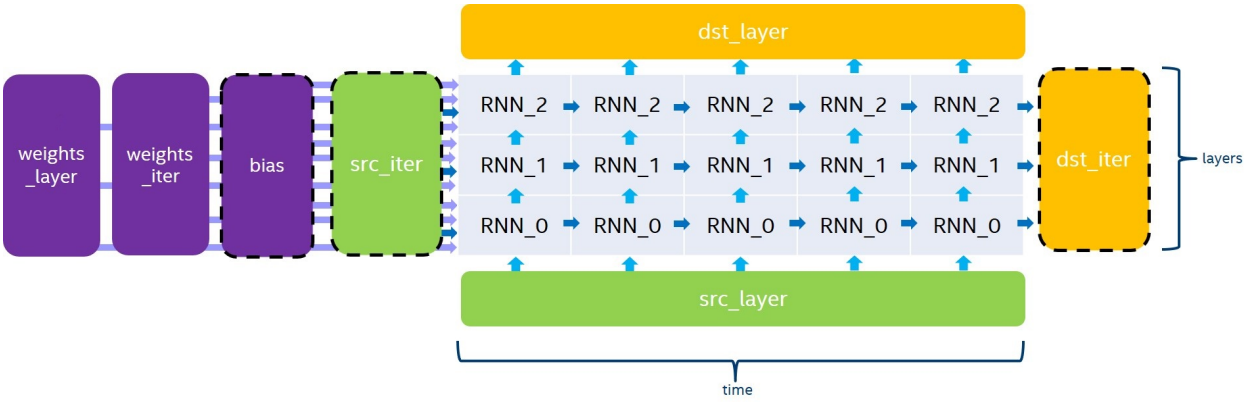
Diff destination memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a diff destination parameter.

4.5.17 RNN

The RNN primitive computes a stack of unrolled recurrent cells, as depicted in Figure 1. bias, src_iter and dst_iter are optional parameters. If not provided, bias and src_iter default to 0. Variable names follow the standard *Conventions*.



The RNN primitive supports four modes for evaluation direction:

- `left2right` will process the input data timestamps by increasing order,
- `right2left` will process the input data timestamps by decreasing order,
- `bidirectional_concat` will process all the stacked layers from `left2right` and from `right2left` independently, and will concatenate the output in dst_layer over the channel dimension,
- `bidirectional_sum` will process all the stacked layers from `left2right` and from `right2left` independently, and will sum the two outputs to dst_layer.

Even though the RNN primitive supports passing a different number of channels for src_layer, src_iter, dst_layer, and dst_iter, we always require the following conditions in order for the dimension to be consistent:

- $channels(\text{dst\_layer}) = channels(\text{dst\_iter})$,
- when $T > 1$, $channels(\text{src\_iter}) = channels(\text{dst\_iter})$,
- when $L > 1$, $channels(\text{src\_layer}) = channels(\text{dst\_layer})$,
- when using the `bidirectional_concat` direction, $channels(\text{dst\_layer}) = 2 * channels(\text{dst\_iter})$.

The general formula for the execution of a stack of unrolled recurrent cells depends on the current iteration of the previous layer ($h_{t,l-1}$ and $c_{t,l-1}$) and the previous iteration of the current layer ($h_{t-1,l}$). Here is the exact equation for non-LSTM cells:

$$h_{t,l} = Cell(h_{t,l-1}, h_{t-1,l})$$

where t, l are the indices of the timestamp and the layer of the cell being executed.

And here is the equation for LSTM cells:

$$(h_{t,l}, c_{t,l}) = Cell(h_{t,l-1}, h_{t-1,l}, c_{t-1,l})$$

where t, l are the indices of the timestamp and the layer of the cell being executed.

Cell Functions

The RNN API provides six cell functions:

- *Vanilla RNN*, a single-gate recurrent cell,
- *LSTM*, a four-gate long short-term memory cell,
- *GRU*, a three-gate gated recurrent unit cell,
- *Linear-before-reset GRU*, a three-gate recurrent unit cell with the linear layer before the reset gate.
- *AUGRU*, a three-gate gated recurrent unit cell with the attention update gate,
- *Linear-before-reset AUGRU*, a three-gate recurrent unit cell with the linear layer before the reset gate and the attention update gate.

Vanilla RNN

A single-gate recurrent cell initialized with *dnnl::vanilla_rnn_forward::primitive_desc* or *dnnl::vanilla_rnn_forward::primitive_desc* as in the following example.

```
auto vanilla_rnn_pd =
    dnnl::vanilla_rnn_forward::primitive_desc(engine, aprop,
    activation, direction, src_layer_desc, src_iter_desc,
    weights_layer_desc, weights_iter_desc, bias_desc,
    dst_layer_desc, dst_iter_desc, attr);
```

The Vanilla RNN cell should support the ReLU, Tanh and Sigmoid activation functions. The following equations defines the mathematical operation performed by the Vanilla RNN cell for the forward pass:

$$a_t = W \cdot h_{t,l-1} + U \cdot h_{t-1,l} + B$$
$$h_t = activation(a_t)$$

LSTM

LSTM (or Vanilla LSTM)

A four-gate long short-term memory recurrent cell initialized with *dnnl::lstm_forward::primitive_desc* or *dnnl::lstm_backward::primitive_desc* as in the following example.

```
auto lstm_pd = dnnl::lstm_forward::primitive_desc(engine, aprop,
    direction, src_layer_desc, src_iter_h_desc, src_iter_c_desc,
    weights_layer_desc, weights_iter_desc, bias_desc,
    dst_layer_desc, dst_iter_h_desc, dst_iter_c_desc, attr);
```

Note that for all tensors with a dimension depending on the gates number, we implicitly require the order of these gates to be i, f, $\tilde{c}$, and o. The following equation gives the mathematical description of these gates and output for the forward pass:

$$
\begin{aligned}
i_t &= \sigma(W_i \cdot h_{t,l-1} + U_i \cdot h_{t-1,l} + B_i) \\
f_t &= \sigma(W_f \cdot h_{t,l-1} + U_f \cdot h_{t-1,l} + B_f)
\end{aligned}
$$

$$
\begin{aligned}
\tilde{c}_t &= tanh(W_{\tilde{c}} \cdot h_{t,l-1} + U_{\tilde{c}} \cdot h_{t-1,l} + B_{\tilde{c}}) \\
c_t &= f_t * c_{t-1} + i_t * \tilde{c}_t
\end{aligned}
$$

$$
\begin{aligned}
o_t &= \sigma(W_o \cdot h_{t,l-1} + U_o \cdot h_{t-1,l} + B_o) \\
h_t &= tanh(c_t) * o_t
\end{aligned}
$$

where W_* are stored in weights_layer, U_* are stored in weights_iter and B_* are stored in bias.

Note: In order for the dimensions to be consistent, we require $channels(\text{src\_iter\_c}) = channels(\text{dst\_iter\_c}) = channels(\text{dst\_iter})$.

LSTM with Peephole

A four-gate long short-term memory recurrent cell with peephole initialized with *dnnl::lstm_forward::primitive_desc* or *dnnl::lstm_backward::primitive_desc* as in the following example.

```
auto lstm_pd = dnnl::lstm_forward::primitive_desc(engine, aprop,
    direction, src_layer_desc, src_iter_h_desc, src_iter_c_desc,
    weights_layer_desc, weights_iter_desc, weights_peephole_desc,
    bias_desc, dst_layer_desc, dst_iter_h_desc, dst_iter_c_desc,
    attr);
```

Similarly to vanilla LSTM, we implicitly require the order of these gates to be i, f, $\tilde{c}$, and o. For peephole weights, the gates order is:math:*i*, f, o. The following equation gives the mathematical description of these gates and output for the

forward pass:

$$
\begin{aligned}
i_t &= \sigma(W_i \cdot h_{t,l-1} + U_i \cdot h_{t-1,l} + P_i \cdot c_{t-1} + B_i) \\
f_t &= \sigma(W_f \cdot h_{t,l-1} + U_f \cdot h_{t-1,l} + P_f \cdot c_{t-1} + B_f)
\end{aligned}
$$

$$
\begin{aligned}
\tilde{c}_t &= tanh(W_{\tilde{c}} \cdot h_{t,l-1} + U_{\tilde{c}} \cdot h_{t-1,l} + B_{\tilde{c}}) \\
c_t &= f_t * c_{t-1} + i_t * \tilde{c}_t
\end{aligned}
$$

$$
\begin{aligned}
o_t &= \sigma(W_o \cdot h_{t,l-1} + U_o \cdot h_{t-1,l} + P_o \cdot c_t + B_o) \\
h_t &= tanh(c_t) * o_t
\end{aligned}
$$

where P_* are stored in `weights_peephole`, and the other parameters are the same as in vanilla LSTM.

Note: If the `weights_peephole_desc` passed to the primitive descriptor constructor is a zero memory descriptor, the primitive will behave the same as in LSTM primitive without peephole.

LSTM with Projection

A four-gate long short-term memory recurrent cell with projection initialized with *dnnl::lstm_forward::primitive_desc* or *dnnl::lstm_backward::primitive_desc* as in the following example.

```
auto lstm_pd = dnnl::lstm_forward::primitive_desc(engine, aprop,
    direction, src_layer_desc, src_iter_h_desc, src_iter_c_desc,
    weights_layer_desc, weights_iter_desc, weights_peephole_desc,
    weights_projection_desc, bias_desc, dst_layer_desc,
    dst_iter_h_desc, dst_iter_c_desc, attr);
```

Similarly to vanilla LSTM, we implicitly require the order of the gates to be *i*, f, $\tilde{c}$, and *o* for all tensors with a dimension depending on the gates. The following equation gives the mathematical description of these gates and output for the forward pass (for simplicity, LSTM without peephole is shown):

$$
\begin{aligned}
i_t &= \sigma(W_i \cdot h_{t,l-1} + U_i \cdot h_{t-1,l} + B_i) \\
f_t &= \sigma(W_f \cdot h_{t,l-1} + U_f \cdot h_{t-1,l} + B_f)
\end{aligned}
$$

$$
\begin{aligned}
\tilde{c}_t &= \tanh(W_{\tilde{c}} \cdot h_{t,l-1} + U_{\tilde{c}} \cdot h_{t-1,l} + B_{\tilde{c}}) \\
c_t &= f_t * c_{t-1} + i_t * \tilde{c}_t
\end{aligned}
$$

$$
\begin{aligned}
o_t &= \sigma(W_o \cdot h_{t,l-1} + U_o \cdot h_{t-1,l} + B_o) \\
h_t &= R \cdot (\tanh(c_t) * o_t)
\end{aligned}
$$

where R is stored in `weights_projection`, and the other parameters are the same as in vanilla LSTM.

Note: If the `weights_projection_desc` passed to the primitive descriptor constructor is a zero memory descriptor, the primitive will behave the same as in LSTM primitive without projection.

GRU

A three-gate gated recurrent unit cell, initialized with *dnnl::gru_forward::primitive_desc* or *dnnl::gru_backward::primitive_desc* as in the following example.

```
auto gru_pd = dnnl::gru_forward::primitive_desc(engine, aprop,
    direction, src_layer_desc, src_iter_desc, weights_layer_desc,
    weights_iter_desc, bias_desc, dst_layer_desc, dst_iter_desc,
    attr);
```

Note that for all tensors with a dimension depending on the gates number, we implicitly require the order of these gates to be:math:u, r, and o. The following equation gives the mathematical definition of these gates.

$$
\begin{aligned}
u_t &= \sigma(W_u \cdot h_{t,l-1} + U_u \cdot h_{t-1,l} + B_u) \\
r_t &= \sigma(W_r \cdot h_{t,l-1} + U_r \cdot h_{t-1,l} + B_r) \\
o_t &= tanh(W_o \cdot h_{t,l-1} + U_o \cdot (r_t * h_{t-1,l}) + B_o) \\
h_t &= u_t * h_{t-1,l} + (1 - u_t) * o_t
\end{aligned}
$$

where W_* are in weights_layer, U_* are in weights_iter, and B_* are stored in bias.

Note: If you need to replace u_t by $(1 - u_t)$ when computing h_t, you can achieve this by multiplying W_u, U_u and B_u by -1. This is possible as $u_t = \sigma(W_u \cdot h_{t,l-1} + U_u \cdot h_{t-1,l} + B_u)$, and $1^{\smile}\sigma(a) = \sigma(-a)$.

Linear-Before-Reset GRU

A three-gate gated recurrent unit cell with linear layer applied before the reset gate, initialized with *dnnl::lbr_gru_forward::primitive_desc* or *dnnl::lbr_gru_backward::primitive_desc* as in the following example.

```
auto lbr_gru_pd = dnnl::lbr_gru_forward::primitive_desc(engine,
    aprop, direction, src_layer_desc, src_iter_desc,
    weights_layer_desc, weights_iter_desc, bias_desc,
    dst_layer_desc, dst_iter_desc, attr);
```

The following equation describes the mathematical behavior of the Linear-Before-Reset GRU cell.

$$
\begin{aligned}
u_t &= \sigma(W_u \cdot h_{t,l-1} + U_u \cdot h_{t-1,l} + B_u) \\
r_t &= \sigma(W_r \cdot h_{t,l-1} + U_r \cdot h_{t-1,l} + B_r) \\
o_t &= tanh(W_o \cdot h_{t,l-1} + r_t * (U_o \cdot h_{t-1,l} + B_{u'}) + B_o) \\
h_t &= u_t * h_{t-1,l} + (1 - u_t) * o_t
\end{aligned}
$$

Note that for all tensors with a dimension depending on the gates number, except the bias, we implicitly require the order of these gates to be u, r, and o. For the bias tensor, we implicitly require the order of the gates to be u, r, o, and u'.

Note: If you need to replace u_t by $(1 - u_t)$ when computing h_t, you can achieve this by multiplying W_u, U_u and B_u by -1. This is possible as $u_t = \sigma(W_u \cdot h_{t,l-1} + U_u \cdot h_{t-1,l} + B_u)$, and $1^{\smile}\sigma(a) = \sigma(-a)$.

AUGRU

A three-gate gated recurrent unit cell, initialized with `dnnl::augru_forward::primitive_desc` or `dnnl::augru_backward::primitive_desc` as in the following example.

```
auto augru_pd = dnnl::augru_forward::primitive_desc(engine, aprop,
    direction, src_layer_desc, src_iter_desc, attention_desc,
    weights_layer_desc, weights_iter_desc, bias_desc,
    dst_layer_desc, dst_iter_desc, attr);
```

Note that for all tensors with a dimension depending on the gate number, we implicitly require the order of these gates to be u, r, and o. The following equation gives the mathematical definition of these gates.

$$
\begin{aligned}
u_t &= \sigma(W_u \cdot h_{t,l-1} + U_u \cdot h_{t-1,l} + B_u) \\
r_t &= \sigma(W_r \cdot h_{t,l-1} + U_r \cdot h_{t-1,l} + B_r) \\
o_t &= \tanh(W_o \cdot h_{t,l-1} + U_o \cdot (r_t * h_{t-1,l}) + B_o) \\
\tilde{u}_t &= (1 - a_t) * u_t \\
h_t &= \tilde{u}_t * h_{t-1,l} + (1 - \tilde{u}_t) * o_t
\end{aligned}
$$

where W_* are in weightslayer, U_* are in weightsiter, and B_* are stored in bias.

Linear-Before-Reset AUGRU

A three-gate gated recurrent unit cell with linear layer applied before the reset gate, initialized with `dnnl::lbr_augru_forward::primitive_desc` or `dnnl::lbr_augru_backward::primitive_desc` as in the following example.

```
auto lbr_augru_pd =
    dnnl::lbr_augru_forward::primitive_desc(engine, aprop,
    direction, src_layer_desc, src_iter_desc, attention_desc,
    weights_layer_desc, weights_iter_desc, bias_desc,
    dst_layer_desc, dst_iter_desc, attr);
```

The following equation describes the mathematical behavior of the Linear-Before-Reset AUGRU cell.

$$
\begin{aligned}
u_t &= \sigma(W_u \cdot h_{t,l-1} + U_u \cdot h_{t-1,l} + B_u) \\
r_t &= \sigma(W_r \cdot h_{t,l-1} + U_r \cdot h_{t-1,l} + B_r) \\
o_t &= \tanh(W_o \cdot h_{t,l-1} + r_t * (U_o \cdot h_{t-1,l} + B_{u'}) + B_o) \\
\tilde{u}_t &= (1 - a_t) * u_t \\
h_t &= \tilde{u}_t * h_{t-1,l} + (1 - \tilde{u}_t) * o_t
\end{aligned}
$$

Note that for all tensors with a dimension depending on the gate number, except the bias, we implicitly require the order of these gates to be u, r, and o. For the bias tensor, we implicitly require the order of the gates to be u, r, o, and u'.

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src_layer	*DNNL_ARG_SRC_LAYER*
src_iter	*DNNL_ARG_SRC_ITER*
src_iter_c	*DNNL_ARG_SRC_ITER_C*
weights_layer	*DNNL_ARG_WEIGHTS_LAYER*
weights_iter	*DNNL_ARG_WEIGHTS_ITER*
weights_peephole	DNNL_ARG_WEIGHTS_PEEPHOLE
weights_projection	DNNL_ARG_WEIGHTS_PROJECTION
bias	*DNNL_ARG_BIAS*
dst_layer	*DNNL_ARG_DST_LAYER*
dst_iter	*DNNL_ARG_DST_ITER*
dst_iter_c	*DNNL_ARG_DST_ITER_C*
workspace	*DNNL_ARG_WORKSPACE*
diff_src_layer	*DNNL_ARG_DIFF_SRC_LAYER*
diff_src_iter	*DNNL_ARG_DIFF_SRC_ITER*
diff_src_iter_c	*DNNL_ARG_DIFF_SRC_ITER_C*
diff_weights_layer	*DNNL_ARG_DIFF_WEIGHTS_LAYER*
diff_weights_iter	*DNNL_ARG_DIFF_WEIGHTS_ITER*
diff_weights_peephole	DNNL_ARG_DIFF_WEIGHTS_PEEPHOLE
diff_weights_projection	DNNL_ARG_DIFF_WEIGHTS_PROJECTION
diff_bias	*DNNL_ARG_DIFF_BIAS*
diff_dst_layer	*DNNL_ARG_DIFF_DST_LAYER*
diff_dst_iter	*DNNL_ARG_DIFF_DST_ITER*
diff_dst_iter_c	*DNNL_ARG_DIFF_DST_ITER_C*

Operation Details

N/A

Data Types Support

The following table lists the combination of data types that should be supported by the RNN primitive for each input and output memory object.

Note: Here we abbreviate data types names for readability. For example, *dnnl::memory::data_type::f32* is abbreviated to *f32*.

Propagation	**Cell Function**	**Input Data**	**Recurrent Data** (1)	**Weights**	**Bias**	**Output Data**
Forward / Backward	All	*f32*	*f32*	*f32*	*f32*	*f32*
Forward / Backward (2)	All (3)	*bf16*	*bf16*	*bf16*	*f32*	*bf16*
Forward	All (3)	*f16*	*f16*	*f16*	*f16*	*f16*
Forward inference	Vanilla LSTM	*u8*	*u8*	*s8*	*f32*	*u8*, *f32*

(1) With LSTM and Peephole LSTM cells, the cell state data type is always f32.

(2) In backward propagation, all `diff_*` tensors are in f32.

(3) Projection LSTM is not defined yet.

Data Representation

In the oneDNN programming model, the RNN primitive is one of a few that support the placeholder memory format #dnnl::memory::format_tag::any (shortened to `any` from now on) and can define data and weight memory objects format based on the primitive parameters.

The following table summarizes the data layouts supported by the RNN primitive.

Input/Output Data	Recurrent Data	Layer and Iteration Weights	Peephole Weights and Bias	Projection LSTM Weights
any	*any*	*any*	*ldgo*	*any*, *ldio* (Forward propagation)
ntc, *tnc*	*ldnc*	*ldigo*, *ldgoi*	*ldgo*	*any*, *ldio* (Forward propagation)

While an RNN primitive can be created with memory formats specified explicitly, the performance is likely to be sub-optimal. When using *any* it is necessary to first create an RNN primitive descriptor and then query it for the actual data and weight memory objects formats.

Note: The RNN primitive should support padded tensors and views. So even if two memory descriptors share the same data layout, they might still be different.

Post-ops and Attributes

Currently post-ops and attributes are only used by the int8 variant of LSTM.

API

enum class `dnnl::rnn_flags` : unsigned

RNN cell flags.

Values:

enumerator **undef**

Undefined RNN flags.

enum class `dnnl::rnn_direction`

A direction of RNN primitive execution.

Values:

enumerator **undef**

Undefined RNN direction.

enumerator **unidirectional_left2right**

Unidirectional execution of RNN primitive from left to right.

enumerator **unidirectional_right2left**

Unidirectional execution of RNN primitive from right to left.

enumerator **bidirectional_concat**

Bidirectional execution of RNN primitive with concatenation of the results.

enumerator **bidirectional_sum**

Bidirectional execution of RNN primitive with summation of the results.

enumerator **unidirectional**

Alias for *dnnl::rnn_direction::unidirectional_left2right*.

struct **vanilla_rnn_forward** : public dnnl::*primitive*

Vanilla RNN forward propagation primitive.

Public Functions

vanilla_rnn_forward()

Default constructor. Produces an empty object.

vanilla_rnn_forward(const *primitive_desc* &pd)

Constructs a vanilla RNN forward propagation primitive.

Parameters
pd – Primitive descriptor for a vanilla RNN forward propagation primitive.

struct **primitive_desc** : public dnnl::*rnn_primitive_desc_base*

Primitive descriptor for a vanilla RNN forward propagation primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *algorithm* activation, *rnn_direction* direction, const *memory*::*desc* &src_layer_desc, const *memory*::*desc* &src_iter_desc, const *memory*::*desc* &weights_layer_desc, const *memory*::*desc* &weights_iter_desc, const *memory*::*desc* &bias_desc, const *memory*::*desc* &dst_layer_desc, const *memory*::*desc* &dst_iter_desc, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a vanilla RNN forward propagation primitive.

The following arguments may point to a zero memory descriptor:
- `src_iter_desc`,
- `bias_desc`,
- `dst_iter_desc`.

This would then indicate that the RNN forward propagation primitive should not use them and should default to zero values instead.

Note: All memory descriptors except `src_iter_desc` can be initialized with an *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- **activation** – Activation kind. Possible values are *dnnl::algorithm::eltwise_relu*, *dnnl::algorithm::eltwise_tanh*, or *dnnl::algorithm::eltwise_logistic*.
- **direction** – RNN direction. See *dnnl::rnn_direction* for more info.
- **src_layer_desc** – Memory descriptor for the input vector.
- **src_iter_desc** – Memory descriptor for the input recurrent hidden state vector.
- **weights_layer_desc** – Memory descriptor for the weights applied to the layer input.
- **weights_iter_desc** – Memory descriptor for the weights applied to the recurrent input.
- **bias_desc** – Bias memory descriptor.
- **dst_layer_desc** – Memory descriptor for the output vector.
- **dst_iter_desc** – Memory descriptor for the output recurrent hidden state vector.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *algorithm* activation, *rnn_direction* direction, const *memory*::*desc* &src_layer_desc, const *memory*::*desc* &src_iter_desc, const *memory*::*desc* &weights_layer_desc, const *memory*::*desc* &weights_iter_desc, const *memory*::*desc* &bias_desc, const *memory*::*desc* &dst_layer_desc, const *memory*::*desc* &dst_iter_desc, float alpha, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a vanilla RNN forward propagation primitive with alpha parameter.

The following arguments may point to a zero memory descriptor:

- `src_iter_desc`,
- `bias_desc`,
- `dst_iter_desc`.

This would then indicate that the RNN forward propagation primitive should not use them and should default to zero values instead.

Note: All memory descriptors except `src_iter_desc` can be initialized with an *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- **activation** – Activation kind. Possible values are *dnnl::algorithm::eltwise_relu*, *dnnl::algorithm::eltwise_tanh*, or *dnnl::algorithm::eltwise_logistic*.
- **direction** – RNN direction. See *dnnl::rnn_direction* for more info.
- **src_layer_desc** – Memory descriptor for the input vector.

- **src_iter_desc** – Memory descriptor for the input recurrent hidden state vector.
- **weights_layer_desc** – Memory descriptor for the weights applied to the layer input.
- **weights_iter_desc** – Memory descriptor for the weights applied to the recurrent input.
- **bias_desc** – Bias memory descriptor.
- **dst_layer_desc** – Memory descriptor for the output vector.
- **dst_iter_desc** – Memory descriptor for the output recurrent hidden state vector.
- **alpha** – Negative slope if activation is *dnnl::algorithm::eltwise_relu*.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_layer_desc**() const

Returns source layer memory descriptor.

Returns

Source layer memory descriptor.

memory::*desc* **src_iter_desc**() const

Returns source iteration memory descriptor.

Returns

Source iteration memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a source iteration parameter.

memory::*desc* **weights_layer_desc**() const

Returns weights layer memory descriptor.

Returns

Weights layer memory descriptor.

memory::*desc* **weights_iter_desc**() const

Returns weights iteration memory descriptor.

Returns

Weights iteration memory descriptor.

memory::*desc* **bias_desc**() const

Returns bias memory descriptor.

Returns

Bias memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a bias parameter.

memory::*desc* **dst_layer_desc**() const

Returns destination layer memory descriptor.

Returns

Destination layer memory descriptor.

memory::*desc* **dst_iter_desc**() const

Returns destination iteration memory descriptor.

Returns

Destination iteration memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a destination iteration parameter.

memory::*desc* **workspace_desc**() const

Returns the workspace memory descriptor.

Returns
Workspace memory descriptor.
Returns
A zero memory descriptor if the primitive does not require workspace parameter.

algorithm **get_cell_kind**() const

Returns an RNN cell kind parameter.
Returns
An RNN cell kind parameter.
Returns
dnnl::algorithm::undef if the primitive does not have an RNN cell kind parameter.

prop_kind **get_prop_kind**() const

Returns a propagation kind.
Returns
A propagation kind.
Returns
dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

algorithm **get_activation_kind**() const

Returns an RNN activation kind parameter.
Returns
An RNN activation kind parameter.
Returns
dnnl::algorithm::undef if the primitive does not have an RNN activation kind parameter.

rnn_direction **get_direction**() const

Returns an RNN direction parameter.
Returns
An RNN direction parameter.
Returns
dnnl::rnn_direction::undef if the primitive does not have an RNN direction parameter.

float **get_alpha**() const

Returns an alpha.
Returns
An alpha.
Returns
Zero if the primitive does not have an alpha parameter.

float **get_beta**() const

Returns a beta.
Returns
A beta.
Returns
Zero if the primitive does not have a beta parameter.

struct **vanilla_rnn_backward** : public dnnl::*primitive*

Vanilla RNN backward propagation primitive.

Public Functions

vanilla_rnn_backward()

Default constructor. Produces an empty object.

vanilla_rnn_backward(const *primitive_desc* &pd)

Constructs a vanilla RNN backward propagation primitive.

Parameters

pd – Primitive descriptor for a vanilla RNN backward propagation primitive.

struct **primitive_desc** : public dnnl::*rnn_primitive_desc_base*

Primitive descriptor for an RNN backward propagation primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *algorithm* activation, *rnn_direction* direction, const *memory*::*desc* &src_layer_desc, const *memory*::*desc* &src_iter_desc, const *memory*::*desc* &weights_layer_desc, const *memory*::*desc* &weights_iter_desc, const *memory*::*desc* &bias_desc, const *memory*::*desc* &dst_layer_desc, const *memory*::*desc* &dst_iter_desc, const *memory*::*desc* &diff_src_layer_desc, const *memory*::*desc* &diff_src_iter_desc, const *memory*::*desc* &diff_weights_layer_desc, const *memory*::*desc* &diff_weights_iter_desc, const *memory*::*desc* &diff_bias_desc, const *memory*::*desc* &diff_dst_layer_desc, const *memory*::*desc* &diff_dst_iter_desc, const *vanilla_rnn_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a vanilla RNN backward propagation primitive.

The following arguments may point to a zero memory descriptor:
- `src_iter_desc` together with `diff_src_iter_desc`,
- `bias_desc` together with `diff_bias_desc`,
- `dst_iter_desc` together with `diff_dst_iter_desc`.

This would then indicate that the RNN backward propagation primitive should not use the respective data and should use zero values instead.

Note: All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Must be *dnnl::prop_kind::backward*.
- **activation** – Activation kind. Possible values are *dnnl::algorithm::eltwise_relu*, *dnnl::algorithm::eltwise_tanh*, or *dnnl::algorithm::eltwise_logistic*.
- **direction** – RNN direction. See *dnnl::rnn_direction* for more info.
- **src_layer_desc** – Memory descriptor for the input vector.
- **src_iter_desc** – Memory descriptor for the input recurrent hidden state vector.
- **weights_layer_desc** – Memory descriptor for the weights applied to the layer input.
- **weights_iter_desc** – Memory descriptor for the weights applied to the recurrent input.
- **bias_desc** – Bias memory descriptor.

- **dst_layer_desc** – Memory descriptor for the output vector.
- **dst_iter_desc** – Memory descriptor for the output recurrent hidden state vector.
- **diff_src_layer_desc** – Memory descriptor for the diff of input vector.
- **diff_src_iter_desc** – Memory descriptor for the diff of input recurrent hidden state vector.
- **diff_weights_layer_desc** – Memory descriptor for the diff of weights applied to the layer input.
- **diff_weights_iter_desc** – Memory descriptor for the diff of weights applied to the recurrent input.
- **diff_bias_desc** – Diff bias memory descriptor.
- **diff_dst_layer_desc** – Memory descriptor for the diff of output vector.
- **diff_dst_iter_desc** – Memory descriptor for the diff of output recurrent hidden state vector.
- **hint_fwd_pd** – Primitive descriptor for a vanilla RNN forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *algorithm* activation, *rnn_direction* direction, const *memory*::*desc* &src_layer_desc, const *memory*::*desc* &src_iter_desc, const *memory*::*desc* &weights_layer_desc, const *memory*::*desc* &weights_iter_desc, const *memory*::*desc* &bias_desc, const *memory*::*desc* &dst_layer_desc, const *memory*::*desc* &dst_iter_desc, const *memory*::*desc* &diff_src_layer_desc, const *memory*::*desc* &diff_src_iter_desc, const *memory*::*desc* &diff_weights_layer_desc, const *memory*::*desc* &diff_weights_iter_desc, const *memory*::*desc* &diff_bias_desc, const *memory*::*desc* &diff_dst_layer_desc, const *memory*::*desc* &diff_dst_iter_desc, float alpha, const *vanilla_rnn_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a vanilla RNN backward propagation primitive with an alpha parameter.

The following arguments may point to a zero memory descriptor:
- `src_iter_desc` together with `diff_src_iter_desc`,
- `bias_desc` together with `diff_bias_desc`,
- `dst_iter_desc` together with `diff_dst_iter_desc`.

This would then indicate that the RNN backward propagation primitive should not use the respective data and should use zero values instead.

Note: All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters
- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Must be *dnnl::prop_kind::backward*.
- **activation** – Activation kind. Possible values are *dnnl::algorithm::eltwise_relu*, *dnnl::algorithm::eltwise_tanh*, or *dnnl::algorithm::eltwise_logistic*.
- **direction** – RNN direction. See *dnnl::rnn_direction* for more info.
- **src_layer_desc** – Memory descriptor for the input vector.
- **src_iter_desc** – Memory descriptor for the input recurrent hidden state vector.
- **weights_layer_desc** – Memory descriptor for the weights applied to the layer input.
- **weights_iter_desc** – Memory descriptor for the weights applied to the recurrent input.

- **bias_desc** – Bias memory descriptor.
- **dst_layer_desc** – Memory descriptor for the output vector.
- **dst_iter_desc** – Memory descriptor for the output recurrent hidden state vector.
- **diff_src_layer_desc** – Memory descriptor for the diff of input vector.
- **diff_src_iter_desc** – Memory descriptor for the diff of input recurrent hidden state vector.
- **diff_weights_layer_desc** – Memory descriptor for the diff of weights applied to the layer input.
- **diff_weights_iter_desc** – Memory descriptor for the diff of weights applied to the recurrent input.
- **diff_bias_desc** – Diff bias memory descriptor.
- **diff_dst_layer_desc** – Memory descriptor for the diff of output vector.
- **diff_dst_iter_desc** – Memory descriptor for the diff of output recurrent hidden state vector.
- **alpha** – Negative slope if activation is *dnnl::algorithm::eltwise_relu*.
- **hint_fwd_pd** – Primitive descriptor for a vanilla RNN forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_layer_desc**() const

Returns source layer memory descriptor.

Returns

Source layer memory descriptor.

memory::*desc* **src_iter_desc**() const

Returns source iteration memory descriptor.

Returns

Source iteration memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a source iteration parameter.

memory::*desc* **weights_layer_desc**() const

Returns weights layer memory descriptor.

Returns

Weights layer memory descriptor.

memory::*desc* **weights_iter_desc**() const

Returns weights iteration memory descriptor.

Returns

Weights iteration memory descriptor.

memory::*desc* **bias_desc**() const

Returns bias memory descriptor.

Returns

Bias memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a bias parameter.

memory::*desc* **dst_layer_desc**() const

Returns destination layer memory descriptor.

Returns

Destination layer memory descriptor.

memory::*desc* **dst_iter_desc**() const

Returns destination iteration memory descriptor.

Returns

Destination iteration memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a destination iteration parameter.

memory::*desc* **workspace_desc**() const

Returns the workspace memory descriptor.

Returns

Workspace memory descriptor.

Returns

A zero memory descriptor if the primitive does not require workspace parameter.

memory::*desc* **diff_src_layer_desc**() const

Returns diff source layer memory descriptor.

Returns

Diff source layer memory descriptor.

memory::*desc* **diff_src_iter_desc**() const

Returns diff source iteration memory descriptor.

Returns

Diff source iteration memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a diff source iteration parameter.

memory::*desc* **diff_weights_layer_desc**() const

Returns diff weights layer memory descriptor.

Returns

Diff weights layer memory descriptor.

memory::*desc* **diff_weights_iter_desc**() const

Returns diff weights iteration memory descriptor.

Returns

Diff weights iteration memory descriptor.

memory::*desc* **diff_bias_desc**() const

Returns diff bias memory descriptor.

Returns

Diff bias memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a diff bias parameter.

memory::*desc* **diff_dst_layer_desc**() const

Returns diff destination layer memory descriptor.

Returns

Diff destination layer memory descriptor.

memory::*desc* **diff_dst_iter_desc**() const

Returns diff destination iteration memory descriptor.

Returns

Diff destination iteration memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a diff destination iteration parameter.

algorithm **get_cell_kind**() const

Returns an RNN cell kind parameter.

Returns

An RNN cell kind parameter.

Returns

dnnl::algorithm::undef if the primitive does not have an RNN cell kind parameter.

prop_kind **get_prop_kind**() const

Returns a propagation kind.

Returns

A propagation kind.

Returns

dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

algorithm **get_activation_kind**() const

Returns an RNN activation kind parameter.

Returns

An RNN activation kind parameter.

Returns

dnnl::algorithm::undef if the primitive does not have an RNN activation kind parameter.

rnn_direction **get_direction**() const

Returns an RNN direction parameter.

Returns

An RNN direction parameter.

Returns

dnnl::rnn_direction::undef if the primitive does not have an RNN direction parameter.

float **get_alpha**() const

Returns an alpha.

Returns

An alpha.

Returns

Zero if the primitive does not have an alpha parameter.

float **get_beta**() const

Returns a beta.

Returns

A beta.

Returns

Zero if the primitive does not have a beta parameter.

struct **lstm_forward** : public dnnl::*primitive*

LSTM forward propagation primitive.

Public Functions

lstm_forward()

Default constructor. Produces an empty object.

lstm_forward(const *primitive_desc* &pd)

Constructs an LSTM forward propagation primitive.

Parameters

pd – Primitive descriptor for an LSTM forward propagation primitive.

struct **primitive_desc** : public dnnl::*rnn_primitive_desc_base*

Primitive descriptor for an LSTM forward propagation primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *rnn_direction* direction, const *memory*::*desc* &src_layer_desc, const *memory*::*desc* &src_iter_desc, const *memory*::*desc* &src_iter_c_desc, const *memory*::*desc* &weights_layer_desc, const *memory*::*desc* &weights_iter_desc, const *memory*::*desc* &weights_peephole_desc, const *memory*::*desc* &weights_projection_desc, const *memory*::*desc* &bias_desc, const *memory*::*desc* &dst_layer_desc, const *memory*::*desc* &dst_iter_desc, const *memory*::*desc* &dst_iter_c_desc, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for an LSTM (with or without peephole and with or without projection) forward propagation primitive.

The following arguments may point to a zero memory descriptor:
- `src_iter_desc` together with `src_iter_c_desc`,
- `weights_peephole_desc`,
- `bias_desc`,
- `dst_iter_desc` together with `dst_iter_c_desc`.

This would then indicate that the LSTM forward propagation primitive should not use them and should default to zero values instead.

The `weights_projection_desc` may point to a zero memory descriptor. This would then indicate that the LSTM doesn't have recurrent projection layer.

Note: All memory descriptors can be initialized with an *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters
- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- **direction** – RNN direction. See *dnnl::rnn_direction* for more info.
- **src_layer_desc** – Memory descriptor for the input vector.
- **src_iter_desc** – Memory descriptor for the input recurrent hidden state vector.
- **src_iter_c_desc** – Memory descriptor for the input recurrent cell state vector.
- **weights_layer_desc** – Memory descriptor for the weights applied to the layer input.

- **weights_iter_desc** – Memory descriptor for the weights applied to the recurrent input.
- **weights_peephole_desc** – Memory descriptor for the weights applied to the cell states (according to the Peephole LSTM formula).
- **weights_projection_desc** – Memory descriptor for the weights applied to the hidden states to get the recurrent projection (according to the Projection LSTM formula).
- **bias_desc** – Bias memory descriptor.
- **dst_layer_desc** – Memory descriptor for the output vector.
- **dst_iter_desc** – Memory descriptor for the output recurrent hidden state vector.
- **dst_iter_c_desc** – Memory descriptor for the output recurrent cell state vector.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *rnn_direction* direction, const *memory*::*desc* &src_layer_desc, const *memory*::*desc* &src_iter_desc, const *memory*::*desc* &src_iter_c_desc, const *memory*::*desc* &weights_layer_desc, const *memory*::*desc* &weights_iter_desc, const *memory*::*desc* &weights_peephole_desc, const *memory*::*desc* &bias_desc, const *memory*::*desc* &dst_layer_desc, const *memory*::*desc* &dst_iter_desc, const *memory*::*desc* &dst_iter_c_desc, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for an LSTM (with or without peephole) forward propagation primitive.

The following arguments may point to a zero memory descriptor:
- `src_iter_desc` together with `src_iter_c_desc`,
- `weights_peephole_desc`,
- `bias_desc`,
- `dst_iter_desc` together with `dst_iter_c_desc`.

This would then indicate that the LSTM forward propagation primitive should not use them and should default to zero values instead.

Note: All memory descriptors can be initialized with an *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters
- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- **direction** – RNN direction. See *dnnl::rnn_direction* for more info.
- **src_layer_desc** – Memory descriptor for the input vector.
- **src_iter_desc** – Memory descriptor for the input recurrent hidden state vector.
- **src_iter_c_desc** – Memory descriptor for the input recurrent cell state vector.
- **weights_layer_desc** – Memory descriptor for the weights applied to the layer input.
- **weights_iter_desc** – Memory descriptor for the weights applied to the recurrent input.
- **weights_peephole_desc** – Memory descriptor for the weights applied to the cell states (according to the Peephole LSTM formula).
- **bias_desc** – Bias memory descriptor.
- **dst_layer_desc** – Memory descriptor for the output vector.
- **dst_iter_desc** – Memory descriptor for the output recurrent hidden state vector.
- **dst_iter_c_desc** – Memory descriptor for the output recurrent cell state vector.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.

- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *rnn_direction* direction, const *memory*::*desc* &src_layer_desc, const *memory*::*desc* &src_iter_desc, const *memory*::*desc* &src_iter_c_desc, const *memory*::*desc* &weights_layer_desc, const *memory*::*desc* &weights_iter_desc, const *memory*::*desc* &bias_desc, const *memory*::*desc* &dst_layer_desc, const *memory*::*desc* &dst_iter_desc, const *memory*::*desc* &dst_iter_c_desc, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for an LSTM forward propagation primitive.

The following arguments may point to a zero memory descriptor:
- `src_iter_desc` together with `src_iter_c_desc`,
- `bias_desc`,
- `dst_iter_desc` together with `dst_iter_c_desc`.

This would then indicate that the LSTM forward propagation primitive should not use them and should default to zero values instead.

Note: All memory descriptors can be initialized with an *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters
- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- **direction** – RNN direction. See *dnnl::rnn_direction* for more info.
- **src_layer_desc** – Memory descriptor for the input vector.
- **src_iter_desc** – Memory descriptor for the input recurrent hidden state vector.
- **src_iter_c_desc** – Memory descriptor for the input recurrent cell state vector.
- **weights_layer_desc** – Memory descriptor for the weights applied to the layer input.
- **weights_iter_desc** – Memory descriptor for the weights applied to the recurrent input.
- **bias_desc** – Bias memory descriptor.
- **dst_layer_desc** – Memory descriptor for the output vector.
- **dst_iter_desc** – Memory descriptor for the output recurrent hidden state vector.
- **dst_iter_c_desc** – Memory descriptor for the output recurrent cell state vector.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_layer_desc**() const

Returns source layer memory descriptor.

Returns
Source layer memory descriptor.

memory::*desc* **src_iter_desc**() const

Returns source iteration memory descriptor.

Returns
Source iteration memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a source iteration parameter.

memory::*desc* **src_iter_c_desc**() const

Returns source iteration memory descriptor.

Returns

Source iteration memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a source iteration parameter.

memory::*desc* **weights_layer_desc**() const

Returns weights layer memory descriptor.

Returns

Weights layer memory descriptor.

memory::*desc* **weights_iter_desc**() const

Returns weights iteration memory descriptor.

Returns

Weights iteration memory descriptor.

memory::*desc* **weights_peephole_desc**() const

Returns weights peephole memory descriptor.

Returns

Weights peephole memory descriptor.

memory::*desc* **weights_projection_desc**() const

Returns weights projection memory descriptor.

Returns

Weights projection memory descriptor.

memory::*desc* **bias_desc**() const

Returns bias memory descriptor.

Returns

Bias memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a bias parameter.

memory::*desc* **dst_layer_desc**() const

Returns destination layer memory descriptor.

Returns

Destination layer memory descriptor.

memory::*desc* **dst_iter_desc**() const

Returns destination iteration memory descriptor.

Returns

Destination iteration memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a destination iteration parameter.

memory::*desc* **dst_iter_c_desc**() const

Returns source iteration memory descriptor.

Returns

Source iteration memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a source iteration parameter.

memory::*desc* **workspace_desc**() const

Returns the workspace memory descriptor.

Returns
Workspace memory descriptor.
Returns
A zero memory descriptor if the primitive does not require workspace parameter.

algorithm **get_cell_kind**() const

Returns an RNN cell kind parameter.
Returns
An RNN cell kind parameter.
Returns
dnnl::algorithm::undef if the primitive does not have an RNN cell kind parameter.

prop_kind **get_prop_kind**() const

Returns a propagation kind.
Returns
A propagation kind.
Returns
dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

rnn_direction **get_direction**() const

Returns an RNN direction parameter.
Returns
An RNN direction parameter.
Returns
dnnl::rnn_direction::undef if the primitive does not have an RNN direction parameter.

struct **lstm_backward** : public dnnl::*primitive*

LSTM backward propagation primitive.

Public Functions

lstm_backward()

Default constructor. Produces an empty object.

lstm_backward(const *primitive_desc* &pd)

Constructs an LSTM backward propagation primitive.

Parameters
pd – Primitive descriptor for an LSTM backward propagation primitive.

struct **primitive_desc** : public dnnl::*rnn_primitive_desc_base*

Primitive descriptor for an LSTM backward propagation primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *rnn_direction* direction, const *memory*::*desc* &src_layer_desc, const *memory*::*desc* &src_iter_desc, const *memory*::*desc* &src_iter_c_desc, const *memory*::*desc* &weights_layer_desc, const *memory*::*desc* &weights_iter_desc, const *memory*::*desc* &weights_peephole_desc, const *memory*::*desc* &weights_projection_desc, const *memory*::*desc* &bias_desc, const *memory*::*desc* &dst_layer_desc, const *memory*::*desc* &dst_iter_desc, const *memory*::*desc* &dst_iter_c_desc, const *memory*::*desc* &diff_src_layer_desc, const *memory*::*desc* &diff_src_iter_desc, const *memory*::*desc* &diff_src_iter_c_desc, const *memory*::*desc* &diff_weights_layer_desc, const *memory*::*desc* &diff_weights_iter_desc, const *memory*::*desc* &diff_weights_peephole_desc, const *memory*::*desc* &diff_weights_projection_desc, const *memory*::*desc* &diff_bias_desc, const *memory*::*desc* &diff_dst_layer_desc, const *memory*::*desc* &diff_dst_iter_desc, const *memory*::*desc* &diff_dst_iter_c_desc, const *lstm_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs an LSTM (with or without peephole and with or without projection) primitive descriptor for backward propagation using `prop_kind`, `direction`, and memory descriptors.

The following arguments may point to a zero memory descriptor:

- `src_iter_desc` together with `src_iter_c_desc`, `diff_src_iter_desc`, and `diff_src_iter_c_desc`,
- `weights_peephole_desc` together with `diff_weights_peephole_desc`
- `bias_desc` together with `diff_bias_desc`,
- `dst_iter_desc` together with `dst_iter_c_desc`, `diff_dst_iter_desc`, and `diff_dst_iter_c_desc`.

This would then indicate that the LSTM backward propagation primitive should not use them and should default to zero values instead.

The `weights_projection_desc` together with `diff_weights_projection_desc` may point to a zero memory descriptor. This would then indicate that the LSTM doesn't have recurrent projection layer.

Note: All memory descriptors can be initialized with *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Must be *dnnl::prop_kind::backward*.
- **direction** – RNN direction. See *dnnl::rnn_direction* for more info.
- **src_layer_desc** – Memory descriptor for the input vector.
- **src_iter_desc** – Memory descriptor for the input recurrent hidden state vector.
- **src_iter_c_desc** – Memory descriptor for the input recurrent cell state vector.
- **weights_layer_desc** – Memory descriptor for the weights applied to the layer input.
- **weights_iter_desc** – Memory descriptor for the weights applied to the recurrent input.
- **weights_peephole_desc** – Memory descriptor for the weights applied to the cell states (according to the Peephole LSTM formula).
- **weights_projection_desc** – Memory descriptor for the weights applied to the hidden states to get the recurrent projection (according to the Projection LSTM formula).
- **bias_desc** – Bias memory descriptor.
- **dst_layer_desc** – Memory descriptor for the output vector.
- **dst_iter_desc** – Memory descriptor for the output recurrent hidden state vector.
- **dst_iter_c_desc** – Memory descriptor for the output recurrent cell state vector.
- **diff_src_layer_desc** – Memory descriptor for the diff of input vector.

- **diff_src_iter_desc** – Memory descriptor for the diff of input recurrent hidden state vector.
- **diff_src_iter_c_desc** – Memory descriptor for the diff of input recurrent cell state vector.
- **diff_weights_layer_desc** – Memory descriptor for the diff of weights applied to the layer input.
- **diff_weights_iter_desc** – Memory descriptor for the diff of weights applied to the recurrent input.
- **diff_weights_peephole_desc** – Memory descriptor for the diff of weights applied to the cell states (according to the Peephole LSTM formula).
- **diff_weights_projection_desc** – Memory descriptor for the diff of weights applied to the hidden states to get the recurrent projection (according to the Projection LSTM formula).
- **diff_bias_desc** – Diff bias memory descriptor.
- **diff_dst_layer_desc** – Memory descriptor for the diff of output vector.
- **diff_dst_iter_desc** – Memory descriptor for the diff of output recurrent hidden state vector.
- **diff_dst_iter_c_desc** – Memory descriptor for the diff of output recurrent cell state vector.
- **hint_fwd_pd** – Primitive descriptor for an LSTM forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *rnn_direction* direction, const *memory*::*desc* &src_layer_desc, const *memory*::*desc* &src_iter_desc, const *memory*::*desc* &src_iter_c_desc, const *memory*::*desc* &weights_layer_desc, const *memory*::*desc* &weights_iter_desc, const *memory*::*desc* &weights_peephole_desc, const *memory*::*desc* &bias_desc, const *memory*::*desc* &dst_layer_desc, const *memory*::*desc* &dst_iter_desc, const *memory*::*desc* &dst_iter_c_desc, const *memory*::*desc* &diff_src_layer_desc, const *memory*::*desc* &diff_src_iter_desc, const *memory*::*desc* &diff_src_iter_c_desc, const *memory*::*desc* &diff_weights_layer_desc, const *memory*::*desc* &diff_weights_iter_desc, const *memory*::*desc* &diff_weights_peephole_desc, const *memory*::*desc* &diff_bias_desc, const *memory*::*desc* &diff_dst_layer_desc, const *memory*::*desc* &diff_dst_iter_desc, const *memory*::*desc* &diff_dst_iter_c_desc, const *lstm_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs an LSTM (with or without peephole) primitive descriptor for backward propagation using `prop_kind`, `direction`, and memory descriptors.

The following arguments may point to a zero memory descriptor:

- `src_iter_desc` together with `src_iter_c_desc`, `diff_src_iter_desc`, and `diff_src_iter_c_desc`,
- `weights_peephole_desc` together with `diff_weights_peephole_desc`
- `bias_desc` together with `diff_bias_desc`,
- `dst_iter_desc` together with `dst_iter_c_desc`, `diff_dst_iter_desc`, and `diff_dst_iter_c_desc`.

This would then indicate that the LSTM backward propagation primitive should not use them and should default to zero values instead.

Note: All memory descriptors may be initialized with *dnnl::memory::format_tag::any* value of

format_tag.

Parameters

- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Must be *dnnl::prop_kind::backward*.
- **direction** – RNN direction. See *dnnl::rnn_direction* for more info.
- **src_layer_desc** – Memory descriptor for the input vector.
- **src_iter_desc** – Memory descriptor for the input recurrent hidden state vector.
- **src_iter_c_desc** – Memory descriptor for the input recurrent cell state vector.
- **weights_layer_desc** – Memory descriptor for the weights applied to the layer input.
- **weights_iter_desc** – Memory descriptor for the weights applied to the recurrent input.
- **weights_peephole_desc** – Memory descriptor for the weights applied to the cell states (according to the Peephole LSTM formula).
- **bias_desc** – Bias memory descriptor.
- **dst_layer_desc** – Memory descriptor for the output vector.
- **dst_iter_desc** – Memory descriptor for the output recurrent hidden state vector.
- **dst_iter_c_desc** – Memory descriptor for the output recurrent cell state vector.
- **diff_src_layer_desc** – Memory descriptor for the diff of input vector.
- **diff_src_iter_desc** – Memory descriptor for the diff of input recurrent hidden state vector.
- **diff_src_iter_c_desc** – Memory descriptor for the diff of input recurrent cell state vector.
- **diff_weights_layer_desc** – Memory descriptor for the diff of weights applied to the layer input.
- **diff_weights_iter_desc** – Memory descriptor for the diff of weights applied to the recurrent input.
- **diff_weights_peephole_desc** – Memory descriptor for the diff of weights applied to the cell states (according to the Peephole LSTM formula).
- **diff_bias_desc** – Diff bias memory descriptor.
- **diff_dst_layer_desc** – Memory descriptor for the diff of output vector.
- **diff_dst_iter_desc** – Memory descriptor for the diff of output recurrent hidden state vector.
- **diff_dst_iter_c_desc** – Memory descriptor for the diff of output recurrent cell state vector.
- **hint_fwd_pd** – Primitive descriptor for an LSTM forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *rnn_direction* direction, const *memory*::*desc* &src_layer_desc, const *memory*::*desc* &src_iter_desc, const *memory*::*desc* &src_iter_c_desc, const *memory*::*desc* &weights_layer_desc, const *memory*::*desc* &weights_iter_desc, const *memory*::*desc* &bias_desc, const *memory*::*desc* &dst_layer_desc, const *memory*::*desc* &dst_iter_desc, const *memory*::*desc* &dst_iter_c_desc, const *memory*::*desc* &diff_src_layer_desc, const *memory*::*desc* &diff_src_iter_desc, const *memory*::*desc* &diff_src_iter_c_desc, const *memory*::*desc* &diff_weights_layer_desc, const *memory*::*desc* &diff_weights_iter_desc, const *memory*::*desc* &diff_bias_desc, const *memory*::*desc* &diff_dst_layer_desc, const *memory*::*desc* &diff_dst_iter_desc, const *memory*::*desc* &diff_dst_iter_c_desc, const *lstm_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs an LSTM primitive descriptor for backward propagation using `prop_kind`, `direction`, and memory descriptors.

The following arguments may point to a zero memory descriptor:

- `src_iter_desc` together with `src_iter_c_desc`, `diff_src_iter_desc`, and `diff_src_iter_c_desc`,
- `bias_desc` together with `diff_bias_desc`,
- `dst_iter_desc` together with `dst_iter_c_desc`, `diff_dst_iter_desc`, and `diff_dst_iter_c_desc`.

This would then indicate that the LSTM backward propagation primitive should not use them and should default to zero values instead.

Note: All memory descriptors may be initialized with *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Must be *dnnl::prop_kind::backward*.
- **direction** – RNN direction. See *dnnl::rnn_direction* for more info.
- **src_layer_desc** – Memory descriptor for the input vector.
- **src_iter_desc** – Memory descriptor for the input recurrent hidden state vector.
- **src_iter_c_desc** – Memory descriptor for the input recurrent cell state vector.
- **weights_layer_desc** – Memory descriptor for the weights applied to the layer input.
- **weights_iter_desc** – Memory descriptor for the weights applied to the recurrent input.
- **bias_desc** – Bias memory descriptor.
- **dst_layer_desc** – Memory descriptor for the output vector.
- **dst_iter_desc** – Memory descriptor for the output recurrent hidden state vector.
- **dst_iter_c_desc** – Memory descriptor for the output recurrent cell state vector.
- **diff_src_layer_desc** – Memory descriptor for the diff of input vector.
- **diff_src_iter_desc** – Memory descriptor for the diff of input recurrent hidden state vector.
- **diff_src_iter_c_desc** – Memory descriptor for the diff of input recurrent cell state vector.
- **diff_weights_layer_desc** – Memory descriptor for the diff of weights applied to the layer input.
- **diff_weights_iter_desc** – Memory descriptor for the diff of weights applied to the recurrent input.
- **diff_bias_desc** – Diff bias memory descriptor.
- **diff_dst_layer_desc** – Memory descriptor for the diff of output vector.
- **diff_dst_iter_desc** – Memory descriptor for the diff of output recurrent hidden state vector.
- **diff_dst_iter_c_desc** – Memory descriptor for the diff of output recurrent cell state vector.
- **hint_fwd_pd** – Primitive descriptor for a convolution forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_layer_desc**() const

Returns source layer memory descriptor.

Returns
Source layer memory descriptor.

memory::*desc* **src_iter_desc**() const

Returns source iteration memory descriptor.

Returns
Source iteration memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a source iteration parameter.

memory::*desc* **src_iter_c_desc**() const

Returns source iteration memory descriptor.

Returns
Source iteration memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a source iteration parameter.

memory::*desc* **weights_layer_desc**() const

Returns weights layer memory descriptor.

Returns
Weights layer memory descriptor.

memory::*desc* **weights_iter_desc**() const

Returns weights iteration memory descriptor.

Returns
Weights iteration memory descriptor.

memory::*desc* **weights_peephole_desc**() const

Returns weights peephole memory descriptor.

Returns
Weights peephole memory descriptor.

memory::*desc* **weights_projection_desc**() const

Returns weights projection memory descriptor.

Returns
Weights projection memory descriptor.

memory::*desc* **bias_desc**() const

Returns bias memory descriptor.

Returns
Bias memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a bias parameter.

memory::*desc* **dst_layer_desc**() const

Returns destination layer memory descriptor.

Returns
Destination layer memory descriptor.

memory::*desc* **dst_iter_desc**() const

Returns destination iteration memory descriptor.

Returns
Destination iteration memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a destination iteration parameter.

memory::*desc* **dst_iter_c_desc**() const

Returns source iteration memory descriptor.

Returns
Source iteration memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a source iteration parameter.

memory::*desc* **workspace_desc**() const

Returns the workspace memory descriptor.

Returns
Workspace memory descriptor.

Returns
A zero memory descriptor if the primitive does not require workspace parameter.

memory::*desc* **diff_src_layer_desc**() const

Returns diff source layer memory descriptor.

Returns
Diff source layer memory descriptor.

memory::*desc* **diff_src_iter_desc**() const

Returns diff source iteration memory descriptor.

Returns
Diff source iteration memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a diff source iteration parameter.

memory::*desc* **diff_src_iter_c_desc**() const

Returns diff source recurrent cell state memory descriptor.

Returns
Diff source recurrent cell state memory descriptor.

memory::*desc* **diff_weights_layer_desc**() const

Returns diff weights layer memory descriptor.

Returns
Diff weights layer memory descriptor.

memory::*desc* **diff_weights_iter_desc**() const

Returns diff weights iteration memory descriptor.

Returns
Diff weights iteration memory descriptor.

memory::*desc* **diff_weights_peephole_desc**() const

Returns diff weights peephole memory descriptor.

Returns
Diff weights peephole memory descriptor.

memory::*desc* **diff_weights_projection_desc**() const

Returns diff weights projection memory descriptor.

Returns
Diff weights projection memory descriptor.

memory::*desc* **diff_bias_desc**() const

Returns diff bias memory descriptor.

Returns
Diff bias memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a diff bias parameter.

memory::*desc* **diff_dst_layer_desc**() const

Returns diff destination layer memory descriptor.

Returns
Diff destination layer memory descriptor.

memory::*desc* **diff_dst_iter_desc**() const

Returns diff destination iteration memory descriptor.

Returns
Diff destination iteration memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a diff destination iteration parameter.

memory::*desc* **diff_dst_iter_c_desc**() const

Returns diff destination recurrent cell state memory descriptor.

Returns
Diff destination recurrent cell state memory descriptor.

algorithm **get_cell_kind**() const

Returns an RNN cell kind parameter.

Returns
An RNN cell kind parameter.

Returns
dnnl::algorithm::undef if the primitive does not have an RNN cell kind parameter.

prop_kind **get_prop_kind**() const

Returns a propagation kind.

Returns
A propagation kind.

Returns
dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

rnn_direction **get_direction**() const

Returns an RNN direction parameter.

Returns
An RNN direction parameter.

Returns
dnnl::rnn_direction::undef if the primitive does not have an RNN direction parameter.

struct **gru_forward** : public dnnl::*primitive*

GRU forward propagation primitive.

Public Functions

gru_forward()

Default constructor. Produces an empty object.

gru_forward(const *primitive_desc* &pd)

Constructs a GRU forward propagation primitive.

Parameters

pd – Primitive descriptor for a GRU forward propagation primitive.

struct **primitive_desc** : public dnnl::*rnn_primitive_desc_base*

Primitive descriptor for a GRU forward propagation primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *rnn_direction* direction, const *memory*::*desc* &src_layer_desc, const *memory*::*desc* &src_iter_desc, const *memory*::*desc* &weights_layer_desc, const *memory*::*desc* &weights_iter_desc, const *memory*::*desc* &bias_desc, const *memory*::*desc* &dst_layer_desc, const *memory*::*desc* &dst_iter_desc, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a GRU forward propagation primitive.

The following arguments may point to a zero memory descriptor:
- `src_iter_desc`,
- `bias_desc`,
- `dst_iter_desc`.

This would then indicate that the GRU forward propagation primitive should not use them and should default to zero values instead.

Note: All memory descriptors except `src_iter_desc` may be initialized with an *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- **direction** – RNN direction. See *dnnl::rnn_direction* for more info.
- **src_layer_desc** – Memory descriptor for the input vector.
- **src_iter_desc** – Memory descriptor for the input recurrent hidden state vector.
- **weights_layer_desc** – Memory descriptor for the weights applied to the layer input.
- **weights_iter_desc** – Memory descriptor for the weights applied to the recurrent input.
- **bias_desc** – Bias memory descriptor.
- **dst_layer_desc** – Memory descriptor for the output vector.
- **dst_iter_desc** – Memory descriptor for the output recurrent hidden state vector.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_layer_desc**() const

Returns source layer memory descriptor.

Returns

Source layer memory descriptor.

memory::*desc* **src_iter_desc**() const

Returns source iteration memory descriptor.

Returns

Source iteration memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a source iteration parameter.

memory::*desc* **weights_layer_desc**() const

Returns weights layer memory descriptor.

Returns

Weights layer memory descriptor.

memory::*desc* **weights_iter_desc**() const

Returns weights iteration memory descriptor.

Returns

Weights iteration memory descriptor.

memory::*desc* **bias_desc**() const

Returns bias memory descriptor.

Returns

Bias memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a bias parameter.

memory::*desc* **dst_layer_desc**() const

Returns destination layer memory descriptor.

Returns

Destination layer memory descriptor.

memory::*desc* **dst_iter_desc**() const

Returns destination iteration memory descriptor.

Returns

Destination iteration memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a destination iteration parameter.

memory::*desc* **workspace_desc**() const

Returns the workspace memory descriptor.

Returns

Workspace memory descriptor.

Returns

A zero memory descriptor if the primitive does not require workspace parameter.

algorithm **get_cell_kind**() const

Returns an RNN cell kind parameter.

Returns

An RNN cell kind parameter.

Returns

dnnl::algorithm::undef if the primitive does not have an RNN cell kind parameter.

prop_kind **get_prop_kind**() const

Returns a propagation kind.

Returns
A propagation kind.

Returns
dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

rnn_direction **get_direction**() const

Returns an RNN direction parameter.

Returns
An RNN direction parameter.

Returns
dnnl::rnn_direction::undef if the primitive does not have an RNN direction parameter.

struct **gru_backward** : public dnnl::*primitive*

GRU backward propagation primitive.

Public Functions

gru_backward()

Default constructor. Produces an empty object.

gru_backward(const *primitive_desc* &pd)

Constructs a GRU backward propagation primitive.

Parameters
pd – Primitive descriptor for a GRU backward propagation primitive.

struct **primitive_desc** : public dnnl::*rnn_primitive_desc_base*

Primitive descriptor for a GRU backward propagation primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *rnn_direction* direction, const *memory*::*desc* &src_layer_desc, const *memory*::*desc* &src_iter_desc, const *memory*::*desc* &weights_layer_desc, const *memory*::*desc* &weights_iter_desc, const *memory*::*desc* &bias_desc, const *memory*::*desc* &dst_layer_desc, const *memory*::*desc* &dst_iter_desc, const *memory*::*desc* &diff_src_layer_desc, const *memory*::*desc* &diff_src_iter_desc, const *memory*::*desc* &diff_weights_layer_desc, const *memory*::*desc* &diff_weights_iter_desc, const *memory*::*desc* &diff_bias_desc, const *memory*::*desc* &diff_dst_layer_desc, const *memory*::*desc* &diff_dst_iter_desc, const *gru_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a GRU backward propagation primitive.

The following arguments may point to a zero memory descriptor:
- `src_iter_desc` together with `diff_src_iter_desc`,
- `bias_desc` together with `diff_bias_desc`,
- `dst_iter_desc` together with `diff_dst_iter_desc`.

This would then indicate that the GRU backward propagation primitive should not use them and should default to zero values instead.

Note: All memory descriptors may be initialized with *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Must be *dnnl::prop_kind::backward*.
- **direction** – RNN direction. See *dnnl::rnn_direction* for more info.
- **src_layer_desc** – Memory descriptor for the input vector.
- **src_iter_desc** – Memory descriptor for the input recurrent hidden state vector.
- **weights_layer_desc** – Memory descriptor for the weights applied to the layer input.
- **weights_iter_desc** – Memory descriptor for the weights applied to the recurrent input.
- **bias_desc** – Bias memory descriptor.
- **dst_layer_desc** – Memory descriptor for the output vector.
- **dst_iter_desc** – Memory descriptor for the output recurrent hidden state vector.
- **diff_src_layer_desc** – Memory descriptor for the diff of input vector.
- **diff_src_iter_desc** – Memory descriptor for the diff of input recurrent hidden state vector.
- **diff_weights_layer_desc** – Memory descriptor for the diff of weights applied to the layer input.
- **diff_weights_iter_desc** – Memory descriptor for the diff of weights applied to the recurrent input.
- **diff_bias_desc** – Diff bias memory descriptor.
- **diff_dst_layer_desc** – Memory descriptor for the diff of output vector.
- **diff_dst_iter_desc** – Memory descriptor for the diff of output recurrent hidden state vector.
- **hint_fwd_pd** – Primitive descriptor for a GRU forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_layer_desc**() const

Returns source layer memory descriptor.

Returns

Source layer memory descriptor.

memory::*desc* **src_iter_desc**() const

Returns source iteration memory descriptor.

Returns

Source iteration memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a source iteration parameter.

memory::*desc* **weights_layer_desc**() const

Returns weights layer memory descriptor.

Returns

Weights layer memory descriptor.

memory::*desc* **weights_iter_desc**() const

Returns weights iteration memory descriptor.

Returns

Weights iteration memory descriptor.

memory::*desc* **bias_desc**() const

Returns bias memory descriptor.

Returns

Bias memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a bias parameter.

memory::*desc* **dst_layer_desc**() const

Returns destination layer memory descriptor.

Returns

Destination layer memory descriptor.

memory::*desc* **dst_iter_desc**() const

Returns destination iteration memory descriptor.

Returns

Destination iteration memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a destination iteration parameter.

memory::*desc* **workspace_desc**() const

Returns the workspace memory descriptor.

Returns

Workspace memory descriptor.

Returns

A zero memory descriptor if the primitive does not require workspace parameter.

memory::*desc* **diff_src_layer_desc**() const

Returns diff source layer memory descriptor.

Returns

Diff source layer memory descriptor.

memory::*desc* **diff_src_iter_desc**() const

Returns diff source iteration memory descriptor.

Returns

Diff source iteration memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a diff source iteration parameter.

memory::*desc* **diff_weights_layer_desc**() const

Returns diff weights layer memory descriptor.

Returns

Diff weights layer memory descriptor.

memory::*desc* **diff_weights_iter_desc**() const

Returns diff weights iteration memory descriptor.

Returns

Diff weights iteration memory descriptor.

memory::*desc* **diff_bias_desc**() const

Returns diff bias memory descriptor.

Returns
Diff bias memory descriptor.
Returns
A zero memory descriptor if the primitive does not have a diff bias parameter.

memory::*desc* **diff_dst_layer_desc**() const
Returns diff destination layer memory descriptor.
Returns
Diff destination layer memory descriptor.

memory::*desc* **diff_dst_iter_desc**() const
Returns diff destination iteration memory descriptor.
Returns
Diff destination iteration memory descriptor.
Returns
A zero memory descriptor if the primitive does not have a diff destination iteration parameter.

algorithm **get_cell_kind**() const
Returns an RNN cell kind parameter.
Returns
An RNN cell kind parameter.
Returns
dnnl::algorithm::undef if the primitive does not have an RNN cell kind parameter.

prop_kind **get_prop_kind**() const
Returns a propagation kind.
Returns
A propagation kind.
Returns
dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

rnn_direction **get_direction**() const
Returns an RNN direction parameter.
Returns
An RNN direction parameter.
Returns
dnnl::rnn_direction::undef if the primitive does not have an RNN direction parameter.

struct **lbr_gru_forward** : public dnnl::*primitive*
LBR GRU forward propagation primitive.

Public Functions

lbr_gru_forward()
Default constructor. Produces an empty object.

lbr_gru_forward(const *primitive_desc* &pd)
Constructs an LBR GRU forward propagation primitive.

Parameters
pd – Primitive descriptor for an LBR GRU forward propagation primitive.

struct **primitive_desc** : public dnnl::*rnn_primitive_desc_base*
Primitive descriptor for an LBR GRU forward propagation primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *rnn_direction* direction, const *memory*::*desc* &src_layer_desc, const *memory*::*desc* &src_iter_desc, const *memory*::*desc* &weights_layer_desc, const *memory*::*desc* &weights_iter_desc, const *memory*::*desc* &bias_desc, const *memory*::*desc* &dst_layer_desc, const *memory*::*desc* &dst_iter_desc, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for LBR GRU forward propagation primitive.

The following arguments may point to a zero memory descriptor:
- `src_iter_desc`,
- `bias_desc`,
- `dst_iter_desc`.

This would then indicate that the LBR GRU forward propagation primitive should not use them and should default to zero values instead.

Note: All memory descriptors except `src_iter_desc` may be initialized with an *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters
- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- **direction** – RNN direction. See *dnnl::rnn_direction* for more info.
- **src_layer_desc** – Memory descriptor for the input vector.
- **src_iter_desc** – Memory descriptor for the input recurrent hidden state vector.
- **weights_layer_desc** – Memory descriptor for the weights applied to the layer input.
- **weights_iter_desc** – Memory descriptor for the weights applied to the recurrent input.
- **bias_desc** – Bias memory descriptor.
- **dst_layer_desc** – Memory descriptor for the output vector.
- **dst_iter_desc** – Memory descriptor for the output recurrent hidden state vector.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_layer_desc**() const

Returns source layer memory descriptor.

Returns
Source layer memory descriptor.

memory::*desc* **src_iter_desc**() const

Returns source iteration memory descriptor.

Returns
Source iteration memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a source iteration parameter.

memory::*desc* **weights_layer_desc**() const

Returns weights layer memory descriptor.

Returns
Weights layer memory descriptor.

memory::*desc* **weights_iter_desc**() const

Returns weights iteration memory descriptor.

Returns
Weights iteration memory descriptor.

memory::*desc* **bias_desc**() const

Returns bias memory descriptor.

Returns
Bias memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a bias parameter.

memory::*desc* **dst_layer_desc**() const

Returns destination layer memory descriptor.

Returns
Destination layer memory descriptor.

memory::*desc* **dst_iter_desc**() const

Returns destination iteration memory descriptor.

Returns
Destination iteration memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a destination iteration parameter.

memory::*desc* **workspace_desc**() const

Returns the workspace memory descriptor.

Returns
Workspace memory descriptor.

Returns
A zero memory descriptor if the primitive does not require workspace parameter.

algorithm **get_cell_kind**() const

Returns an RNN cell kind parameter.

Returns
An RNN cell kind parameter.

Returns
dnnl::algorithm::undef if the primitive does not have an RNN cell kind parameter.

prop_kind **get_prop_kind**() const

Returns a propagation kind.

Returns
A propagation kind.

Returns
dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

rnn_direction **get_direction**() const

Returns an RNN direction parameter.

Returns
An RNN direction parameter.

Returns
dnnl::rnn_direction::undef if the primitive does not have an RNN direction parameter.

struct **lbr_gru_backward** : public dnnl::*primitive*

LBR GRU backward propagation primitive.

Public Functions

lbr_gru_backward()

Default constructor. Produces an empty object.

lbr_gru_backward(const *primitive_desc* &pd)

Constructs an LBR GRU backward propagation primitive.

Parameters
pd – Primitive descriptor for an LBR GRU backward propagation primitive.

struct **primitive_desc** : public dnnl::*rnn_primitive_desc_base*

Primitive descriptor for an LBR GRU backward propagation primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *rnn_direction* direction, const *memory*::*desc* &src_layer_desc, const *memory*::*desc* &src_iter_desc, const *memory*::*desc* &weights_layer_desc, const *memory*::*desc* &weights_iter_desc, const *memory*::*desc* &bias_desc, const *memory*::*desc* &dst_layer_desc, const *memory*::*desc* &dst_iter_desc, const *memory*::*desc* &diff_src_layer_desc, const *memory*::*desc* &diff_src_iter_desc, const *memory*::*desc* &diff_weights_layer_desc, const *memory*::*desc* &diff_weights_iter_desc, const *memory*::*desc* &diff_bias_desc, const *memory*::*desc* &diff_dst_layer_desc, const *memory*::*desc* &diff_dst_iter_desc, const *gru_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for LBR GRU backward propagation primitive.

The following arguments may point to a zero memory descriptor:
- `src_iter_desc` together with `diff_src_iter_desc`,
- `bias_desc` together with `diff_bias_desc`,
- `dst_iter_desc` together with `diff_dst_iter_desc`.

This would then indicate that the LBR GRU backward propagation primitive should not use them and should default to zero values instead.

Note: All memory descriptors may be initialized with *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters
- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Must be *dnnl::prop_kind::backward*.
- **direction** – RNN direction. See *dnnl::rnn_direction* for more info.
- **src_layer_desc** – Memory descriptor for the input vector.
- **src_iter_desc** – Memory descriptor for the input recurrent hidden state vector.
- **weights_layer_desc** – Memory descriptor for the weights applied to the layer input.

- **weights_iter_desc** – Memory descriptor for the weights applied to the recurrent input.
- **bias_desc** – Bias memory descriptor.
- **dst_layer_desc** – Memory descriptor for the output vector.
- **dst_iter_desc** – Memory descriptor for the output recurrent hidden state vector.
- **diff_src_layer_desc** – Memory descriptor for the diff of input vector.
- **diff_src_iter_desc** – Memory descriptor for the diff of input recurrent hidden state vector.
- **diff_weights_layer_desc** – Memory descriptor for the diff of weights applied to the layer input.
- **diff_weights_iter_desc** – Memory descriptor for the diff of weights applied to the recurrent input.
- **diff_bias_desc** – Diff bias memory descriptor.
- **diff_dst_layer_desc** – Memory descriptor for the diff of output vector.
- **diff_dst_iter_desc** – Memory descriptor for the diff of output recurrent hidden state vector.
- **hint_fwd_pd** – Primitive descriptor for an LBR GRU forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_layer_desc**() const

Returns source layer memory descriptor.

Returns

Source layer memory descriptor.

memory::*desc* **src_iter_desc**() const

Returns source iteration memory descriptor.

Returns

Source iteration memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a source iteration parameter.

memory::*desc* **weights_layer_desc**() const

Returns weights layer memory descriptor.

Returns

Weights layer memory descriptor.

memory::*desc* **weights_iter_desc**() const

Returns weights iteration memory descriptor.

Returns

Weights iteration memory descriptor.

memory::*desc* **bias_desc**() const

Returns bias memory descriptor.

Returns

Bias memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a bias parameter.

memory::*desc* **dst_layer_desc**() const

Returns destination layer memory descriptor.

Returns

Destination layer memory descriptor.

memory::*desc* **dst_iter_desc**() const

Returns destination iteration memory descriptor.

Returns

Destination iteration memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a destination iteration parameter.

memory::*desc* **workspace_desc**() const

Returns the workspace memory descriptor.

Returns

Workspace memory descriptor.

Returns

A zero memory descriptor if the primitive does not require workspace parameter.

memory::*desc* **diff_src_layer_desc**() const

Returns diff source layer memory descriptor.

Returns

Diff source layer memory descriptor.

memory::*desc* **diff_src_iter_desc**() const

Returns diff source iteration memory descriptor.

Returns

Diff source iteration memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a diff source iteration parameter.

memory::*desc* **diff_weights_layer_desc**() const

Returns diff weights layer memory descriptor.

Returns

Diff weights layer memory descriptor.

memory::*desc* **diff_weights_iter_desc**() const

Returns diff weights iteration memory descriptor.

Returns

Diff weights iteration memory descriptor.

memory::*desc* **diff_bias_desc**() const

Returns diff bias memory descriptor.

Returns

Diff bias memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a diff bias parameter.

memory::*desc* **diff_dst_layer_desc**() const

Returns diff destination layer memory descriptor.

Returns

Diff destination layer memory descriptor.

memory::*desc* **diff_dst_iter_desc**() const

Returns diff destination iteration memory descriptor.

Returns

Diff destination iteration memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a diff destination iteration parameter.

algorithm **get_cell_kind**() const

Returns an RNN cell kind parameter.

Returns

An RNN cell kind parameter.

Returns

dnnl::algorithm::undef if the primitive does not have an RNN cell kind parameter.

prop_kind **get_prop_kind**() const

Returns a propagation kind.

Returns

A propagation kind.

Returns

dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

rnn_direction **get_direction**() const

Returns an RNN direction parameter.

Returns

An RNN direction parameter.

Returns

dnnl::rnn_direction::undef if the primitive does not have an RNN direction parameter.

4.5.18 Shuffle

The shuffle primitive shuffles data along the shuffle axis (here is designated as C) with the group parameter G. Namely, the shuffle axis is thought to be a 2D tensor of size $(\frac{C}{G} \times G)$ and it is being transposed to $(G \times \frac{C}{G})$. Variable names follow the standard *Conventions*.

The formal definition is shown below:

Forward

$$\mathrm{dst}(\overline{ou}, c, \overline{in}) = \mathrm{src}(\overline{ou}, c', \overline{in})$$

where

- c dimension is called a shuffle `axis`,
- G is a `group_size`,
- $\overline{ou}$ is the outermost indices (to the left from shuffle axis),
- $\overline{in}$ is the innermost indices (to the right from shuffle axis), and
- c' and c relate to each other as define by the system:

$$\begin{cases} c &= u + v \cdot \frac{C}{G}, \\ c' &= u \cdot G + v, \end{cases}$$

Here, $0 \leq u < \frac{C}{G}$ and $0 \leq v < G$.

Difference Between Forward Training and Forward Inference

There is no difference between the *forward_training* and *forward_inference* propagation kinds.

Backward

The backward propagation computes $\mathrm{diff\_src}(ou, c, in)$, based on $\mathrm{diff\_dst}(ou, c, in)$.

Essentially, backward propagation is the same as forward propagation with g replaced by C/g.

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	*DNNL_ARG_SRC*
dst	*DNNL_ARG_DST*
diff_src	*DNNL_ARG_DIFF_SRC*
diff_dst	*DNNL_ARG_DIFF_DST*

Operation Details

Data Types Support

The shuffle primitive supports the following combinations of data types:

Note: Here we abbreviate data types names for readability. For example, *dnnl::memory::data_type::f32* is abbreviated to *f32*.

Propagation	Source / Destination
forward / backward	*f32*, *bf16*
forward	*s32*, *s8*, *u8*

Data Layouts

The shuffle primitive works with arbitrary data tensors. There is no special meaning associated with any logical dimensions. However, the shuffle axis is typically referred to as channels (hence in formulas we use c).

Shuffle operation typically appear in CNN topologies. Hence, in the library the shuffle primitive is optimized for the corresponding memory formats:

Spatial	Logical tensor	Shuffle Axis	Implementations optimized for memory formats
2D	NCHW	1 (C)	*nchw* (*abcd*), *nhwc* (*acdb*), *optimized^*
3D	NCDHW	1 (C)	*ncdhw* (*abcde*), *ndhwc* (*acdeb*), *optimized^*

Here *optimized^* means the format that comes out of any preceding compute-intensive primitive.

Post-ops and Attributes

The shuffle primitive does not have to support any post-ops or attributes.

API

struct **shuffle_forward** : public dnnl::*primitive*

Shuffle forward propagation primitive.

Public Functions

shuffle_forward()

Default constructor. Produces an empty object.

shuffle_forward(const *primitive_desc* &pd)

Constructs a shuffle forward propagation primitive.

Parameters
pd – Primitive descriptor for a shuffle forward propagation primitive.

struct **primitive_desc** : public dnnl::*primitive_desc*

Primitive descriptor for a shuffle forward propagation primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, const *memory*::*desc* &src_desc, const *memory*::*desc* &dst_desc, int axis, int group_size, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a shuffle forward propagation primitive.

Parameters

- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- **src_desc** – Source memory descriptor.
- **dst_desc** – Destination memory descriptor.
- **axis** – The axis along which the data is shuffled.
- **group_size** – Shuffle group size.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() const

Returns a source memory descriptor.

Returns
Source memory descriptor.
Returns
A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **dst_desc**() const

Returns a destination memory descriptor.
Returns
Destination memory descriptor.
Returns
A zero memory descriptor if the primitive does not have a destination parameter.

prop_kind **get_prop_kind**() const

Returns a propagation kind.
Returns
A propagation kind.
Returns
dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

int **get_axis**() const

Returns an axis.
Returns
An axis.
Returns
A negative number if the primitive does not have an axis parameter.

memory::*dim* **get_group_size**() const

Returns a shuffle group size parameter.
Returns
A shuffle group size parameter.
Returns
Zero if the primitive does not have a shuffle group size parameter.

struct **shuffle_backward** : public dnnl::*primitive*

Shuffle backward propagation primitive.

Public Functions

shuffle_backward()

Default constructor. Produces an empty object.

shuffle_backward(const *primitive_desc* &pd)

Constructs a shuffle backward propagation primitive.

Parameters
pd – Primitive descriptor for a shuffle backward propagation primitive.

struct **primitive_desc** : public dnnl::*primitive_desc*

Primitive descriptor for a shuffle backward propagation primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, const *memory*::*desc* &diff_src_desc, const *memory*::*desc* &diff_dst_desc, int axis, int group_size, const *shuffle_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a shuffle backward propagation primitive.

Parameters

- **aengine** – Engine to use.
- **diff_src_desc** – Diff source memory descriptor.
- **diff_dst_desc** – Diff destination memory descriptor.
- **axis** – The axis along which the data is shuffled.
- **group_size** – Shuffle group size.
- **hint_fwd_pd** – Primitive descriptor for a shuffle forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **diff_src_desc**() const

Returns a diff source memory descriptor.

Returns

Diff source memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a diff source memory with.

memory::*desc* **diff_dst_desc**() const

Returns a diff destination memory descriptor.

Returns

Diff destination memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a diff destination parameter.

prop_kind **get_prop_kind**() const

Returns a propagation kind.

Returns

A propagation kind.

Returns

dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

int **get_axis**() const

Returns an axis.

Returns

An axis.

Returns

A negative number if the primitive does not have an axis parameter.

memory::*dim* **get_group_size**() const

Returns a shuffle group size parameter.

Returns

A shuffle group size parameter.

Returns
Zero if the primitive does not have a shuffle group size parameter.

4.5.19 Softmax

The softmax primitive performs softmax along a particular axis on data with arbitrary dimensions. All other axes are treated as independent (batch).

In general form, the operation is defined by the following formulas. The variable names follow the standard *Conventions*.

Forward

When the specified algorithm is softmax:

$$\mathrm{dst}(\overline{ou}, c, \overline{in}) = \frac{e^{\mathrm{src}(\overline{ou}, c, \overline{in}) - \nu(\overline{ou}, \overline{in})}}{\sum\limits_{ic} e^{\mathrm{src}(\overline{ou}, ic, \overline{in}) - \nu(\overline{ou}, \overline{in})}}.$$

When the specified algorithm is logsoftmax, the following numerically stable formula is used:

$$\mathrm{dst}(\overline{ou}, c, \overline{in}) = \ln\left(\frac{e^{\mathrm{src}(\overline{ou}, c, \overline{in}) - \nu(\overline{ou}, \overline{in})}}{\sum\limits_{ic} e^{\mathrm{src}(\overline{ou}, ic, \overline{in}) - \nu(\overline{ou}, \overline{in})}}\right) = \left(\mathrm{src}(\overline{ou}, c, \overline{in}) - \nu(\overline{ou}, \overline{in})\right) - \ln\left(\sum_{ic} e^{\mathrm{src}(\overline{ou}, ic, \overline{in}) - \nu(\overline{ou}, \overline{in})}\right)$$

where

- c axis over which the softmax computation is computed on,
- $\overline{ou}$ is the outermost index (to the left of softmax axis),
- $\overline{in}$ is the innermost index (to the right of softmax axis), and
- ν is used to produce more accurate results and defined as:

$$\nu(\overline{ou}, \overline{in}) = \max_{ic} \mathrm{src}(\overline{ou}, ic, \overline{in})$$

Difference Between Forward Training and Forward Inference

There is no difference between the *forward_training* and *forward_inference* propagation kinds.

Backward

The backward propagation computes $\mathrm{diff\_src}(ou, c, in)$, based on $\mathrm{diff\_dst}(ou, c, in)$ and $\mathrm{dst}(ou, c, in)$.

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	*DNNL_ARG_SRC*
dst	*DNNL_ARG_DST*
diff_src	*DNNL_ARG_DIFF_SRC*
diff_dst	*DNNL_ARG_DIFF_DST*

Operation Details

1. Both forward and backward propagation support in-place operations, meaning that `src` can be used as input and output for forward propagation, and `diff_dst` can be used as input and output for backward propagation. In case of in-place operation, the original data will be overwritten.

Post-ops and Attributes

The softmax primitive does not have to support any post-ops or attributes.

Data Types Support

The softmax primitive supports the following combinations of data types.

Note: Here we abbreviate data types names for readability. For example, *dnnl::memory::data_type::f32* is abbreviated to *f32*.

Propagation	Source / Destination
forward / backward	*bf16*, *f32*
forward	*f16*

Data Representation

Source, Destination, and Their Gradients

The softmax primitive works with arbitrary data tensors. There is no special meaning associated with any logical dimensions. However, the softmax axis is typically referred to as channels (hence in formulas we use c).

API

struct **softmax_forward** : public dnnl::*primitive*

Softmax forward propagation primitive.

Public Functions

softmax_forward()

Default constructor. Produces an empty object.

softmax_forward(const *primitive_desc* &pd)

Constructs a softmax forward propagation primitive.

Parameters
pd – Primitive descriptor for a softmax forward propagation primitive.

struct **primitive_desc** : public dnnl::*primitive_desc*

Primitive descriptor for a softmax forward propagation primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *prop_kind* aprop_kind, *algorithm* aalgorithm, const *memory*::*desc* &src_desc, const *memory*::*desc* &dst_desc, int axis, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a softmax forward propagation primitive.

Parameters
- **aengine** – Engine to use.
- **aprop_kind** – Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- **aalgorithm** – Softmax algorithm kind: either *dnnl::algorithm::softmax_accurate*, or *dnnl::algorithm::softmax_log*.
- **src_desc** – Source memory descriptor.
- **dst_desc** – Destination memory descriptor.
- **axis** – Axis over which softmax is computed.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() const

Returns a source memory descriptor.

Returns
Source memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **dst_desc**() const

Returns a destination memory descriptor.

Returns
Destination memory descriptor.
Returns
A zero memory descriptor if the primitive does not have a destination parameter.

dnnl::*algorithm* **get_algorithm**() const

Returns an algorithm kind.
Returns
An algorithm kind.
Returns
dnnl::algorithm::undef if the primitive does not have an algorithm parameter.

dnnl::*prop_kind* **get_prop_kind**() const

Returns a propagation kind.
Returns
A propagation kind.
Returns
dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

int **get_axis**() const

Returns an axis.
Returns
An axis.
Returns
A negative number if the primitive does not have an axis parameter.

struct **softmax_backward** : public dnnl::*primitive*

Softmax backward propagation primitive.

Public Functions

softmax_backward()

Default constructor. Produces an empty object.

softmax_backward(const *primitive_desc* &pd)

Constructs a softmax backward propagation primitive.

Parameters
pd – Primitive descriptor for a softmax backward propagation primitive.

struct **primitive_desc** : public dnnl::*primitive_desc*

Primitive descriptor for a softmax backward propagation primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(const *engine* &aengine, *algorithm* aalgorithm, const *memory*::*desc* &diff_src_desc, const *memory*::*desc* &diff_dst_desc, const *memory*::*desc* &dst_desc, int axis, const *softmax_forward*::*primitive_desc* &hint_fwd_pd, const *primitive_attr* &attr = default_attr(), bool allow_empty = false)

Constructs a primitive descriptor for a softmax backward propagation primitive.

Parameters

- **aengine** – Engine to use.
- **aalgorithm** – Softmax algorithm kind: either *dnnl::algorithm::softmax_accurate*, or *dnnl::algorithm::softmax_log*.
- **diff_src_desc** – Diff source memory descriptor.
- **diff_dst_desc** – Diff destination memory descriptor.
- **dst_desc** – Destination memory descriptor.
- **axis** – Axis over which softmax is computed.
- **hint_fwd_pd** – Primitive descriptor for a softmax forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **attr** – Primitive attributes to use. Attributes are optional and default to empty attributes.
- **allow_empty** – A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **dst_desc**() const

Returns a destination memory descriptor.

Returns

Destination memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a destination parameter.

memory::*desc* **diff_src_desc**() const

Returns a diff source memory descriptor.

Returns

Diff source memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a diff source memory with.

memory::*desc* **diff_dst_desc**() const

Returns a destination memory descriptor.

Returns

Destination memory descriptor.

Returns

A zero memory descriptor if the primitive does not have a destination parameter.

dnnl::*algorithm* **get_algorithm**() const

Returns an algorithm kind.

Returns

An algorithm kind.

Returns

dnnl::algorithm::undef if the primitive does not have an algorithm parameter.

dnnl::*prop_kind* **get_prop_kind**() const

Returns a propagation kind.

Returns

A propagation kind.

Returns

dnnl::prop_kind::undef if the primitive does not have a propagation parameter.

int **get_axis**() const

Returns an axis.

Returns

An axis.

Returns

A negative number if the primitive does not have an axis parameter.

4.5.20 Sum

The sum primitive sums N tensors. The variable names follow the standard *Conventions*.

$$\mathrm{dst}(\overline{x}) = \sum_{i=1}^{N} scales(i) \cdot \mathrm{src}_i(\overline{x})$$

The sum primitive does not have a notion of forward or backward propagations. The backward propagation for the sum operation is simply an identity operation.

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

primitive input/output	execution argument index
src	*DNNL_ARG_MULTIPLE_SRC*
dst	*DNNL_ARG_DST*

Operation Details

- The dst memory format can be either specified by a user or derived the most appropriate one by the primitive. The recommended way is to allow the primitive to choose the appropriate format.
- The sum primitive requires all source and destination tensors to have the same shape. Implicit broadcasting is not supported.

Post-ops and Attributes

The sum primitive does not support any post-ops or attributes.

Data Types Support

The sum primitive supports arbitrary data types for source and destination tensors.

Data Representation

Sources, Destination

The sum primitive works with arbitrary data tensors. There is no special meaning associated with any logical dimensions.

API

struct **sum** : public dnnl::*primitive*

Out-of-place summation (sum) primitive.

Public Functions

sum()

Default constructor. Produces an empty object.

sum(const *primitive_desc* &pd)

Constructs a sum primitive.

Parameters
pd – Primitive descriptor for sum primitive.

struct **primitive_desc** : public dnnl::*primitive_desc_base*

Primitive descriptor for a sum primitive.

Public Functions

primitive_desc()

Default constructor. Produces an empty object.

primitive_desc(const *memory*::*desc* &dst, const std::vector<float> &scales, const std::vector<*memory*::*desc*> &srcs, const *engine* &aengine, const *primitive_attr* &attr = *primitive_attr*())

Constructs a primitive descriptor for a sum primitive.

Parameters
- **dst** – Destination memory descriptor.
- **scales** – Vector of scales to multiply data in each source memory by.
- **srcs** – Vector of source memory descriptors.
- **aengine** – Engine to perform the operation on.
- **attr** – Primitive attributes to use (optional).

primitive_desc(const std::vector<float> &scales, const std::vector<*memory*::*desc*> &srcs, const *engine* &aengine, const *primitive_attr* &attr = *primitive_attr*())

Constructs a primitive descriptor for a sum primitive.

This version derives the destination memory descriptor automatically.

Parameters
- **scales** – Vector of scales by which to multiply data in each source memory object.
- **srcs** – Vector of source memory descriptors.
- **aengine** – Engine on which to perform the operation.
- **attr** – Primitive attributes to use (optional).

memory::*desc* **src_desc**(int idx = 0) const

Returns a source memory descriptor.

Parameters
idx – Source index.

Returns
Source memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a source parameter with index `pdx`.

memory::*desc* **dst_desc**() const

Returns a destination memory descriptor.

Returns
Destination memory descriptor.

Returns
A zero memory descriptor if the primitive does not have a destination parameter.

4.6 Graph extension

oneDNN Graph extension is a flexible graph interface to maximize operation fusions in a scalable way. oneDNN Graph API accepts a full computational graph as input and performs a engine-aware graph partitioning, where sub-graphs of operations that are candidate for fusion are grouped together. Those partitions are then compiled and executed as fused operations.

4.6.1 Common Definitions

This section lists common types and definitions used by all or multiple graph operations.

Logical tensor enums and type

struct **logical_tensor**

Logical tensor object.

Public Types

enum class **data_type**

Data Type.

Values:

enumerator **undef**

enumerator **f16**

16-bit/half-precision floating point.

enumerator **bf16**

non-standard 16-bit (bfloat16 w/ 7 bit mantissa) floating point.

enumerator **f32**

32-bit/single-precision floating point.

enumerator **s32**

32-bit signed integer.

enumerator **s8**

8-bit signed integer.

enumerator **u8**

8-bit unsigned integer.

enumerator **boolean**

Boolean data type. Size is C++ implementation defined.

enum class **layout_type**

Layout type.

Values:

enumerator **undef**

Undefined layout type.

enumerator **any**

Any means to let the library to decide the layout for a tensor during partition compilation.

enumerator **strided**

Strided means that the layout of a tensor is determined by the strides field in the logical tensor.

enumerator **opaque**

Opaque means that the layout of a tensor is the library specific. Usually, an opaque layout is generated by a partition which is compiled with layout type any.

enum class **property_type**

Tensor property.

Values:

enumerator **undef**

Undefined tensor property.

enumerator **variable**

Variable means the tensor may be changed during computation or between different iterations.

enumerator **constant**

Constant means the tensor will keep unchanged during computation and between different iterations. It's useful for the library to apply optimizations for constant tensors or cache constant tensors inside the library. For example, constant weight tensors in inference scenarios.

using **dim** = dnnl_dim_t

Integer type for representing dimension sizes and indices.

using **dims** = std::vector<*dim*>

Vector of dimensions. Implementations are free to force a limit on the vector's length.

Public Functions

logical_tensor() = default

Default constructor Construct an empty object

logical_tensor(const *logical_tensor* &other) = default

Copy constructor.

logical_tensor &**operator=**(const *logical_tensor* &other) = default

Assignment operator.

logical_tensor(size_t tid, *data_type* dtype, int32_t ndims, *layout_type* ltype, *property_type* ptype = *property_type*::*undef*)

Constructs a logical tensor object with ID, data type, ndims, layout type, and property type.

Parameters

- **tid** – Logical tensor ID.
- **dtype** – Elements data type.
- **ndims** – Number of dimensions. *DNNL_GRAPH_UNKNOWN_NDIMS* for an unknown number of dimensions, 0 for a scalar tensor.
- **ltype** – Layout type.
- **ptype** – Property type.

inline **logical_tensor**(size_t tid, *data_type* dtype, *layout_type* ltype = *layout_type*::*undef*)

Delegated constructor.

Parameters

- **tid** – Logical tensor ID.
- **dtype** – Elements data type.
- **ltype** – Layout type.

logical_tensor(size_t tid, *data_type* dtype, const *dims* &adims, *layout_type* ltype, *property_type* ptype = *property_type*::*undef*)

Constructs a logical tensor object with basic information and detailed dims.

Parameters

- **tid** – Logical tensor ID.
- **dtype** – Elements data type.
- **adims** – Logical tensor dimensions. *DNNL_GRAPH_UNKNOWN_DIM* for dimensions of unknown size, 0 for zero-dimension tensor.
- **ltype** – Layout type. If strided, the strides field in the output logical tensor will be deduced accordingly.
- **ptype** – Property type.

logical_tensor(size_t tid, *data_type* dtype, const *dims* &adims, const *dims* &strides, *property_type* ptype = *property_type*::*undef*)

Constructs a logical tensor object with detailed dims and strides. The layout_type of the output logical tensor object will always be strided.

Parameters

- **tid** – Logical tensor ID.
- **dtype** – Elements data type.
- **adims** – Logical tensor dimensions. *DNNL_GRAPH_UNKNOWN_DIM* for dimensions of unknown size, 0 for zero-dimension tensor.
- **strides** – Logical tensor strides. *DNNL_GRAPH_UNKNOWN_DIM* for unknown stride. No negative stride is supported.
- **ptype** – Property type.

logical_tensor(size_t tid, *data_type* dtype, const *dims* &adims, size_t lid, *property_type* ptype = *property_type*::*undef*)

Constructs a logical tensor object with detailed dims and an opaque layout ID. layout_type of the output logical tensor object will always be opaque.

Parameters

- **tid** – Logical tensor ID.
- **dtype** – Elements data type.
- **adims** – Logical tensor dimensions. *DNNL_GRAPH_UNKNOWN_DIM* for dimensions of unknown size, 0 for zero-dimension tensor.
- **lid** – Opaque layout id.
- **ptype** – Property type

dims **get_dims**() const

Returns the dimensions of a logical tensor.

Returns
A vector describing the size of each dimension.

size_t **get_id**() const

Returns the unique id of a logical tensor.

Returns
An integer value describing the ID.

data_type **get_data_type**()

Returns the data type of a logical tensor.

Returns
The data type.

property_type **get_property_type**() const

Returns the property type of a logical tensor.

Returns
The property type.

layout_type **get_layout_type**() const

Returns the layout type of a logical tensor.

Returns
The layout type.

size_t **get_layout_id**() const

Returns the layout ID of a logical tensor. The API should be called on a logical tensor with opaque layout type. Otherwise, an exception will be raised.

Returns
Layout ID.

dims **get_strides**() const

Returns the strides of a logical tensor. The API should be called on a logical tensor with strided layout type. Otherwise, an exception will be raised.

Returns
A vector describing the stride size of each dimension.

size_t **get_mem_size**() const

Returns memory size in bytes required by this logical tensor.

Returns
The memory size in bytes.

bool **is_equal**(const *logical_tensor* &lt)

Compares if two logical tenors are equal. Users can decide accordingly if layout reordering is needed for two logical tensors. The method will return true for below two circumstances:

i. the two logical tensors are equal regarding each field in the struct, eg. id, ndims, dims, layout type, property, etc.

ii. If all other fields are equal but the layout types in two logical tensors are different, the method will return true when the underlying memory layout is the same. For example, one logical tensor has strided layout type while the other one has opaque layout type, but underneath, both layouts are NHWC, the method will still return true for this case.

Parameters
lt – The input logical tensor to be compared.

Returns
`true` if the two logical tensors are equal. `false` otherwise

Operation attributes and kinds

struct **op**

An op object.

Public Types

enum class **kind**

Kinds of operations.

Values:

enumerator **Abs**

enumerator **AbsBackward**

enumerator **Add**

enumerator **AvgPool**

enumerator **AvgPoolBackward**

enumerator **BatchNormForwardTraining**

enumerator **BatchNormInference**

enumerator **BatchNormTrainingBackward**

enumerator **BiasAdd**

enumerator **BiasAddBackward**

enumerator **Clamp**

enumerator **ClampBackward**

enumerator **Concat**

enumerator **Convolution**

enumerator **ConvolutionBackwardData**

enumerator **ConvolutionBackwardWeights**

enumerator **ConvTranspose**

enumerator **ConvTransposeBackwardData**

enumerator **ConvTransposeBackwardWeights**

enumerator **Dequantize**

enumerator **Divide**

enumerator **DynamicDequantize**

enumerator **DynamicQuantize**

enumerator **Elu**

enumerator **EluBackward**

enumerator **End**

enumerator **Exp**

enumerator **GELU**

enumerator **GELUBackward**

enumerator **HardSigmoid**

enumerator **HardSigmoidBackward**

enumerator **HardSwish**

enumerator **HardSwishBackward**

enumerator **Interpolate**

enumerator **InterpolateBackward**

enumerator **LayerNorm**

enumerator **LayerNormBackward**

enumerator **LeakyReLU**

enumerator **Log**

enumerator **LogSoftmax**

enumerator **LogSoftmaxBackward**

enumerator **MatMul**

enumerator **Maximum**

enumerator **MaxPool**

enumerator **MaxPoolBackward**

enumerator **Minimum**

enumerator **Mish**

enumerator **MishBackward**

enumerator **Multiply**

enumerator **Pow**

enumerator **PReLU**

enumerator **PReLUBackward**

enumerator **Quantize**

enumerator **Reciprocal**

enumerator **ReduceL1**

enumerator **ReduceL2**

enumerator **ReduceMax**

enumerator **ReduceMean**

enumerator **ReduceMin**

enumerator **ReduceProd**

enumerator **ReduceSum**

enumerator **ReLU**

enumerator **ReLUBackward**

enumerator **Reorder**

enumerator **Round**

enumerator **Select**

enumerator **Sigmoid**

enumerator **SigmoidBackward**

enumerator **SoftMax**

enumerator **SoftMaxBackward**

enumerator **SoftPlus**

enumerator **SoftPlusBackward**

enumerator **Sqrt**

enumerator **SqrtBackward**

enumerator **Square**

enumerator **SquaredDifference**

enumerator **StaticReshape**

enumerator **StaticTranspose**

enumerator **Subtract**

enumerator **Tanh**

enumerator **TanhBackward**

enumerator **TypeCast**

enumerator **Wildcard**

enumerator **LastSymbol**

enum class **attr**

Attributes of operations. Different operations support different attributes. Check the document of each operation for what attributes are supported and what are the potential values for them. Missing required attribute or illegal attribute value may lead to failure when adding the operation to a graph.

Values:

enumerator **undef**

Undefined op attribute.

enumerator **alpha**

Specifies an alpha attribute to an op.

enumerator **beta**

Specifies an beta attribute to an op.

enumerator **epsilon**

Specifies an epsilon attribute to an op.

enumerator **max**

Specifies a max attribute to an op.

enumerator **min**

Specifies a min attribute to an op.

enumerator **momentum**

Specifies a momentum attribute to an op.

enumerator **scales**

Specifies a scales attribute to an op.

enumerator **axis**

Specifies an axis attribute to an op.

enumerator **begin_norm_axis**

Specifies a begin_norm_axis attribute to an op.

enumerator **groups**

Specifies a groups attribute to an op.

enumerator **axes**

Specifies an axes attribute to an op.

enumerator **dilations**

Specifies a dilations attribute to an op.

enumerator **dst_shape**

Specifies an dst_shape attribute to an op.

enumerator **kernel**

Specifies a kernel attribute to an op.

enumerator **order**

Specifies an order attribute to an op.

enumerator **output_padding**

Specifies an output_padding attribute to an op.

enumerator **pads_begin**

Specifies a pads_begin attribute to an op.

enumerator **pads_end**

Specifies a pads_end attribute to an op.

enumerator **shape**

Specifies a shape attribute to an op.

enumerator **sizes**

Specifies a sizes attribute to an op.

enumerator **src_shape**

Specifies an src_shape attribute to an op.

enumerator **strides**

Specifies a strides attribute to an op.

enumerator **weights_shape**

Specifies a weight_shape attribute to an op.

enumerator **zps**

Specifies a zps attribute to an op.

enumerator **exclude_pad**

Specifies an exclude_pad attribute to an op.

enumerator **keep_dims**

Specifies a keep_dims attribute to an op.

enumerator **keep_stats**

Specifies a keep_stats attribute to an op.

enumerator **per_channel_broadcast**

Specifies a per_channel_broadcast attribute to an op.

enumerator **special_zero**

Specifies a special_zero attribute to an op.

enumerator **transpose_a**

Specifies a transpose_a attribute to an op.

enumerator **transpose_b**

Specifies a transpose_b attribute to an op.

enumerator **use_affine**

Specifies an use_affine attribute to an op.

enumerator **use_dst**

Specifies an use_dst attribute to an op.

enumerator **auto_broadcast**

Specifies an auto_broadcast attribute to an op. The value can be "none" or "numpy".

enumerator **auto_pad**

Specifies an auto_pad attribute to an op. The value can be "none", "same_upper", "same_lower", or "valid".

enumerator **coordinate_transformation_mode**

Specifies an coordinate_transformation_mode attribute to an op. The value can be "half_pixel" or "align_corners". The attribute is defined for Interpolate operations.

enumerator **data_format**

Specifies a data_format of an op. The value can be "NCX" or "NXC".

enumerator **mode**

Specifies a mode attribute of an op. The value can be "nearest", "linear", "bilinear", or "trilinear". The attribute is defined for Interpolate operations.

enumerator **qtype**

Specifies a qtype attribute to an op. The value can be "per_channel" or "per_tensor". The attribute is defined for quantization operations.

enumerator **rounding_type**

Specifies a rounding_type attribute to an op. The value can be "ceil" or "floor".

enumerator **weights_format**

Specifies a weights_format of an op. The value can be "OIX", "XIO", "IOX", or "XOI". Different operations may support different values.

Public Functions

op(size_t id, *kind* akind, const std::string &name = "")

Constructs an op object with an unique ID, an operation kind, and a name string.

Parameters

- **id** – The unique ID of the op.
- **akind** – The op kind specifies which computation is represented by the op, such as Convolution or ReLU.
- **name** – The string added as the op name.

op(size_t id, *kind* akind, const std::vector<*logical_tensor*> &inputs, const std::vector<*logical_tensor*> &outputs, const std::string &name = "")

Constructs an op object with an unique ID, an operation kind, and input/output logical tensors.

Parameters

- **id** – The unique ID of this op.
- **akind** – The op kind specifies which computation is represented by this op, such as Convolution or ReLU.
- **inputs** – Input logical tensor to be bound to this op.
- **outputs** – Output logical tensor to be bound to this op.
- **name** – The string added as the op name.

void **add_input**(const *logical_tensor* &t)

Adds an input logical tensor to the op.

Parameters
t – Input logical tensor.

void **add_inputs**(const std::vector<*logical_tensor*> &ts)

Adds a vector of input logical tensors to the op.

Parameters
ts – The list of input logical tensors.

void **add_output**(const *logical_tensor* &t)

Adds an output logical tensor to the op.

Parameters
t – Output logical tensor.

void **add_outputs**(const std::vector<*logical_tensor*> &ts)

Adds a vector of output logical tensors to the op.

Parameters
ts – The list of output logical tensors.

template<typename **Type**>
op &**set_attr**(*attr* name, const *Type* &value)

Sets the attribute according to the name and type.

Template Parameters
Type – Attribute's type.

Parameters

- **name** – Attribute's name.
- **value** – The attribute's value.

Returns
The Op self. Raises an exception if Type is incompatible with name.

Graph objects member functions

struct **graph**

A graph object.

Public Functions

graph(dnnl::*engine*::*kind* engine_kind)

Constructs a graph with an engine kind.

Parameters
engine_kind – Engine kind.

graph(dnnl::*engine*::*kind* engine_kind, dnnl::fpmath_mode mode)

Creates a new empty graph with an engine kind and a floating-point math mode. All partitions returned from the graph will inherit the engine kind and floating-point math mode.

Parameters

- **engine_kind** – Engine kind.
- **mode** – Floating-point math mode.

status **add_op**(const *op* &op, bool allow_exception = true)

Adds an op into the graph to construct a computational DAG. The API will return failure if the operator has already been added to the graph or the operation cannot pass the schema check in the library (eg. input and output numbers and data types, the attributes of the operation, etc.).

Parameters

- **op** – An operation to be added.
- **allow_exception** – A flag indicating whether the method is allowed to throw an exception if it fails to add the op to the graph.

Returns
status::success or a status describing the error otherwise.

void **finalize**()

Finalizes a graph. It means users have finished adding operations into the graph and the graph is ready for partitioning. Adding a new operation into a finalized graph will return failures. Similarly, partitioning on a un-finalized graph will also return failures.

bool **is_finalized**() const

Checks if a graph is finalized.

Returns
True if the graph is finalized or false if the graph is not finalized.

std::vector<partition> **get_partitions**(partition::policy policy = partition::policy::fusion)

Gets filtered partitions from a graph. Partitions will be claimed internally according to the capability of the library, the engine kind of the graph, and the policy.

Parameters

policy – Partition policy, defaults to policy dnnl::graph::partition::policy::fusion.

Returns

A vector storing the partitions.

Macros to specify unknown shapes

DNNL_GRAPH_UNKNOWN_NDIMS

A wildcard value for number of dimensions which is unknown at a tensor or operation creation time.

DNNL_GRAPH_UNKNOWN_DIM

A wildcard value for dimensions that are unknown at a tensor or operation creation time.

4.6.2 Programming Model

oneDNN Graph programming model allows users to pass a computation graph and get partitions. Users then compile partitions, bind tensor data, and execute compiled partitions. Partitions are decided by the oneDNN Graph implementation, which allows a scalable (no change in user code to benefit from new fusion patterns) and platform aware partitioning.

The programming model assumes that the main usage is to support deep learning (DL) frameworks or inference engines. DL frameworks have their own representation for the computation graph. oneDNN Graph API is used to offload or accelerate graph partitions from a framework graph. In the description below, "graph" refers to the graph built by oneDNN Graph implementation, and "framework graph" refers to the graph built by the DL framework.

A deep learning computation graph consists of deep neural network (DNN) operations. A DNN operation is a function that takes input data and returns output data. The input and output data are multidimensional arrays called tensors. A DNN operation may consume multiple tensors and produce multiple tensors. A tensor must be produced by a single operation and may be consumed by multiple operations.

oneDNN Graph API uses logical tensor, OP, and graph to represent a computation graph. Logical tensor represents tensor's metadata, like element data type, shape, and layout. OP represents an operation on a computation graph. OP has kind, attribute, and input and output logical tensors. OPs are added to a graph. Both OP and logical tensor contains a unique ID, so that the graph knows how to connect a producer OP to a consumer OP through a logical tensor. The graph constructed is immutable. The purpose of creating the graph object is to get partitions. After partitions are created, the graph object is not useful anymore. Once users get partitions, users should not add OP to the graph.

oneDNN Graph defines operation set. Users should convert their DNN operation definition to oneDNN Graph operation for graph construction. For operation outside oneDNN Graph operation set, users may use wild-card OP. The wild-card OP represents any OP. With its input and output logical tensors, it enables the oneDNN Graph implementation to receive a full graph and conduct a complete analysis. User needs to use a special "End" op to indicate output tensors of the graph. For any tensors needs to be alive after a graph being executed, it needs to be connected to a "End" op which consumes the tensor. Users may have multiple "End" ops for one graph. For each OP users add to the graph, users must describe its input and output logical tensors. Users must describe data type for each logical tensor. If tensor's shape and layout are known, users must describe them along with the logical tensor.

A partition is a connected subgraph in a graph. oneDNN Graph implementation analyzes a graph and returns a number of partitions. The returned partitions completely cover all the OPs of the graph and follow topological order. A partition typically contains multiple Ops. Sometimes a partition may contain just one OP, like a Wildcard OP or unsupported

OP. A partition contains a flag to indicate whether the partition is supported and thus can be compiled and executed. User needs to check the flag before using the partition.

Partition's input and output is also called as port. The ports record the logical tensor information which was passed during graph construction. With the logical tensor ID, users can track the producer and consumer relationship between partitions. The ports also record the data type of corresponding logical tensors.

The returned partitions to users must not form a dependence cycle. For example, a graph contains 3 OPs: A, B, and C. If C consumes A's output and produces B's input, oneDNN Graph implementation must not put A and B into one partition. However, if C is not added to the graph, the returned partition may include A and B, since C is not visible to oneDNN Graph implementation. In this case, it is the user's responsibility to detect the dependence cycle. Once users pass a complete graph, users don't need to check the dependence cycle among the partitions returned by oneDNN Graph.

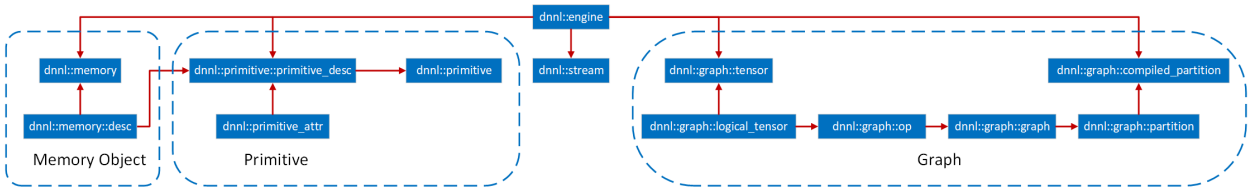
A partition needs to be compiled before execution. The compilation lowers down the compute logic to hardware ISA level and generates binary code. The generated code is specialized for the input and output tensor's metadata. Users must create new logical tensors to pass complete metadata with the compilation API. The logical tensors should fully specify id, data type, shape (can be incomplete for outputs), and layout, the compilation should succeed. The logical tensors passed during compilation time must match IDs with partition's ports. The logical tensors must have same data types with the ports with the port of the same ID.

For the output logical tensors, users must either specify a public layout using size and stride for each tensor dimension or request oneDNN Graph implementation to decide a target-specific layout. For the input logical tensors, users must either specify a public layout or using a target-specific layout produced by predecessor partition compilation. For the logical tensor with target-specific layout, it must be produced by a partition and used only by partitions.

A compiled partition represents the generated code specialized for target hardware and tensor metadata passed with compilation API. Users may cache the compiled partition to amortize the compilation cost among many iterations. If tensor metadata is identical, a compiled partition generated in previous iterations may be reused. Alternatively, implementations may reduce the partition compilation cost by caching the compiled partition internally. This optimization falls outside of the scope of this specification.

To execute a compiled partition, users must pass input and output tensors. Input tensors must bind input data buffers to logical tensors. Users may query the compiled partition for output data buffer sizes. If the sizes are known, users may allocate the output data buffers and bind to output tensors. If the sizes are unknown, users must provide an allocator for oneDNN Graph implementation to allocate the output tensor buffer. The execution API takes a compiled partition, input tensors, and return output tensors with the data buffer updated.

An engine represents a target device and context in the system. It needs to be passed as a parameter for partition compilation. A stream abstracts hardware execution resources of a target device. It is required to execute a compiled partition.



The diagram above summarizes the key programming concepts, and how they interact with each other. The arrow indicates the destination object contains or uses the source object. For example, OP contains logical tensor, and compiled partition uses partition.

Logical Tensor

Logical tensor describes the metadata of the input or output tensor, like element data type, number of dimensions, size for each dimension, layout.

Besides helping oneDNN Graph implementation to build the graph, Logical tensor plays a critical role to exchange tensor metadata information between users and oneDNN Graph implementation. Users pass input tensor shape information and get the inferred shape for output tensors from a partition. Users pass logical tensors to compilation API for specifying shape and layout information. Users also use a special logical tensor to allow oneDNN Graph implementation to decide the layout for output tensors. After compilation, users can query the compiled partition for output tensors' shape, layout, and sizes.

Each logical tensor has an ID. The tensor metadata may include new shape information in the framework graph as it progresses toward execution. As a logical tensor is not mutable, users must create a new logical tensor with the same ID to pass any new additional information to oneDNN Graph implementation. Users should guarantee that the logical tensor ID is unique within the graph which the logical tensor belongs to.

Operations

An operation (or *OP*) describes a deep neural network operation. OP contains kind, attribute, and input and output logical tensor shapes and properties. In particular, activation and weights tensor formats are specified as attributes to the operation.

Each operation has a unique ID and users should guarantee that uniqueness within the graph which the OP is added to.

Graph

Graph contains a set of OPs. *dnnl::graph::graph::add_op()* adds an OP and its logical tensors to a graph. oneDNN Graph implementation accumulates the OPs and logical tensors and constructs and validates the graph as internal state. During *dnnl::graph::graph::add_op()*, the target OP will be validated against its schema. Once the validation fails, an exception will be thrown out from the API. When `allow_exception=false` is specified, *dnnl::graph::graph::add_op()* call returns a status. It is the user's responsibility to handle the error either by checking the return value of the API or handling the exception.

A same logical tensor may appear more than twice in *dnnl::graph::graph::add_op()* call, since it is passed with the producer OP and consumer OPs. oneDNN Graph validates logical tensors with the same id should be identical at the graph construction time.

Once the graph is fully described, *dnnl::graph::graph::finalize()* should be called. This prevents any other operation from being added, and allows to call *dnnl::graph::graph::get_partitions()* in order to get the set of partitions for that graph. The graph does not hold any meaning to the user after partitioning and should freed by the user.

All the OPs added to the graph will be contained in one of the returned partitions. If an OP is not supported by the oneDNN Graph API implementation, the corresponding partition will be marked as "not supported". Users can check the supporting status of a partition via the `dnnl::graph::partition::is_supported`. Partitions should not form cyclic dependence within the graph. If user does not pass a complete graph, it is the user's responsibility to detect any dependence cycle between the partitions and operations not passing to oneDNN Graph implementation.

The logical tensor passed at the graph construction stage might contain incomplete information, for example, dimension and shape information are spatially known. Complete information is not required but helps the oneDNN Graph to form better partition decisions. Adding op to a graph is not thread-safe. Users must create a graph, add op, and get partition in the same thread.

Partition

Partition represents a collection of OPs identified by oneDNN Graph implementation as the basic unit for compilation and execution. It contains a list of OP, input ports, output ports, and a flag indicating whether the partition is supported. When a partition is created, it's assigned with an ID. oneDNN Graph implementation should guarantee the partition ID is globally unique.

Users can pass the output logical tensors with incomplete shape information (containing *DNNL_GRAPH_UNKNOWN_NDIMS* or *DNNL_GRAPH_UNKNOWN_DIM*) to partition compilation API. oneDNN Graph implementation needs calculate the output shapes according to the given input shapes and schema of the OP. After compilation finished, a compiled partition will be generated with full shape information for the input and output logical tensors. Users can query the compiled partition for the output logical tensors and get the shapes.

Partition can be compiled to generate a `compiled_partition`: an executable object to run the computation for that partition. Users must create an input logical tensor list and an output logical tensor list to pass the additional tensor metadata as parameters to the compilation API. The input and output logical tensors must match the id of partitions' ports, which captures the logical tensors information during graph partitioning. Typically, the more information is given before the partition step (e.g. number of dimensions and tensor dimensions), the most performant the code under the compiled partition will be.

Users must specify `strided`, `any`, or `opaque` as the `layout_type` for the parameter logical tensors. When users specify `any` for a logical tensor, the tensor must be an output tensor, and oneDNN Graph implementation decides the best performant layout for the compiled partition. If it is `strided`, it must use the public data layout described by the logical tensor. For `opaque`, the parameter logical tensor contains a target-specific layout, which must be determined by the compilation of preceding partitions producing the tensor. If the layout is row-major contiguous, the compilation must succeed. If the layout has a stride, it is implementation dependent whether the compilation succeed. If certain dimension of shape or the rank is unknown, it is implementation dependent whether the compilation succeed. If the compilation succeeds for unknown dimension or rank, the compiled partition should be able to handle any value for that dimension or any rank at the execution time.

Tensor

Tensor is an abstraction for multidimensional input and output data needed in the execution of a compiled partition. A tensor contains a logical tensor, an engine and a data handle.

Users are responsible for managing the tensor's lifecycle, e.g. free the resource allocated, when it is not used anymore.

Compiled Partition

A *compiled partition* represents the generated code specialized for target hardware and meta data described by parameter logical tensors. Compiled partition contains a partition and a handle representing the target specific compiled object.

After the compilation API is invoked, users must query the logical output tensor of the compiled partition to know the output tensor's layout id and size. The layout id is an opaque identifier for the target-specific layout. Users may pass the layout id for the next partition compilation so that it can be optimized to expect a specific input layout. Users may use the size to allocate the memory buffer of the output tensors for execution.

Framework passes the tensors and compiled partition as parameters to execution API. The parameter logical tensors must be in the same order when they are passed in the compilation API, and their IDs must match with the compiled partition's internal logical tensors. The layout type of each tensor must be `strided` or `opaque`.

The compiled partition may support in-place optimization, which reuses the input tensor data buffer for the output tensor for lower memory footprint and better data locality. For each compiled partition, users can get pairs of input and output ports. For the pair of input and output ports, user can use a same memory buffer when passing input and output tensors along with execution API. The in-place optimization is optional, when users use another memory buffer for the output tensor, oneDNN Graph must update the output tensor.

If users place a tensor with data buffer pointer in outputs, the backend shall use the data buffer provided by users.

Users may convert the parameter tensor with public layout to the target specific layout expected by the compiled partition. A common optimization in deep learning inference is that users may prepack the weight in the target-specific layout required by the compiled partition and cache the reordered weight for late use.

Engine

Engine (`dnnl::engine`) are an abstraction of a computational device. The graph extension additionally allows to create an engine with specific host/device allocators to conveniently manage memory inside the Graph API calls.

Stream

Stream (`dnnl::stream`) encapsulate execution context tied to a particular engine.

General API notes

There are certain assumptions on how oneDNN Graph objects behave:

- Logical tensor behave similarly to trivial types.
- All other objects behave like shared pointers. Copying is always shallow.

Error Handling

The C++ API throws exceptions for error handling.

4.6.3 Data Model

oneDNN Graph uses logical tensor to describe data type, shape, and layout. Besides 32-bit IEEE single-precision floating-point data type, oneDNN Graph can also support other data types. The shape contains multiple dimensions, and the total dimension and the size of the dimension could be set as unknown.

oneDNN Graph uses the following enumeration to refer to data types it supports. Different operation may support inputs and outputs with different data types, so it's suggested to refer to the definition page of each operation.

enum *dnnl::graph::logical_tensor::data_type*

undef
: Undefined data type (used for empty logical tensor).

f16
: 16-bit/half-precision floating point.

bf16
: non-standard 16-bit floating point with 7-bit mantissa.

f32
: 32-bit/single-precision floating point.

s32
: 32-bit signed integer.

s8
: 8-bit signed integer.

u8
8-bit unsigned integer.

boolean
Boolean data type. Size is C++ implementation defined.

oneDNN Graph supports both public layout and *opaque* layout. When the *layout_type* of logical tensor is *strided*, it means that the tensor layout is public which the user can identify each tensor element in the physical memory.

For example, for $dims[][][] = x, y, z$, $strides[][][] = s0, s1, s2$, the physical memory location should be in $s0 * x + s1 * y + s2 * z$.

When the *layout_type* of logical tensor is *opaque*, users are not supposed to interpret the memory buffer directly. An *opaque* tensor can only be output from oneDNN Graph's compiled partition and can only be fed to another compile partition as an input tensor.

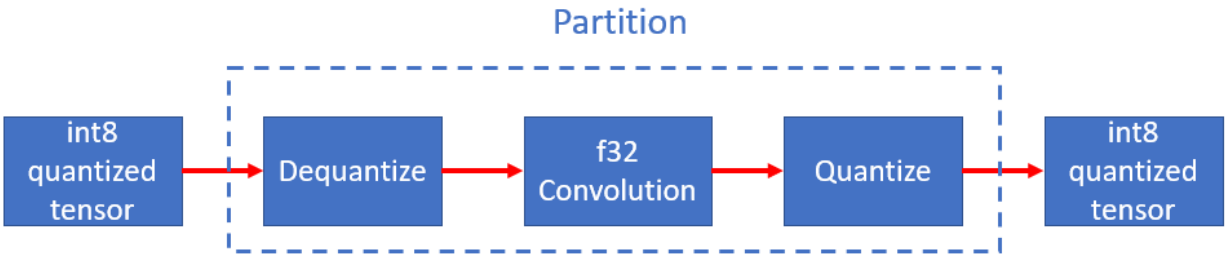
Low Precision Support

oneDNN Graph extension provides the same low precision support as the oneDNN primitives, including *u8*, *s8*, *bf16* and *f16*. For int8, oneDNN Graph API supports quantized model with static quantization. For bf16 or f16, oneDNN Graph supports deep learning framework's auto mixed precision mechanism. In both cases, oneDNN Graph API expects users to convert the computation graph to low precision representation and specify the data's precision and quantization parameters. oneDNN Graph API implementation should strictly respect the numeric precision of the computation.

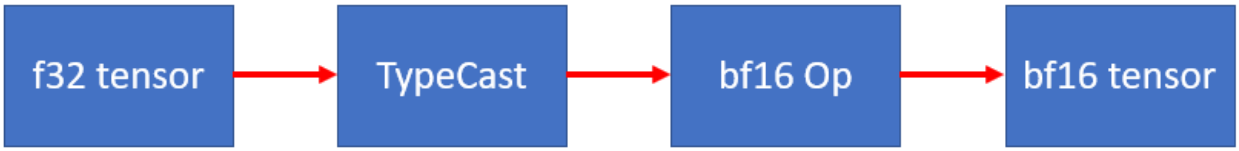
For int8, oneDNN Graph API provides two operations *dequantize* and *quantize*. Dequantize takes integer tensor with its associated scale and zero point and returns f32 tensor. Quantize takes an f32 tensor, scale, zero point, and returns an integer tensor. The scale and zero point are single dimension tensors, which could contain one value for the per-tensor quantization case or multiple values for the per-channel quantization case. The integer tensor could be represented in both unsigned int8 or signed int8. Zero point could be zero for symmetric quantization scheme, and a non-zero value for asymmetric quantization scheme.

Users should insert Dequantize and Quantize in the graph as part of quantization process before passing to oneDNN Graph. oneDNN Graph honors the data type passed from user and faithfully follows the data type semantics. For example, if the graph has a Quantize followed by Dequantize with exact same scale and zero point, oneDNN Graph implementation should not eliminate them since that implicitly changes the numerical precision.

oneDNN Graph partitioning API may return a partition containing the Dequantize, Quantize, and Convolution operations in between. Users don't need to recognize the subgraph pattern explicitly and convert to fused op. Depending on oneDNN Graph implementation capability, the partition may include more or fewer operations.



For bf16, oneDNN Graph provides the *typecast* operation, which can convert an f32 tensor to bf16 or f16, and vice versa. All oneDNN Graph operations support bf16 and f16. It is the user's responsibility to insert TypeCast to clearly indicate the numerical precision. oneDNN Graph implementation fully honors the user-specified numerical precision. If users first typecast from f32 to bf16 and convert back, oneDNN Graph implementation does the exact data type conversions underneath.



4.6.4 Operation Set

oneDNN Graph defines an operation set. oneDNN Graph implementation may support a subset of the operation set.

Abs

Abs operation performs element-wise the absolute value with given tensor, it applies following formula on every element of src tensor (the variable names follow the standard *Conventions*):

$$dst = \begin{cases} src & \text{if } src \geq 0 \\ -src & \text{if } src < 0 \end{cases}$$

Operation Attributes

Abs operation does not support any attribute.

Execution Arguments

Inputs

Index	Argument Name	Required or Optional
0	src	Required

Outputs

Index	Argument Name	Required or Optional
0	dst	Required

Supported Data Types

Abs operation supports the following data type combinations.

Src	Dst
f32	f32
f16	f16
bf16	bf16

AbsBackward

AbsBackward operation computes gradient for Abs operation.

$$\text{dst} = \begin{cases} \text{diff\_dst} & \text{if } src > 0 \\ -\text{diff\_dst} & \text{if } src < 0 \\ 0 & \text{if } src = 0 \end{cases}$$

Operation Attributes

AbsBackward operation does not support any attribute.

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	src	Required
1	diff_dst	Required

Outputs

Index	Argument Name	Required or Optional
0	diff_src	Required

Supported Data Types

AbsBackward operation supports the following data type combinations.

Src	Diff_dst	Diff_src
f32	f32	f32
f16	f16	f16
bf16	bf16	bf16

Add

Add operation performs element-wise addition operation with two given tensors applying multi-directional broadcast rules.

$$\mathrm{dst}(\overline{x}) = \mathrm{src\_0}(\overline{x}) + \mathrm{src\_1}(\overline{x}).$$

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
auto_broadcast	Specifies rules used for auto-broadcasting of src tensors	string	`none`, `numpy`, (default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src_0`	Required
1	`src_1`	Required

@note Both src shapes should match and no auto-broadcasting is allowed if *auto_broadcast* attributes is `none`. `src_0` and `src_1` shapes can be different and auto-broadcasting is allowed if *auto_broadcast* attributes is `numpy`.

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

Add operation supports the following data type combinations.

Src_0 / Src_1	Dst
f32	f32
bf16	bf16
f16	f16

AvgPool

AvgPool operation performs the computation following the below formulas. Variable names follow the standard *Conventions*.

$$\mathrm{dst}(n, c, oh, ow) = \frac{1}{DENOM} \sum_{kh,kw} \mathrm{src}(n, c, oh \cdot SH + kh - PH_L, ow \cdot SW + kw - PW_L)$$

where,

- when attribute `exclude_pad` is set to false, in which case $DENOM = KH \cdot KW$,
- when attribute `exclude_pad` is set to true, in which case $DENOM$ equals to the size of overlap between an averaging window and images.

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
strides	Controls the strides the window is moved	s64	A s64 list containing positive values	Required
pads_begin	Controls number of zeros to be add to the front/top/left of spatial dimensions, the attribute will be ignored when *auto_pad* attribute is specified to `same_upper`, `same_lower` or `valid`	s64	A s64 list containing non-negative values	Required
pads_end	Controls number of zeros to be add to the back/bottom/right of spatial dimensions, the attribute will be ignored when *auto_pad* attribute is specified to `same_upper`, `same_lower` or `valid`	s64	A s64 list containing non-negative values	Required
kernel	Size of pooling window	s64	A s64 list containing positive values	Required
exclude_pad	Controls whether the padded values are counted	bool	True, False	required
rounding_type	Controls how to do rounding	string	`floor` (default), `ceil`	Optional
auto_pad	Controls how the paddings are calculated	string	`none` (default), `same_upper`, `same_lower`, `valid`	Optional
data_format	Controls how to interpret the shape of `src` and `dst`.	string	`NCX`, `NXC` (default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

AvgPool operation supports the following data type combinations.

Src	Dst
f32	f32
bf16	bf16
f16	f16

AvgPoolBackward

AvgPoolBackward operation accepts diffdst tensor and srcshape tensor (optional), and calculates diffsrc tensor.

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
strides	Controls the strides the window is moved	s64	A s64 list containing positive values	Required
pads_begin	Controls number of zeros to be add to the front/top/left of spatial dimensions, the attribute will be ignored when *auto_pad* attribute is specified to `same_upper`, `same_lower` or `valid`	s64	A s64 list containing n on-negative values	Required
pads_end	Controls number of zeros to be add to the back/bottom/right of spatial dimensions, the attribute will be ignored when *auto_pad* attribute is specified to `same_upper`, `same_lower` or `valid`	s64	A s64 list containing n on-negative values	Required
kernel	Size of pooling window	s64	A s64 list containing positive values	Required
exclude_pad	Controls whether the padded values are counted	bool	True, False	Required
auto_pad	Controls how the paddings are calculated	string	`none` (default), `same_upper`, `same_lower`, `valid`	Optional
data_format	Controls how to interpret the shape of `src` and `dst`.	string	`NCX`, `NXC` (default)	Optional
src_shape	Denotes the shape of input of forward op	string	`NCX`, `NXC` (default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`diff_dst`	Required
1	`src_shape`	Optional

@note Either `src_shape` input or *`src_shape`* attribute should be provided. If both provided, `src_shape` input will precede over *`src_shape`* attribute.

Outputs

Index	Argument Name	Required or Optional
0	`diff_src`	Required

Supported Data Types

AvgPoolBackward operation supports the following data type combinations.

Diff_dst	Diff_src	Src_shape
f32	f32	s64
bf16	bf16	s64
f16	f16	s64

BatchNormForwardTraining

BatchNormForwardTraining operation performs batch normalization at training mode.

Mean and variance are computed at runtime, the following formulas are used:

- $\mu(c) = \frac{1}{NHW} \sum_{nhw} \text{src}(n, c, h, w)$,
- $\sigma^2(c) = \frac{1}{NHW} \sum_{nhw} (\text{src}(n, c, h, w) - \mu(c))^2$.

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
epsilon	A number to be added to the variance to avoid division by zero	f32	A positive f32 value	Required
momentum	A number to be used to calculate running mean and running variance	f32	A positive f32 value	Optional
data_format	Controls how to interpret the shape of `src` and `dst`.	string	`NCX`, `NXC` (default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required
1	`mean`	Required
2	`variance`	Required
3	`gamma`	Optional
4	`beta` (σ^2)	Optional

@note `gamma` and `beta` should be either both provided or neither provided.

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required
1	`running_mean`	Required
2	`running_variance`	Required
3	`batch_mean`	Required
4	`batch_variance`	Required

Supported Data Types

BatchNormInference operation supports the following data type combinations.

Src / Dst	Gamma / Beta / Mean / Variance / Batch_mean / Batch_variance / Running_mean / Running_variance
f32	f32
bf16	f32, bf16
f16	f32

BatchNormInference

The formula is the same as *Batch Normalization primitive :ref: `batch_normalization-label* like below.

$$\mathrm{dst}(n,c,h,w) = \gamma(c) \cdot \frac{\mathrm{src}(n,c,h,w) - \mu(c)}{\sqrt{\sigma^2(c) + \varepsilon}} + \beta(c),$$

where

- $\gamma(c), \beta(c)$ are required scale and shift for a channel,
- $\mu(c), \sigma^2(c)$ are mean and variance for a channel, and
- ε is a constant to improve numerical stability.

Operation Attributes

Attribute Name	Description	Value Type	**Supported** Values	**Required or** Optional
epsilon	A number to be added to the variance to avoid division by zero	f32	A positive float value	Required
data_format	Controls how to interpret the shape of `src` and `dst`.	string	`NCX`, `NXC` (default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required
1	`gamma`	Required
2	`beta`	Required
3	`mean`	Required
4	`variance` (σ^2)	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

BatchNormInference operation supports the following data type combinations.

Src / Dst	Gamma / Beta / Mean / Variance
f32	f32
bf16	f32, bf16
f16	f32

BatchNormTrainingBackward

BatchNormTrainingBackward operation calculated the gradients of input tensors.

Operation Attributes

Attribute Name	Description	Value Type	**Supported** Values	**Required or** Optional
epsilon	A number to be added to the variance to avoid division by zero	f32	A positive f32 value	Required
data_format	Controls how to interpret the shape of `src` and `dst`.	string	`NCX`, `NXC` (default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required
1	`diff_dst`	Required
2	`mean`	Required
3	`variance`	Required
4	`gamma` (σ^2)	Optional

Outputs

Index	Argument Name	Required or Optional
0	`diff_src`	Required
1	`diff_gamma`	Optional
2	`diff_beta`	Optional

@note `diff_gamma` and `diff_beta` should be either both provided or neither provided. If neither provided, the input `gamma` will be ignored.

Supported Data Types

BatchNormTrainingBackward operation supports the following data type combinations.

Src / Diff_dst / Diff_src	Mean / Variance / Gamma / Diff_gamma / Diff_beta
f32	f32
bf16	f32, bf16
f16	f32

BiasAdd

Add bias to channel dimension of input. This is a special `Add` with bias restricted to be 1-D. Broadcasting is supported.

$$\mathrm{dst}(n, c, h, w) = \mathrm{src}(n, c, h, w) + \mathrm{bias}(c)$$

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
data_format	Controls how to interpret the shape of `src` and `dst`.	string	`NCX` , `NXC` (default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required
1	`bias`	Required

@note `bias` is a 1D tensor to be added to `src` tensor. The size should be the same as size of channel dimension of `src` tensor.

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

BiasAdd operation supports the following data type combinations.

Src	Bias	Dst
f32	f32	f32
bf16	bf16	bf16
f16	f16	f16

BiasAddBackward

BiasAddBackward operation computes the gradients on the bias tensor for BiasAdd operator. This op accumulates all the values from diff_dst into the channel dimension, the axis depends on the layout of src tensor.

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
data_format	Controls how to interpret the shape of `diff_dst` and `diff_bias`.	string	`NCX` , `NXC` (default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`diff_dst`	Required

Outputs

Index	Argument Name	Required or Optional
0	`diff_bias`	Required

Supported Data Types

BiasAddBackward operation supports the following data type combinations.

diff_dst	diff_bias
f32	f32
bf16	bf16
f16	f16

Clamp

Clamp operation represents clipping activation function, it applies following formula on every element of src tensor (the variable names follow the standard *Conventions*):

$$clamp(src_i) = min(max(src_i, min\_value), max\_value)$$

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
min	The lower bound of values in the output. Any value in the input that is smaller than the bound, is replaced with the `min` value.	f32	Arbitrary valid f32 value	Required
max	The upper bound of values in the output. Any value in the input that is greater than the bound, is replaced with the `max` value.	f32	Arbitrary valid f32 value	Required

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

Clamp operation supports the following data type combinations.

Src	Dst
f32	f32
f16	f16
bf16	bf16

ClampBackward

ClampBackward operation computes gradient for Clamp.

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
min	The lower bound of values in the output. Any value in the input that is smaller than the bound, is replaced with the `min` value.	f32	Arbitrary valid f32 value	Required
max	The upper bound of values in the output. Any value in the input that is greater than the bound, is replaced with the `max` value.	f32	Arbitrary valid f32 value	Required
use_dst	If true, use `dst` of Clamp operation to calculate the gradient. Otherwise, use `src`.	bool	`true` (default), `false`	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src` / `dst`	Required
1	`diff_dst`	Required

Outputs

Index	Argument Name	Required or Optional
0	`diff_src`	Required

Supported Data Types

ClampBackward operation supports the following data type combinations.

Src / Dst	Diff_dst	Diff_src
f32	f32	f32
f16	f16	f16
bf16	bf16	bf16

Concat

Concat operation concatenates N tensors over `axis` (here designated C) and is defined as (the variable names follow the standard *Conventions*):

$$\mathrm{dst}(\overline{ou}, c, \overline{in}) = \mathrm{src}\_i(\overline{ou}, c', \overline{in}),$$

where $c = C_1 + .. + C_{i-1} + c'$.

Operation Attributes

Attribute Name	Description	Value Type	**Supported** Values	**Required or** Optional
`axis`	Specifies dimension along which concatenation happens	s64	A s64 value in the range of [-r, r-1] where r = rank(src)	Required

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src_i`	Required

@note At least one input tensor is required. Data types and ranks of all input tensors should match. The dimensions of all input tensors should be the same except for the dimension specified by `axis` attribute.

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

Concat operation supports the following data type combinations.

Src_i	Dst
f32	f32
bf16	bf16
f16	f16

ConvTranspose

ConvTranspose operation performs the same computation as in ConvolutionBackwardData, except the source and destination are swapped.

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
strides	Controls the strides the weights tensor is moved when computing convolution	s64	A s64 list containing positive values	Required
pads_begin	Controls number of zeros to be add to the front/top/left of spatial dimensions	s64	A s64 list containing non-negative values	Required
pads_end	Controls number of zeros to be add to the back/bottom/right of spatial dimensions	s64	A s64 list containing non-negative values	Required
dilations	Controls the amount of stretching the kernel before convolution	s64	A s64 list containing positive values (>1 means dilated convolution)	Required
auto_pad	Controls how the padding is calculated	string	`none` (default), `same_upper`, `same_lower`, `valid`	Optional
output_padding	Adds additional amount of padding per each spatial axis in `dst`	s64	A s64 list containing non-negative values, all zeros by default	Optional
groups	Controls how input channels and output channels are divided into	s64	A positive s64 value, 1 by default	Optional
data_format	Controls how to interpret the shape of `src` and `dst`.	string	`NCX`, `NXC` (default)	Optional
weights_format	Controls how to interpret the shape of `weights`	string	`IOX`, `XOI` (default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required
1	`weights`	Required
2	`bias`	Optional

@note The shape of weights is $(in\_channels/groups, out\_channels, spatial\_shape)$ for `IOX` format or $(spatial\_shape, out\_channels, in\_channels/groups)$ for `XOI` format. Both $in\_channels$ and $out\_channels$ must be divisible by *groups* attribute.

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

ConvTranspose operation supports the following data type combinations.

Src	Weights	Bias	Dst
f32	f32	f32	f32
bf16	bf16	bf16	bf16
f16	f16	f16	f16

ConvTransposeBackwardData

ConvTransposeBackwardData operation takes diffdst and weights and computes diffsrc.

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
strides	Controls the strides the weights tensor is moved when computing convolution	s64	A s64 list containing positive values	Required
pads_begin	Controls number of zeros to be add to the front/top/left of spatial dimensions	s64	A s64 list containing non-negative values	Required
pads_end	Controls number of zeros to be add to the back/bottom/right of spatial dimensions	s64	A s64 list containing non-negative values	Required
dilations	Controls the amount of stretching the kernel before convolution	s64	A s64 list containing positive values (>1 means dilated convolution)	Required
auto_pad	Controls how the padding is calculated	string	`none` (default), `same_upper`, `same_lower`, `valid`	Optional
output_padding	Adds additional amount of padding per each spatial axis in `dst`	s64	A s64 list containing non-negative values, all zeros by default	Optional
groups	Controls how input channels and output channels are divided into	s64	A positive s64 value, 1 by default	Optional
data_format	Controls how to interpret the shape of `src` and `dst`.	string	`NCX`, `NXC` (default)	Optional
weights_format	Controls how to interpret the shape of `weights`	string	`IOX`, `XOI` (default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`diff_dst`	Required
1	`weights`	Required

@note The shape of weights is $(in\_channels/groups, out\_channels, spatial\_shape)$ for `IOX` format or $(spatial\_shape, out\_channels, in\_channels/groups)$ for `XOI` format. Both $in\_channels$ and $out\_channels$ must be divisible by *groups* attribute.

Outputs

Index	Argument Name	Required or Optional
0	`diff_src`	Required

Supported Data Types

ConvTransposeBackwardData operation supports the following data type combinations.

Diff_dst	Weights	Diff_src
f32	f32	f32
bf16	bf16	bf16
f16	f16	f16

ConvTransposeBackwardWeights

ConvTransposeBackwardWeights operation takes $\mathrm{diff\_dst}$, src and optional $weights\_shape$ computes $\mathrm{diff\_weights}$.

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
strides	Controls the strides the weights tensor is moved when computing convolution	s64	A s64 list containing positive values	Required
pads_begin	Controls number of zeros to be add to the front/top/left of spatial dimensions	s64	A s64 list containing non-negative values	Required
pads_end	Controls number of zeros to be add to the back/bottom/right of spatial dimensions	s64	A s64 list containing non-negative values	Required
dilations	Controls the amount of stretching the kernel before convolution	s64	A s64 list containing positive values (>1 means dilated convolution)	Required
auto_pad	Controls how the padding is calculated	string	`none` (default), `same_upper`, `same_lower`, `valid`	Optional
output_padding	Adds additional amount of padding per each spatial axis in `dst`	s64	A s64 list containing non-negative values, all zeros by default	Optional
groups	Controls how input channels and output channels are divided into	s64	A positive s64 value, 1 by default	Optional
data_format	Controls how to interpret the shape of `src` and `dst`.	string	`NCX`, `NXC` (default)	Optional
weights_format	Controls how to interpret the shape of `weights`	string	`IOX`, `XOI` (default)	Optional
weights_shape	Denotes the shape of the `weights` tensor	s64	A s64 list containing positive values	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required
1	`diff_dst`	Required
2	`weights_shape`	Optional

@note The shape of weights is $(in\_channels/groups, out\_channels, spatial\_shape)$ for `IOX` format or $(spatial\_shape, out\_channels, in\_channels/groups)$ for `XOI` format. Both $in\_channels$ and $out\_channels$ must be divisible by *groups* attribute.

@note Either `weights_shape` input or *weights_shape* attribute should be provided. If both provided, `weights_shape` input will precede over the *weights_shape* attribute.

Outputs

Index	Argument Name	Required or Optional
0	`diff_weights`	Required

Supported Data Types

ConvTransposeBackwardWeights operation supports the following data type combinations.

Src	Diff_dst	Diff_weights	Weights_shape
f32	f32	f32	s32
bf16	bf16	bf16	s32
f16	f16	f16	s32

Convolution

Convolution operation performs the convolution between src tensor and weight tensor, which is defined as by the following formulas. Variable names follow the standard *Conventions*.

Let src, weights and dst tensors have shape $N \times IC \times IH \times IW$, $OC \times IC \times KH \times KW$, and $N \times OC \times OH \times OW$ respectively.

Furthermore, let the remaining convolution parameters be:

Parameter	Depth	Height	Width	Comment
Paddings: Front, top, and left	PD_L	PH_L	PW_L	In the attributes we use `pads_begin` to indicate the corresponding vector of paddings
Padding: Back, bottom, and right	PD_R	PH_R	PW_R	In the attributes we use `pads_end` to indicate the corresponding vector of paddings
Stride	SD	SH	SW	In the attributes we use `strides` to indicate the corresponding vector of strides
Dilation	DD	DH	DW	In the attributes we use `dilations` to indicate the corresponding vector of dilations

To further simplify the formulas, we assume that the attribute `data_format` and `weights_format` are set to `NCX` and `OIX` respectively. `NCX` means the fist axis represents batch dimension, the second axis represents channel dimension and the rest represents spatial dimensions. `OIX` means the first axis represents output channel dimension, the second axis represents input channel dimension and the rest represents weights spatial dimensions.

Regular Convolution

This is the same as the formula in *Convolution primitive :ref: `convolution-label*.

$$\begin{aligned}&\operatorname{dst}(n, oc, oh, ow) = \operatorname{bias}(oc) \\ &+ \sum_{ic=0}^{IC-1} \sum_{kh=0}^{KH-1} \sum_{kw=0}^{KW-1} \operatorname{src}(n, ic, oh', ow') \cdot \operatorname{weights}(oc, ic, kh, kw).\end{aligned}$$

Here:

- $OH = \left\lfloor \frac{IH - KH + PH_L + PH_R}{SH} \right\rfloor + 1$,
- $OW = \left\lfloor \frac{IW - KW + PW_L + PW_R}{SW} \right\rfloor + 1$.

Convolution with Groups

The attribute `groups` is set to > 1.

$$\begin{aligned}&\operatorname{dst}(n, g \cdot OC_G + oc_g, oh, ow) = \operatorname{bias}(g \cdot OC_G + oc_g) \\ &+ \sum_{ic_g=0}^{IC_G-1} \sum_{kh=0}^{KH-1} \sum_{kw=0}^{KW-1} \operatorname{src}(n, g \cdot IC_G + ic_g, oh', ow') \cdot \operatorname{weights}(g, oc_g, ic_g, kh, kw),\end{aligned}$$

where

- $IC_G = \frac{IC}{G}$,
- $OC_G = \frac{OC}{G}$, and
- $oc_g \in [0, OC_G)$.

Convolution with Dilation

The attribute *dilations* contains the element which is > 1.

$$\begin{aligned}&\mathrm{dst}(n, oc, oh, ow) = \mathrm{bias}(oc) + \\ &+ \sum_{ic=0}^{IC-1} \sum_{kh=0}^{KH-1} \sum_{kw=0}^{KW-1} \mathrm{src}(n, ic, oh'', ow'') \cdot \mathrm{weights}(oc, ic, kh, kw).\end{aligned}$$

Here:

- $OH = \left\lfloor \frac{IH - DKH + PH_L + PH_R}{SH} \right\rfloor + 1$, where $DKH = 1 + (KH - 1) \cdot DH$, and
- $OW = \left\lfloor \frac{IW - DKW + PW_L + PW_R}{SW} \right\rfloor + 1$, where $DKW = 1 + (KW - 1) \cdot DW$.

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
strides	Controls the strides the weights tensor is moved when computing convolution	s64	A s64 list containing positive values	Required
pads_begin	Controls number of zeros to be add to the front/top/left of spatial dimensions	s64	A s64 list containing non-negative values	Required
pads_end	Controls number of zeros to be add to the back/bottom/right of spatial dimensions	s64	A s64 list containing non-negative values	Required
dilations	Controls the amount of stretching the kernel before convolution	s64	A s64 list containing positive values (>1 means dilated convolution)	Required
auto_pad	Controls how the padding is calculated	string	`none` (default), `same_upper`, `same_lower`, `valid`	Optional
groups	Controls how input channels and output channels are divided into	s64	A positive s64 value, 1 by default	Optional
data_format	Controls how to interpret the shape of `src` and `dst`.	string	`NCX`, `NXC` (default)	Optional
weights_format	Controls how to interpret the shape of `weights`	string	`OIX`, `XIO` (default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required
1	`weights`	Required
2	`bias`	Optional

@note The shape of weights is $(out\_channels, in\_channels/groups, spatial\_shape)$ for `OIX` format or $(spatial\_shape, in\_channels/groups, out\_channels)$ for `XIO` format. Both $in\_channels$ and $out\_channels$ must be divisible by *groups* attribute.

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

Convolution operation supports the following data type combinations.

Src	Weights	Bias	Dst
f32	f32	f32	f32
bf16	bf16	bf16	bf16
f16	f16	f16	f16

ConvolutionBackwardData

ConvolutionBackwardData operation accepts diff_dst, weights and optional dst shape as inputs, and compute the diff_src.

If *auto_pad* attribute is specified to one of `valid`, `same_upper` and `same_lower`, *pads_begin* and `pads_end` attributes will be ignored. The paddings will be calculated by following the below formula:

Let the parameters be:

Parameter	Depth	Height	Width	Comment
Paddings: Front, top, and left	PD_L	PH_L	PW_L	In the attributes we use `pads_begin` to indicate the corresponding vector of paddings
Padding: Back, bottom, and right	PD_R	PH_R	PW_R	In the attributes we use `pads_end` to indicate the corresponding vector of paddings
Stride	SD	SH	SW	In the attributes we use `strides` to indicate the corresponding vector of strides
Dilation	DD	DH	DW	In the attributes we use `dilations` to indicate the corresponding vector of dilations

Firstly, $total\_padding$ is calculated according to src_shape and dst_shape. Let src_h be height dimension of src_shape and dst_h be height dimension of dst_shape.

$$total\_padding_h = SH \times (src_h - 1) + ((KH - 1) \times DH + 1) - \mathrm{dst}_h + output\_padding_h$$

If `auto_pad` attribute is specified as `valid`:

$$PD_L = 0$$
$$PD_R = 0$$

If `auto_pad` attribute is specified as `same_lower`:

$$PD_L = floor(total\_padding/2)$$
$$PD_R = total\_padding - PD_L$$

If `auto_pad` attribute is specified as `same_upper`:

$$PD_L = total\_padding - PD_R$$
$$PD_R = floor(total\_padding/2)$$

where:

- dst_shape is either an attribute or an input tensor,
- $output\_padding$ is an optional attribute.

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
strides	Controls the strides the weights tensor is moved when computing convolution	s64	A s64 list containing positive values	Required
pads_begin	Controls number of zeros to be add to the front/top/left of spatial dimensions, the attribute will be ignored when *auto_pad* attribute is specified to `same_upper`, `same_lower` or `valid`	s64	A s64 list containing non-negative values	Required
pads_end	Controls number of zeros to be add to the back/bottom/right of spatial dimensions, the attribute will be ignored when *auto_pad* attribute is specified to `same_upper`, `same_lower` or `valid`	s64	A s64 list containing non-negative values	Required
dilations	Controls the amount of stretching the kernel before convolution	s64	A s64 list containing positive values (>1 means dilated convolution)	Required
auto_pad	Controls how the padding is calculated	string	`none` (default), `same_upper`, `same_lower`, `valid`	Optional
output_padding	Adds additional amount of padding per each spatial axis in `dst`	s64	A s64 list containing non-negative values, all zeros by default	Optional
groups	Controls how input channels and output channels are divided into	s64	A positive s64 value, 1 by default	Optional
data_format	Controls how to interpret the shape of `src` and `dst`.	string	`NCX`, `NXC` (default)	Optional
weights_format	Controls how to interpret the shape of `weights`	string	`OIX`, `XIO` (default)	Optional
dst_shape	Denotes the shape of the `dst` tensor	s64	A s64 list containing positive values	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`diff_dst`	Required
1	`weights`	Required
2	`dst_shape`	Optional

@note The shape of weights is $(out\_channels, in\_channels/groups, spatial\_shape)$ for `OIX` format or $(spatial\_shape, in\_channels/groups, out\_channels)$ for `XIO` format. Both $in\_channels$ and $out\_channels$ must be divisible by *groups* attribute.

@note Either `dst_shape` input or `dst_shape` attribute should be provided. If both provided, `dst_shape` input will precede over `dst_shape` attribute.

Outputs

Index	Argument Name	Required or Optional
0	`diff_src`	Required

Supported Data Types

ConvolutionBackwardData operation supports the following data type combinations.

Diff_dst	Weights	Diff_src	Dst_shape
f32	f32	f32	s32
bf16	bf16	bf16	s32
f16	f16	f16	s32

ConvolutionBackwardWeights

ConvolutionBackwardWeights operation accepts src, $\mathrm{diff\_dst}$ and optional weights shape as inputs, and compute the $\mathrm{diff\_weights}$.

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
strides	Controls the strides the weights tensor is moved when computing convolution	s64	A s64 list containing positive values	Required
pads_begin	Controls number of zeros to be add to the front/top/left of spatial dimensions, the attribute will be ignored when *auto_pad* attribute is specified to `same_upper`, `same_lower` or `valid`	s64	A s64 list containing non-negative values	Required
pads_end	Controls number of zeros to be add to the back/bottom/right of spatial dimensions, the attribute will be ignored when *auto_pad* attribute is specified to `same_upper`, `same_lower` or `valid`	s64	A s64 list containing non-negative values	Required
dilations	Controls the amount of stretching the kernel before convolution	s64	A s64 list containing positive values (>1 means dilated convolution)	Required
auto_pad	Controls how the padding is calculated	string	`none` (default), `same_upper`, `same_lower`, `valid`	Optional
groups	Controls how input channels and output channels are divided into	s64	A positive s64 value, 1 by default	Optional
data_format	Controls how to interpret the shape of `src` and `dst`.	string	`NCX`, `NXC` (default)	Optional
weights_format	Controls how to interpret the shape of `weights`	string	`OIX`, `XIO` (default)	Optional
weights_shape	Denotes the shape of the `weights` tensor	s64	A s64 list containing positive values	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required
1	`diff_dst`	Required
2	`weights_shape`	Optional

@note The shape of weights is $(out\_channels, in\_channels/groups, spatial\_shape)$ for `OIX` format or $(spatial\_shape, in\_channels/groups, out\_channels)$ for `XIO` format. Both $in\_channels$ and $out\_channels$ must be divisible by *groups* attribute.

@note Either `weights_shape` input or *weights_shape* attribute should be provided. If both provided, `weights_shape` input will precede over *weights_shape* attribute.

Outputs

Index	Argument Name	Required or Optional
0	`diff_weights`	Required

Supported Data Types

ConvolutionBackwardWeights operation supports the following data type combinations.

Src	Diff_dst	Diff_weights	Weights_shape
f32	f32	f32	s32
bf16	bf16	bf16	s32
f16	f16	f16	s32

Dequantize

Dequantize operation converts a quantized (u8 or s8) tensor to a f32 tensor. It supports both per-tensor and per-channel asymmetric linear de-quantization. Rounding mode is library-implementation defined.

For per-tensor de-quantization:

$$\mathrm{dst}_{f32} = round((\mathrm{src}_i - zps) \times scale)$$

For per-channel de-quantization, taking channel axis = 1 as an example:

$$\mathrm{dst}_{\dots,i,\dots,\dots} = (\mathrm{src}_{\dots,i,\dots,\dots} - zps_i) \times scale_i, i \in [0, ic-1]$$

where ic is the number of channels.

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
qtype	Specifies which dequantization type is used	string	`per_tensor` (default), `per_channel`	Optional
axis	Specifies dimension on which per-channel dequantization is applied	s64	A s64 value in the range of [-r, r-1] where r = rank(src), 1 by default	Optional
scales	Scalings applied on the src data	f32	A f32 list (only contain one element if *qtype* is `per_tensor`)	Required
zps	Offset values that maps to float zero	s64	A s64 list (only contain one element if *qtype* is `per_tensor`)	Required

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

Dequantize operation supports the following data type combinations.

Src	Dst
s8, u8	f32

@note This operation is to support *int8 quantization* model.

Divide

Divide operation performs element-wise division operation with two given tensors applying multi-directional broadcast rules.

$$\mathrm{dst}(\overline{x}) = \mathrm{src\_0}(\overline{x}) \ / \ \mathrm{src\_1}(\overline{x}),$$

Operation Attributes

Attribute Name	Description	Value Type	**Supported** Values	**Required or** Optional
auto_broadcast	Specifies rules used for auto-broadcasting of src tensors	string	`none`, `numpy`, (default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src_0`	Required
1	`src_1`	Required

@note Both src shapes should match and no auto-broadcasting is allowed if `auto_broadcast` attributes is `none`. `src_0` and `src_1` shapes can be different and auto-broadcasting is allowed if `auto_broadcast` attributes is `numpy`. Broadcasting is performed according to auto_broadcast value.

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

Divide operation supports the following data type combinations.

Src_0 / Src_1	Dst
f32	f32
bf16	bf16
f16	f16

DynamicDequantize

DynamicDequantize operation converts a quantized (s8 or u8) tensor to a f32 tensor. It supports both per-tensor and per-channel asymmetric linear de-quantization. Rounding mode is library-implementation defined. Unlike the *Dequantize*, DynamicDequantize takes scales and zero-points as operator src tensors.

For per-tensor de-quantization

$$dst = (src - zps) * scales$$

For per-channel de-quantization, taking channel axis = 1 as an example:

$$\mathrm{dst}_{\dots,i,\dots,\dots} = (\mathrm{src}_{\dots,i,\dots,\dots} - zps_i) * scales_i, i \in [0, channelNum - 1]$$

Operation Attributes

Attribute Name	Description	Value Type	**Supported** Values	**Required or** Optional
qtype	Specifies which de-quantization type is used	string	`per_tensor` (default), `per_channel`	Optional
axis	Specifies dimension on which per-channel de-quantization is applied	s64	A s64 value in the range of [-r, r-1] where r = rank(src), 1 by default	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required
1	`scales`	Required
2	`zps`	Optional

@note `scales` is a f32 1D tensor to be applied to the de-quantization formula. For `qtype` = `per-tensor`, there should be only one element in the scales tensor. For `qtype` = `per-channel`, the element number should be equal to the element number of src tensor along the dimension axis.

@note `zps` is a 1D tensor with offset values that map to zero. For `qtype` = `per-tensor`, there should be only one element in the zps tensor. For `qtype` = `per-channel`, the element number should be equal to the element number of input tensor along the dimension axis. If not specified, the library can assume the operator is symmetric de-quantization and perform kernel optimization accordingly.

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

DynamicDequantize operation supports the following data type combinations.

Src	Dst	Scales	Zps
s8	f32	f32	s8, u8, s32
u8	f32	f32	s8, u8, s32

DynamicQuantize

DynamicQuantize operation converts a f32 tensor to a quantized (s8 or u8) tensor. It supports both per-tensor and per-channel asymmetric linear quantization. The target quantized data type is specified via the data type of dst logical tensor. Rounding mode is library-implementation defined.

For per-tensor quantization:

$$\mathrm{dst}_i = round(\mathrm{src}_i / scale + zp)$$

For per-channel quantization, taking channel axis = 1 as an example:

$$dst_{\dots,i,\dots,\dots} = round(\mathrm{src}_{\dots,i,\dots,\dots} / scale_i + zp_i), i \in [0, ic-1]$$

where ic is the number of channels.

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
qtype	Specifies which dequantization type is used	string	`per_tensor` (default), `per_channel`	Optional
axis	Specifies dimension on which per-channel dequantization is applied	s64	A s64 value in the range of [-r, r-1] where r = rank(src), 1 by default	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required
1	`scales`	Required
2	`zps`	Optional

@note `scales` is a f32 1D tensor to be applied to the quantization formula. For `qtype` = `per-tensor`, there should be only one element in the scales tensor. For `qtype` = `per-channel`, the element number should be equal to the element number of src tensor along the dimension axis.

@note `zps` is a 1D tensor with offset values that map to zero. For `qtype` = `per-tensor`, there should be only one element in the zps tensor. For `qtype` = `per-channel`, the element number should be equal to the element number of input tensor along the dimension axis. If not specified, the library can assume the operator is symmetric quantization and perform kernel optimization accordingly.

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

DynamicQuantize operation supports the following data type combinations.

Src	Scales	Zps	Dst
f32	f32	s8, u8, s32	s8
f32	f32	s8, u8, s32	u8

Elu

Elu operation applies following formula on every element of src tensor (the variable names follow the standard *Conventions*):

$$\text{dst} = \begin{cases} \alpha(e^{\text{src}} - 1) & \text{if src} < 0 \\ \text{src} & \text{if src} \geq 0 \end{cases}$$

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
alpha	Scale for the negative factor.	f32	Arbitrary non-negative f32 value	Required

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

Elu operation supports the following data type combinations.

Src	Dst
f32	f32
bf16	bf16
f16	f16

EluBackward

EluBackward operation computes gradient for Elu operation.

Operation Attributes

Attribute Name	Description	Value Type	**Supported** Values	**Required or** Optional
alpha	Scale for the negative factor.	f32	Arbitrary non-negative f32 value	Required
use_dst	If true, use dst of Elu operation to calculate the gradient. Otherwise, use src.	bool	`true` (default), `false`	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src` / `dst`	Required
1	`diff_dst`	Required

Outputs

Index	Argument Name	Required or Optional
0	`diff_src`	Required

Supported Data Types

EluBackward operation supports the following data type combinations.

Src / Dst	Diff_dst	Diff_src
f32	f32	f32
f16	f16	f16
bf16	bf16	bf16

End

End operation is used to help construct graph, for example tracking the uses of a tensor.

Operation Attributes

End operation does not support any attribute.

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required

Outputs

End operation does not support output tensor.

Supported Data Types

End operation supports the following data type combinations.

Src	Destination
f32	f32
bf16	bf16
f16	f16

Exp

Exp operation is an exponential element-wise activation function, it applies following formula on every element of src tensor (the variable names follow the standard *Conventions*):

$$dst = e^{src}$$

Operation Attributes

Exp operation does not support any attribute.

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

Exp operation supports the following data type combinations.

Src	Dst
f32	f32
f16	f16
bf16	bf16

GELU

GELU operation applies following formula on every element of src tensor (the variable names follow the standard *Conventions*):

$$\mathrm{dst} = 0.5\,\mathrm{src} \cdot (1.0 + erf(\mathrm{src})/\sqrt{2})$$

Operation Attributes

GELU operation does not support any attribute.

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

GELU operation supports the following data type combinations.

Src	Dst
f32	f32
bf16	bf16
f16	f16

GELUBackward

GELUBackward operation computes gradient for GELU.

Operation Attributes

GELUBackward operation does not support any attribute.

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required
1	`diff_dst`	Required

Outputs

Index	Argument Name	Required or Optional
0	`diff_src`	Required

Supported Data Types

GELUBackward operation supports the following data type combinations.

Src	Diff_dst	Diff_src
f32	f32	f32
f16	f16	f16
bf16	bf16	bf16

HardSigmoid

HardSigmoid operation applies the following formula on every element of src tensor (the variable names follow the standard @ref dev_guide_conventions):

$$\mathrm{dst} = \max(0, \min(1, \alpha\,\mathrm{src} + \beta))$$

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
alpha	α in the formula.	f32	Arbitrary f32 value	Required
beta	β in the formula.	f32	Arbitrary f32 value	Required

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

HardSigmoid operation supports the following data type combinations.

Src	Dst
f32	f32
bf16	bf16
f16	f16

HardSigmoidBackward

HardSigmoidBackward operation computes gradient for HardSigmoid. The formula is defined as follows:

$$\text{diff\_src} = \begin{cases} \text{diff\_dst} \cdot \alpha & \text{if } 0 < \alpha \, \text{src} + \beta < 1 \\ 0 & \text{otherwise} \end{cases}$$

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
alpha	α in the formula.	f32	Arbitrary f32 value	Required
beta	β in the formula.	f32	Arbitrary f32 value	Required

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required
1	`diff_dst`	Required

Outputs

Index	Argument Name	Required or Optional
0	`diff_src`	Required

Supported Data Types

HardSigmoidBackward operation supports the following data type combinations.

Src	Diff_dst	Diff_src
f32	f32	f32
f16	f16	f16
bf16	bf16	bf16

HardSwish

HardSwish operation applies following formula on every element of src tensor (the variable names follow the standard *Conventions*):

$$\mathrm{dst} = \mathrm{src} * \frac{\min(\max(\mathrm{src}+3, 0), 6)}{6}$$

Operation Attributes

HardSwish operation does not support any attribute.

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

HardSwish operation supports the following data type combinations.

Src	Dst
f32	f32
bf16	bf16
f16	f16

HardSwishBackward

HardSwishBackward operation computes gradient for HardSwish.

Operation Attributes

HardSwishBackward operation does not support any attribute.

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required
1	`diff_dst`	Required

Outputs

Index	Argument Name	Required or Optional
0	`diff_src`	Required

Supported Data Types

HardSwishBackward operation supports the following data type combinations.

Src	Diff_dst	Diff_src
f32	f32	f32
f16	f16	f16
bf16	bf16	bf16

Interpolate

Interpolate layer performs interpolation on src tensor at spatial dimensions.

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
mode	Specifies type of interpolation	string	`nearest`, `linear`, `bilinear`, `trilinear`	Required
coordinate_transf	Specifies how to transform the coordinate in the resized tensor to the coordinate in the original tensor	string	`half_pixel` (default), `align_corners`	Optional
sizes	Specifies dst shape for spatial axes.	s64	A s64 list containing positive values, `none` is default	Optional
scales	Specifies `scales` for spatial axes.	f32	A f32 list, `none` is default	Optional
data_format	Controls how to interpret the shape of `src` and `dst`	string	`NCX`, `NXC` (default)	Optional

@note Either `sizes` or `scales` should be provided. When `sizes` is used, `scales` will be ignored.

@note The attribute *coordinate_transformation_mode* is the name of transformation mode in string format. Here `scale[x]` is `dst_shape[x]/src_shape[x]` and `x_resized` is a coordinate in axis `x`,for any axis `x` from the src axis.n For `half_pixel`: the coordinate in the original tensor axis `x` is calculated as `((x_resized + 0.5) / scale[x]) - 0.5`.n For `align_corners`: the coordinate in the original tensor axis `x` is calculated as 0 if `dst_shape[x] == 1` else `x_resized * (src_shape[x] - 1) / (dst_shape[x] - 1)`.

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required
1	`sizes`	Optional

@note `sizes` is a 1D tensor describing output shape for spatial axes. It is a non-differentiable tensor.

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

@note The shape of the dst matches src shape except spatial dimensions. For spatial dimensions, they should match sizes from sizes or calculated from *scales* attribute.

Supported Data Types

Interpolate operation supports the following data type combinations.

Src / Dst	Sizes
f32	s32
bf16	s32
f16	s32

InterpolateBackward

InterpolateBackward computes the gradients of Interpolate operation.

Operation Attributes

Attribute Name	Description	Value Type	**Supported** Values	**Required or** Optional
mode	Specifies type of interpolation	string	`nearest`, `linear`, `bilinear`, `trilinear`	Required
coordinate_transf	Specifies how to transform the coordinate in the resized tensor to the coordinate in the original tensor	string	`half_pixel` (default), `align_corners`	Optional
sizes	Specifies dst shape for spatial axes.	s64	A s64 list containing positive values, `none` is default	Optional
scales	Specifies `scales` for spatial axes.	f32	A f32 list, `none` is default	Optional
data_format	Controls how to interpret the shape of `src` and `dst`	string	`NCX`, `NXC` (default)	Optional

@note Either `sizes` or `scales` should be provided. When `sizes` is used, `scales` will be ignored.

@note The attribute *coordinate_transformation_mode* is the name of transformation mode in string format.n Here `scale[x]` is `dst_shape[x]/src_shape[x]` and `x_resized` is a coordinate in axis `x`,for any axis `x` from the src axis.n For `half_pixel`: the coordinate in the original tensor axis `x` is calculated as `((x_resized + 0.5) / scale[x]) - 0.5`.n For `align_corners`: the coordinate in the original tensor axis `x` is calculated as 0 if `dst_shape[x] == 1` else `x_resized * (src_shape[x] - 1) / (dst_shape[x] - 1)`.n

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required
1	`diff_dst`	Required
2	`sizes`	Optional

@note `src` is original input tensor of Interpolate op.n `diff_dst` is the gradient tensor with respect to the dst.n `sizes` is a 1D tensor describing output shape for spatial axes.

Outputs

Index	Argument Name	Required or Optional
0	`diff_src`	Required

@note `diff_src` is the gradient tensor with respect to the src of Interpolate.

Supported Data Types

InterpolateBackward operation supports the following data type combinations.

Src	Diff_dst	Diff_src	Sizes
f32	f32	f32	s32
bf16	bf16	bf16	s32
f16	f16	f16	s32

LayerNorm

LayerNorm performs a layer normalization operation on src tensor.

The layerNorm operation performs normalization from `begin_norm_axis` to last dimension of the data tensor. It is defined by the following formulas which is the same as *Layer normalization*.

$$\mathrm{dst}(t,n,c) = \gamma(c) \cdot \frac{\mathrm{src}(t,n,c) - \mu(t,n)}{\sqrt{\sigma^2(t,n) + \epsilon}} + \beta(c),$$

where

- $\gamma(c), \beta(c)$ are optional scale and shift for a channel
- $\mu(t,n), \sigma^2(t,n)$ are mean and variance (see
- ϵ is a constant to improve numerical stability.

Mean and variance are computed at runtime or provided by a user. When mean and variance are computed at runtime, the following formulas are used:

- $\mu(t,n) = \frac{1}{C} \sum_c \mathrm{src}(t,n,c)$,
- $\sigma^2(t,n) = \frac{1}{C} \sum_c (\mathrm{src}(t,n,c) - \mu(t,n))^2$.

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
keep_stats	Indicate whether to output mean and variance which can be later passed to backward op	bool	`false`, `true` (default)	Optional
begin_norm_axis	*begin_norm_axis* is used to indicate which axis to start layer normalization. The normalization is from *begin_norm_axis* to last dimension. Negative values means indexing from right to left. This op normalizes over the last dimension by default, e.g. C in TNC for 3D and LDNC for 4D.	s64	[-r,r-1],where r=rank(src). -1 is default	Optional
use_affine	When set to True, this module has learnable per-element affine parameters.	bool	`false`, `true` (default)	Optional
epsilon	The constant to improve numerical stability	f32	Arbitrary positive f32 value, `1e-5` (default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required
1	`gamma`	Optional
2	`beta`	Optional

@note `gamma` is scaling for normalized value. `beta` is the bias added to the scaled normalized value. They are both 1D tensor with the same span as src's channel axis and required if attribute `use_affine` is set to True.

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required
1	`mean`	Optional
2	`variance`	Optional

@note Both `mean` and `variance` are required if attribute `keep_stats` is set to True.

Supported Data Types

LayerNorm operation supports the following data type combinations.

Src / Dst	Gamma / Beta / Mean / Variance
f32	f32
bf16	f32, bf16
f16	f32

LayerNormBackward

LayerNormBackward performs the backward of LayerNorm operation.

The backward propagation computes $\mathrm{diff\_src}(t,n,c)$, $\mathrm{diff\_}\gamma(c)^*$, and $\mathrm{diff\_}\beta(c)^*$ based on $\mathrm{diff\_dst}(t,n,c)$, $src(t,n,c)$, $\mu(t,n)$, $\sigma^2(t,n)$, $\gamma(c)^*$, and $\beta(c)^*$.

The tensors marked with an asterisk are used only when the operation is configured to use $\gamma(c)$, and $\beta(c)$

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
begin_norm_axis	*begin_norm_axis* is used to indicate which axis to start layer normalization. The normalization is from *begin_norm_axis* to last dimension. Negative values means indexing from right to left. This op normalizes over the last dimension by default, e.g. C in TNC for 3D and LDNC for 4D.	s64	[-r,r-1],where r=rank(src). -1 is default	Optional
use_affine	When set to True, this module has learnable per-element affine parameters.	bool	`false`, `true` (default)	Optional
epsilon	The constant to improve numerical stability	f32	Arbitrary positive f32 value, `1e-5` (default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required
1	`diff_dst`	Required
2	`mean`	Required
3	`variance`	Required
4	`gamma`	Optional
5	`beta`	Optional

@note `gamma` is scaling for normalized value. `beta` is the bias added to the scaled normalized value. They are both 1D tensor with the same span as src's channel axis and required if attribute *use_affine* is set to True.

Outputs

Index	Argument Name	Required or Optional
0	`diff_src`	Required
1	`diff_gamma`	Optional
2	`diff_beta`	Optional

Supported Data Types

LayerNormBackward operation supports the following data type combinations.

Src / Diff_dst / Diff_src	Gamma / Beta / Mean / Variance / Diff_gamma / Diff_beta
f32	f32
bf16	f32, bf16
f16	f32

LeakyReLU

LeakyReLU operation is a type of activation function based on ReLU. It has a small slope for negative values with which LeakyReLU can produce small, non-zero, and constant gradients with respect to the negative values. The slope is also called the coefficient of leakage.

Unlike *PReLU*, the coefficient α is constant and defined before training.

LeakyReLU operation applies following formula on every element of src tensor (the variable names follow the standard *Conventions*):

$$\text{dst} = \begin{cases} \text{src} & \text{if } \text{src} \geq 0 \\ \alpha \, \text{src} & \text{if } \text{src} < 0 \end{cases}$$

Operation Attributes

Attribute Name	Description	Value Type	**Supported** Values	**Required or** Optional
alpha	Alpha is the coefficient of leakage.	f32	Arbitrary f32 value but usually a small positive value.	Required

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

LeakyReLU operation supports the following data type combinations.

Src	Dst
f32	f32
bf16	bf16
f16	f16

Log

Log operation performs element-wise natural logarithm operation with given tensor, it applies following formula on every element of src tensor (the variable names follow the standard *Conventions*):

$$\mathrm{dst} = \log(\mathrm{src})$$

Operation Attributes

Log operation does not support any attribute.

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

Log operation supports the following data type combinations.

Src	Dst
f32	f32
f16	f16
bf16	bf16

LogSoftmax

LogSoftmax operation applies the $\log(softmax(src))$‘ function to an n-dimensional input Tensor. The formulation can be simplified as:

$$\mathrm{dst}_i = \log\left(\frac{e^{(\mathrm{src}_i)}}{\sum_j e^{(\mathrm{src}_j)}}\right)$$

Operation Attributes

Attribute Name	Description	Value Type	**Supported** Values	**Required or** Optional
`axis`	Represents the axis of which the LogSoftmax is calculated.	s64	Arbitrary s64 value (-1 in default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

LogSoftmax operation supports the following data type combinations.

Src	Dst
f32	f32
bf16	bf16
f16	f16

LogSoftmaxBackward

LogSoftmaxBackward operation computes gradient for LogSoftmax.

Operation Attributes

Attribute Name	Description	Value Type	**Supported** Values	**Required or** Optional
`axis`	Represents the axis of which the LogSoftmax is calculated.	s64	Arbitrary s64 value (-1 in default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`diff_dst`	Required
1	`dst`	Required

Outputs

Index	Argument Name	Required or Optional
0	`diff_src`	Required

Supported Data Types

LogSoftmaxBackward operation supports the following data type combinations.

Diff_dst	Dst	Diff_src
f32	f32	f32
bf16	bf16	bf16
f16	f16	f16

MatMul

MatMul operation computes the product of two tensors with optional bias addition The variable names follow the standard *Conventions*, typically taking 2D input tensors as an example, the formula is below:

$$\mathrm{dst}(m,n) = \sum_{k=0}^{K-1} (\mathrm{src}(m,k) \cdot \mathrm{weights}(k,n)) + \mathrm{bias}(m,n)$$

In the shape of a tensor, two right-most axes are interpreted as row and column dimensions of a matrix while all left-most axes (if present) are interpreted as batch dimensions. The operation supports broadcasting semantics for those batch dimensions. For example src can be broadcasted to weights if the corresponding dimension in src is 1 (and vice versa). Additionally, if ranks of src and weights are different, the tensor with a smaller rank will be *unsqueezed* from the left side of dimensions (inserting 1) to make sure two ranks matched.

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
transpose_a	Controls whether to transpose the last two dimensions of src	bool	True, False (default)	Optional
transpose_b	Controls whether to transpose the last two dimensions of weights	bool	True, False (default)	Optional

The above transpose attributes will not be in effect when rank of an input tensor is less than 2. For example, in library implementation 1D tensor is unsqueezed firstly before compilation. The rule is applied independently.

- For src tensor, the rule is defined like: `[d] -> [1, d]`.
- For weights tensor, the rule is defined like: `[d] -> [d, 1]`.

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required
1	`weights`	Required
2	`bias`	Optional

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

MatMul operation supports the following data type combinations.

Src	Weights	Bias	Dst
f32	f32	f32	f32
bf16	bf16	bf16	bf16
f16	f16	f16	f16

MaxPool

MaxPool operation performs the computation following the below formulas. Variable names follow the standard *Conventions*.

$$\mathrm{dst}(n, c, oh, ow) = \max_{kh,kw} \left(\mathrm{src}(n, c, oh \cdot SH + kh \cdot (DH + 1) - PH_L, ow \cdot SW + kw \cdot (DW + 1) - PW_L)\right)$$

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
strides	Controls the strides the window is moved	s64	A s64 list containing positive values	Required
pads_begin	Controls number of zeros to be add to the front/top/left of spatial dimensions, the attribute will be ignored when *auto_pad* attribute is specified to `same_upper`, `same_lower` or `valid`	s64	A s64 list containing non-negative values	Required
pads_end	Controls number of zeros to be add to the back/bottom/right of spatial dimensions, the attribute will be ignored when *auto_pad* attribute is specified to `same_upper`, `same_lower` or `valid`	s64	A s64 list containing non-negative values	Required
kernel	Size of pooling window	s64	A s64 list containing positive values	Required
dilations	Denotes the distance in width and height between elements.	s64	A s64 list containing positive values, a list of `1` s (default) means no dilation	Optional
rounding_type	Controls how to do rounding	string	`floor` (default), `ceil`	Optional
auto_pad	Controls how the paddings are calculated	string	`none` (default), `same_upper`, `same_lower`, `valid`	Optional
data_format	Controls how to interpret the shape of `src` and `dst`.	string	`NCX`, `NXC` (default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

MaxPool operation supports the following data type combinations.

Src	Dst
f32	f32
bf16	bf16
f16	f16

MaxPoolBackward

AvgPoolBackward operation accepts src tensor and $\mathrm{diff\_dst}$ tensor, and calculates $\mathrm{diff\_src}$ tensor.

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
strides	Controls the strides the window is moved	s64	A s64 list containing positive values	Required
pads_begin	Controls number of zeros to be add to the front/top/left of spatial dimensions, the attribute will be ignored when *auto_pad* attribute is specified to `same_upper`, `same_lower` or `valid`	s64	A s64 list containing non-negative values	Required
pads_end	Controls number of zeros to be add to the back/bottom/right of spatial dimensions, the attribute will be ignored when *auto_pad* attribute is specified to `same_upper`, `same_lower` or `valid`	s64	A s64 list containing non-negative values	Required
kernel	Size of pooling window	s64	A s64 list containing positive values	Required
dilations	Denotes the distance in width and height between elements.	s64	A s64 list containing positive values, a list of `1` s (default) means no dilation	Optional
auto_pad	Controls how the paddings are calculated	string	`none` (default), `same_upper`, `same_lower`, `valid`	Optional
data_format	Controls how to interpret the shape of `src` and `dst`.	string	`NCX`, `NXC` (default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required
1	`diff_dst`	Required

Outputs

Index	Argument Name	Required or Optional
0	`diff_src`	Required

Supported Data Types

MaxPoolBackward operation supports the following data type combinations.

Src	Diff_dst	Diff_src
f32	f32	f32
bf16	bf16	bf16
f16	f16	f16

Maximum

Maximum operation performs element-wise maximum operation with two given tensors applying multi-directional broadcast rules.

$$\mathrm{dst}(\overline{x}) = \max(\mathrm{src\_0}(\overline{x}), \mathrm{src\_1}(\overline{x}))$$

Operation Attributes

Attribute Name	Description	Value Type	**Supported** Values	**Required or** Optional
auto_broadcast	Specifies rules used for auto-broadcasting of src tensors	string	`none`, `numpy`, (default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src_0`	Required
1	`src_1`	Required

@note Both src shapes should match and no auto-broadcasting is allowed if `auto_broadcast` attributes is `none`. `src_0` and `src_1` shapes can be different and auto-broadcasting is allowed if `auto_broadcast` attributes is `numpy`. Broadcasting is performed according to auto_broadcast value.

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

Maximum operation supports the following data type combinations.

Src_0 / Src_1	Dst
f32	f32
bf16	bf16
f16	f16

Minimum

Minimum operation performs element-wise minimum operation with two given tensors applying multi-directional broadcast rules.

$$\mathrm{dst}(\overline{x}) = \min(\mathrm{src\_0}(\overline{x}), \mathrm{src\_1}(\overline{x}))$$

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
auto_broadcast	Specifies rules used for auto-broadcasting of src tensors	string	`none`, `numpy`, (default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src_0`	Required
1	`src_1`	Required

@note Both src shapes should match and no auto-broadcasting is allowed if `auto_broadcast` attributes is `none`. `src_0` and `src_1` shapes can be different and auto-broadcasting is allowed if `auto_broadcast` attributes is `numpy`. Broadcasting is performed according to auto_broadcast value.

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

Minimum operation supports the following data type combinations.

Src_0 / Src_1	Dst
f32	f32
bf16	bf16
f16	f16

Mish

Mish performs element-wise activation function on a given input tensor, based on the following mathematical formula:

$$\text{dst} = \text{src} * \tanh(\text{SoftPlus}(src)) = \text{src} * \tanh(\ln(1 + e^{\text{src}}))$$

Operation Attributes

Mish operation does not support any attribute.

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

Mish operation supports the following data type combinations.

Src	Dst
f32	f32
bf16	bf16
f16	f16

MishBackward

MishBackward operation computes gradient for Mish.

$$\text{dst} = \text{diff\_dst} * \frac{e^{\text{src}} * \omega}{\delta^2}$$

where

$$\omega = e^{3\,\text{src}} + 4 * e^{2\,\text{src}} + e^{\text{src}} * (4 * \text{src} + 6) + 4 * (\text{src} + 1)$$
$$\delta = e^{2\,\text{src}} + 2 * e^{\text{src}} + 2$$

Operation Attributes

MishBackward operation does not support any attribute.

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required
1	`diff_dst`	Required

Outputs

Index	Argument Name	Required or Optional
0	`diff_src`	Required

Supported Data Types

MishBackward operation supports the following data type combinations.

Src	Diff_dst	Diff_src
f32	f32	f32
f16	f16	f16
bf16	bf16	bf16

Multiply

Multiply operation performs element-wise multiply operation with two given tensors applying multi-directional broadcast rules.

$$\mathrm{dst}(\overline{x}) = \mathrm{src}\_0(\overline{x}) \times \mathrm{src}\_1(\overline{x}),$$

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
auto_broadcast	Specifies rules used for auto-broadcasting of src tensors	string	`none`, `numpy`, (default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src_0`	Required
1	`src_1`	Required

@note Both src shapes should match and no auto-broadcasting is allowed if `auto_broadcast` attributes is `none`. `src_0` and `src_1` shapes can be different and auto-broadcasting is allowed if `auto_broadcast` attributes is `numpy`. Broadcasting is performed according to auto_broadcast value.

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

Multiply operation supports the following data type combinations.

Src_0 / Src_1	Dst
f32	f32
bf16	bf16
f16	f16

Pow

General

Pow operation performs an element-wise power operation on a given input tensor with a single value attribute beta as its exponent. It is based on the following mathematical formula:

$$\mathrm{dst}_i = \mathrm{src}_i^{\beta}$$

Operation attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
beta	exponent, β in the formula.	f32	Arbitrary f32 value	Required

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

Pow operation supports the following data type combinations.

Src	Dst
f32	f32
bf16	bf16
f16	f16

PReLU

PReLU operation performs element-wise parametric ReLU operation on a given input tensor, based on the following mathematical formula:

$$\mathrm{dst} = \begin{cases} \mathrm{src} & \text{if } \mathrm{src} \geq 0 \\ \alpha \, \mathrm{src} & \text{if } \mathrm{src} < 0 \end{cases}$$

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
data_format	Denotes the data format of the input and output data.	string	`NCX`, `NXC` (default)	Optional
per_channel_broac	Denotes whether to apply per_channel broadcast when slope is 1D tensor.	bool	`false`, `true` (default)	Optional

Broadcasting Rules

Only slope tensor supports broadcasting semantics. Slope tensor is uni-directionally broadcasted to src if one of the following rules is met:

- 1: slope is 1D tensor and `per_channel_broadcast` is set to `true`, the length of slope tensor is equal to the length of src of channel dimension.
- 2: slope is 1D tensor and `per_channel_broadcast` is set to `false`, the length of slope tensor is equal to the length of src of the rightmost dimension.
- 3: slope is nD tensor, starting from the rightmost dimension, $input.shape_i == slope.shape_i$ or $slope.shape_i == 1$ or slope dimension i is empty.

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required
1	`slope`	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

PReLU operation supports the following data type combinations.

Src	Dst	Slope
f32	f32	f32
bf16	bf16	bf16
f16	f16	f16

PReLUBackward

PReLUBackward operation computes gradient for PReLU.

Operation Attributes

Attribute Name	Description	Value Type	**Supported** Values	**Required or** Optional
data_format	Denotes the data format of the input and output data.	string	`NCX`, `NXC` (default)	Optional

Broadcasting Rules

Only slope tensor supports broadcasting semantics. Slope tensor is uni-directionally broadcasted to src if one of the following rules is met:

- 1: PyTorch case: slope is 1D tensor and broadcast per channel, length of slope is equal to the length of src in channel dimension.
- 2: PyTorch case: slope is 1D tensor and broadcast per tensor, length of slope is equal to 1.
- 3: Tensorflow case: slope is nD tensor and its dimensions must be equal to the src dimensions starting from the second element: $ slope_shape = input_forward_shape[1:] $

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required
1	`slope`	Required
2	`diff_dst`	Required

Outputs

Index	Argument Name	Required or Optional
0	`diff_src`	Required
1	`diff_slope`	Required

Supported Data Types

PReLUBackward operation supports the following data type combinations.

Src	Slope	Diff_dst	Diff_src	Diff_slope
f32	f32	f32	f32	f32
bf16	bf16	bf16	bf16	bf16
f16	f16	f16	f16	f16

Quantize

Quantize operation converts a f32 tensor to a quantized (u8/s8) tensor. It supports both per-tensor and per-channel asymmetric linear quantization. Output data type is specified in output tensor data type. Rounding mode is library-implementation defined.

For per-tensor quantization:

$$\mathrm{dst}_i = round(\mathrm{src}_i / scale + zp)$$

For per-channel quantization, taking channel axis = 1 as an example:

$$\mathrm{dst}_{\dots,i,\dots,\dots} = round(\mathrm{src}_{\dots,i,\dots,\dots} / scale_i + zp_i), i \in [0, ic-1]$$

where ic is the number of channels.

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
qtype	Specifies which dequantization type is used	string	`per_tensor` (default), `per_channel`	Optional
axis	Specifies dimension on which per-channel dequantization is applied	s64	A s64 value in the range of [-r, r-1] where r = rank(src), 1 by default	Optional
scales	Scalings applied on the src data	f32	A f32 list (only contain one element if qtype is `per_tensor`)	Required
zps	Offset values that maps to float zero	s64	A s64 list (only contain one element if qtype is `per_tensor`)	Required

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

Quantize operation supports the following data type combinations.

Src	Dst
f32	s8, u8

@note This operation is to support *int8 quantization* model.

ReLU

ReLU applies following formula on every element of src tensor (the variable names follow the standard *Conventions*):

$$\text{dst} = \begin{cases} \text{src} & \text{if } \text{src} > 0 \\ 0 & \text{if } \text{src} \leq 0 \end{cases}$$

Operation Attributes

ReLU operation does not support any attribute.

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

ReLU operation supports the following data type combinations.

Src	Dst
f32	f32
bf16	bf16
f16	f16

ReLUBackward

ReLUBackward operation computes gradient for ReLU.

Operation Attributes

Attribute Name	Description	Value Type	**Supported** Values	**Required or** Optional
use_dst	If true, use dst of ReLU operation to calculate the gradient. Otherwise, use src.	bool	`true` (default), `false`	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src` / `dst`	Required
1	`diff_dst`	Required

Outputs

Index	Argument Name	Required or Optional
0	`diff_src`	Required

Supported Data Types

ReLUBackward operation supports the following data type combinations.

Src / Dst	Diff_dst	Diff_src
f32	f32	f32
f16	f16	f16
bf16	bf16	bf16

Reciprocal

Reciprocal operation is element-wise Power operation where exponent(power) equals to -1. Reciprocal of 0 is infinity.

$$\text{dst} = \begin{cases} \text{src}^{-1} & \text{if } \text{src} \neq 0 \\ \infty & \text{if } \text{src} = 0 \end{cases}$$

Operation Attributes

Reciprocal operation does not support any attribute.

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

Reciprocal operation supports the following data type combinations.

Src	Dst
f32	f32
bf16	bf16
f16	f16

ReduceL1

ReduceL1 operation performs the reduction with finding the L1 norm (sum of absolute values) on a given src data along dimensions specified by axes.

Take channel axis = 0 and keep_dims = True as an example:

$$\mathrm{dst}_{0,\cdots,\cdots} = \sum_i |\,\mathrm{src}_{i,\cdots,\cdots}\,|, i \in [0, channelNum - 1]$$

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
axes	Specify indices of src tensor, along which the reduction is performed. If axes is a list, reduce over all of them. If axes is empty, corresponds to the identity operation. If axes contains all dimensions of src tensor, a single reduction value is calculated for the entire src tensor. Exactly one of attribute *axes* and the second input tensor `axes` should be available.	s64	A s64 list values which is in the range of [-r,r-1] where r = rank(src). Empty list(default)	Optional
keep_dims	If set to `true` it holds axes that are used for reduction. For each such axes, dst dimension is equal to 1.	bool	`true`, `false` (default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required
1	`axes`	Optional

@note `axes` is an 1-D tensor specifying the axis along which the reduction is performed. 1D tensor of unique elements. The range of elements is [-r, r-1], where r is the rank of src tensor. Exactly one of attribute axes and the second input tensor axes should be available.

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

@note The result of ReduceL1 function applied to src tensor. shape[i] = shapeOf(data)[i] for all i that is not in the list of axes from the second input. For dimensions from axes, shape[i] == 1 if keep_dims == True, or i-th dimension is removed from the dst otherwise.

Supported Data Types

ReduceL1 operation supports the following data type combinations.

Src	Dst	Axes
f32	f32	s32
bf16	bf16	s32
f16	f16	s32

ReduceL2

ReduceL2 operation performs the reduction with finding the L2 norm (square root of sum of squares) on a given src data along dimensions specified by axes.

Take channel axis = 0 and keep_dims = True as an example:

$$\mathrm{dst}_{0,\cdots,\cdots} = \sqrt{\sum_{i} \mathrm{src}^2_{i,\cdots,\cdots}}, i \in [0, channelNum - 1]$$

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
axes	Specify indices of src tensor, along which the reduction is performed. If axes is a list, reduce over all of them. If axes is empty, corresponds to the identity operation. If axes contains all dimensions of src tensor, a single reduction value is calculated for the entire src tensor. Exactly one of attribute *axes* and the second input tensor `axes` should be available.	s64	A s64 list values which is in the range of [-r,r-1] where r = rank(src). Empty list(default)	Optional
keep_dims	If set to `true` it holds axes that are used for reduction. For each such axes, dst dimension is equal to 1.	bool	`true`, `false` (default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required
1	`axes`	Optional

@note `axes` is an 1-D tensor specifying the axis along which the reduction is performed. 1D tensor of unique elements. The range of elements is [-r, r-1], where r is the rank of src tensor. Exactly one of attribute axes and the second input tensor axes should be available.

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

@note The result of ReduceL2 function applied to src tensor. shape[i] = shapeOf(data)[i] for all i that is not in the list of axes from the second input. For dimensions from axes, shape[i] == 1 if keep_dims == True, or i-th dimension is removed from the dst otherwise.

Supported Data Types

ReduceL2 operation supports the following data type combinations.

Src	Dst	Axes
f32	f32	s32
bf16	bf16	s32
f16	f16	s32

ReduceMax

ReduceMax operation performs the reduction with finding the maximum value on a given src data along dimensions specified by axes.

Take channel axis = 0 and keep_dims = True as an example:

$$\mathrm{dst}_{0,\dots,\dots} = \max \mathrm{src}_{i,\dots,\dots,\dots}, i \in [0, channelNum - 1]$$

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
axes	Specify indices of src tensor, along which the reduction is performed. If axes is a list, reduce over all of them. If axes is empty, corresponds to the identity operation. If axes contains all dimensions of src tensor, a single reduction value is calculated for the entire src tensor. Exactly one of attribute *axes* and the second input tensor `axes` should be available.	s64	A s64 list values which is in the range of [-r,r-1] where r = rank(src). Empty list(default)	Optional
keep_dims	If set to `true` it holds axes that are used for reduction. For each such axes, dst dimension is equal to 1.	bool	`true`, `false` (default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required
1	`axes`	Optional

@note `axes` is an 1-D tensor specifying the axis along which the reduction is performed. 1D tensor of unique elements. The range of elements is [-r, r-1], where r is the rank of src tensor. Exactly one of attribute axes and the second input tensor axes should be available.

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

@note The result of ReduceMax function applied to src tensor. shape[i] = shapeOf(data)[i] for all i that is not in the list of axes from the second input. For dimensions from axes, shape[i] == 1 if keep_dims == True, or i-th dimension is removed from the dst otherwise.

Supported Data Types

ReduceMax operation supports the following data type combinations.

Src	Dst	Axes
f32	f32	s32
bf16	bf16	s32
f16	f16	s32

ReduceMean

ReduceMean operation performs the reduction with finding the arithmetic mean on a given src data along dimensions specified by axes.

Take channel axis = 0 and keep_dims = True as an example:

$$\mathrm{dst}_{0,\dots,\dots} = \frac{1}{channelNum} \cdot \sum_{i} \mathrm{src}_{i,\dots,\dots}, i \in [0, channelNum - 1]$$

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
axes	Specify indices of src tensor, along which the reduction is performed. If axes is a list, reduce over all of them. If axes is empty, corresponds to the identity operation. If axes contains all dimensions of src tensor, a single reduction value is calculated for the entire src tensor. Exactly one of attribute *axes* and the second input tensor `axes` should be available.	s64	A s64 list values which is in the range of [-r,r-1] where r = rank(src). Empty list(default)	Optional
keep_dims	If set to `true` it holds axes that are used for reduction. For each such axes, dst dimension is equal to 1.	bool	`true`, `false` (default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required
1	`axes`	Optional

@note `axes` is an 1-D tensor specifying the axis along which the reduction is performed. 1D tensor of unique elements. The range of elements is [-r, r-1], where r is the rank of src tensor. Exactly one of attribute axes and the second input tensor axes should be available.

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

@note The result of ReduceMean function applied to src tensor. shape[i] = shapeOf(data)[i] for all i that is not in the list of axes from the second input. For dimensions from axes, shape[i] == 1 if keep_dims == True, or i-th dimension is removed from the dst otherwise.

Supported Data Types

ReduceMean operation supports the following data type combinations.

Src	Dst	Axes
f32	f32	s32
bf16	bf16	s32
f16	f16	s32

ReduceMin

ReduceMin operation performs the reduction with finding the minimum value on a given src data along dimensions specified by axes.

Take channel axis = 0 and keep_dims = True as an example:

$$\mathrm{dst}_{0,\dots,\dots} = \min \mathrm{src}_{i,\dots,\dots,}, i \in [0, channelNum - 1]$$

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
axes	Specify indices of src tensor, along which the reduction is performed. If axes is a list, reduce over all of them. If axes is empty, corresponds to the identity operation. If axes contains all dimensions of src tensor, a single reduction value is calculated for the entire src tensor. Exactly one of attribute *axes* and the second input tensor `axes` should be available.	s64	A s64 list values which is in the range of [-r,r-1] where r = rank(src). Empty list(default)	Optional
keep_dims	If set to `true` it holds axes that are used for reduction. For each such axes, dst dimension is equal to 1.	bool	`true`, `false` (default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required
1	`axes`	Optional

@note `axes` is a 1-D tensor specifying the axis along which the reduction is performed. 1D tensor of unique elements. The range of elements is [-r, r-1], where r is the rank of src tensor. Exactly one of attribute axes and the second input tensor axes should be available.

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

@note The result of ReduceMin function applied to src tensor. shape[i] = shapeOf[data](i) for all i that is not in the list of axes from the second input. For dimensions from axes, shape[i] == 1 if keep_dims == True, or i-th dimension is removed from the dst otherwise.

Supported Data Types

ReduceMin operation supports the following data type combinations.

Src	Dst	Axes
f32	f32	s32
bf16	bf16	s32
f16	f16	s32

ReduceProd

ReduceProd operation performs the reduction with multiplication on a given src data along dimensions specified by axes.

Take channel axis = 0 and keep_dims = True as an example:

$$\mathrm{dst}_{0,\cdots,\cdots} = \prod_{i} \mathrm{src}_{i,\cdots,\cdots}, i \in [0, channelNum - 1]$$

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
axes	Specify indices of src tensor, along which the reduction is performed. If axes is a list, reduce over all of them. If axes is empty, corresponds to the identity operation. If axes contains all dimensions of src tensor, a single reduction value is calculated for the entire src tensor. Exactly one of attribute *axes* and the second input tensor `axes` should be available.	s64	A s64 list values which is in the range of [-r,r-1] where r = rank(src). Empty list(default)	Optional
keep_dims	If set to `true` it holds axes that are used for reduction. For each such axes, dst dimension is equal to 1.	bool	`true`, `false` (default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required
1	`axes`	Optional

@note `axes` is an 1-D tensor specifying the axis along which the reduction is performed. 1D tensor of unique elements. The range of elements is [-r, r-1], where r is the rank of src tensor. Exactly one of attribute axes and the second input tensor axes should be available.

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

@note The result of ReduceProd function applied to src tensor. shape[i] = shapeOf(data)[i] for all i that is not in the list of axes from the second input. For dimensions from axes, shape[i] == 1 if keep_dims == True, or i-th dimension is removed from the dst otherwise.

Supported Data Types

ReduceProd operation supports the following data type combinations.

Src	Dst	Axes
f32	f32	s32
bf16	bf16	s32
f16	f16	s32

ReduceSum

ReduceSum operation performs the reduction with addition on a given src data along dimensions specified by axes.

Take channel axis = 0 and keep_dims = True as an example:

$$\mathrm{dst}_{0,\cdots,\cdots} = \sum_{i} \mathrm{src}_{i,\cdots,\cdots}, i \in [0, channelNum - 1]$$

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
axes	Specify indices of src tensor, along which the reduction is performed. If axes is a list, reduce over all of them. If axes is empty, corresponds to the identity operation. If axes contains all dimensions of src tensor, a single reduction value is calculated for the entire src tensor. Exactly one of attribute *axes* and the second input tensor `axes` should be available.	s64	A s64 list values which is in the range of [-r,r-1] where r = rank(src). Empty list(default)	Optional
keep_dims	If set to `true` it holds axes that are used for reduction. For each such axes, dst dimension is equal to 1.	bool	`true`, `false` (default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required
1	`axes`	Optional

@note `axes` is an 1-D tensor specifying the axis along which the reduction is performed. 1D tensor of unique elements. The range of elements is [-r, r-1], where r is the rank of src tensor. Exactly one of attribute axes and the second input tensor axes should be available.

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

@note The result of ReduceSum function applied to src tensor. shape[i] = shapeOf(data)[i] for all i that is not in the list of axes from the second input. For dimensions from axes, shape[i] == 1 if keep_dims == True, or i-th dimension is removed from the dst otherwise.

Supported Data Types

ReduceSum operation supports the following data type combinations.

Src	Dst	Axes
f32	f32	s32
bf16	bf16	s32
f16	f16	s32

Reorder

Reorder operation converts src tensor to dst tensor with different layout. It supports the conversion between:

- two different opaque layouts
- two different strided layouts
- one strided layout and another opaque layout

Reorder operation does not support layout conversion cross different backends or different engines. Unlike *reorder primitive :ref: `reorder-label*, Reorder operation cannot be used to cast the data type from src to dst. Please check the usage of *TypeCast :ref: `op_typecast-label* and *Dequantize :ref: `op_dequantize-label* operation.

Operation Attributes

Reorder operation does not support any attribute.

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

Reorder operation supports the following data type combinations.

Src	Dst
f32	f32
bf16	bf16
f16	f16

Round

Round operation rounds the values of a tensor to the nearest integer, element-wise.

Operation Attributes

Round operation does not support any attribute.

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

Round operation supports the following data type combinations.

Src	Dst
f32	f32
f16	f16
bf16	bf16

Select

Select operation returns a tensor filled with the elements from the second or the third input, depending on the condition (the first input) value.

$$\mathrm{dst}[i] = cond[i] ? \mathrm{src\_0}[i] : \mathrm{src\_1}[i]$$

Broadcasting is supported.

If the auto_broadcast attribute is not none, the select operation takes a two-step broadcast before performing the selection:

- **Step 1**: Input tensors src_0 and src_1 are broadcasted to dst_shape according to the Numpy broadcast rules.
- **Step 2**: Then, the cond tensor will be one-way broadcasted to the dst_shape of broadcasted src_0 and src_1. To be more specific, we align the two shapes to the right and compare them from right to left. Each dimension should be either equal or the dimension of cond should be 1.
- **example**:
  - cond={4, 5}, dst_shape={2, 3, 4, 5} => dst = {2, 3, 4, 5}
  - cond={3, 1, 5}, dst_shape={2, 3, 4, 5} => dst = {2, 3, 4, 5}
  - cond={3,5}, dst_shape={2, 3, 4, 5} => dst = invalid_shape

Operation Attributes

Attribute Name	Description	Value Type	**Supported** Values	**Required or** Optional
auto_broadcast	Specifies rules used for auto-broadcasting of src tensors	string	`none`, `numpy`, (default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`cond`	Required
1	`src_0`	Required
2	`src_1`	Required

@note All input shapes should match and no broadcasting is allowed if the *auto_broadcast* attribute is set to *none*, or can be broadcasted according to the broadcasting rules mentioned above if *auto_broadcast* attribute set to *numpy*.

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

Select operation supports the following data type combinations.

Cond	Src_0	Src_1	Dst
boolean	f32	f32	f32
boolean	bf16	bf16	bf16
boolean	f16	f16	f16

Sigmoid

Sigmoid operation applies following formula on every element of src tensor (the variable names follow the standard *Conventions*):

$$\mathrm{dst} = \frac{1}{1 + e^{-\mathrm{src}}}$$

Operation Attributes

Sigmoid operation does not support any attribute.

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

Sigmoid operation supports the following data type combinations.

Src	Dst
f32	f32
bf16	bf16
f16	f16

SigmoidBackward

SigmoidBackward operation computes gradient for Sigmoid.

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
use_dst	If true, use dst of Sigmoid operation to calculate the gradient. Otherwise, use src.	bool	`true` (default), `false`	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src` / `dst`	Required
1	`diff_dst`	Required

Outputs

Index	Argument Name	Required or Optional
0	`diff_src`	Required

Supported Data Types

SigmoidBackward operation supports the following data type combinations.

Src / Dst	Diff_dst	Diff_src
f32	f32	f32
f16	f16	f16
bf16	bf16	bf16

SoftPlus

SoftPlus operation applies following formula on every element of src tensor (the variable names follow the standard *Conventions*):

$$\mathrm{dst} = 1/\,\mathrm{beta}\,\ln(e^{\mathrm{beta}\,*\,\mathrm{src}} + 1.0)$$

Operation Attributes

Attribute Name	Description	Value Type	**Supported** Values	**Required or** Optional
beta	Value for the SoftPlus formulation.	s64	Arbitrary s64 value (1 in default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

SoftPlus operation supports the following data type combinations.

Src	Dst
f32	f32
bf16	bf16
f16	f16

SoftPlusBackward

SoftPlusBackward operation computes gradient for SoftPlus.

Operation Attributes

Attribute Name	Description	Value Type	**Supported** Values	**Required or** Optional
beta	Value for the SoftPlus formulation.	s64	Arbitrary s64 value (1 in default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required
1	`diff_dst`	Required

Outputs

Index	Argument Name	Required or Optional
0	`diff_src`	Required

Supported Data Types

SoftPlusBackward operation supports the following data type combinations.

Src	Diff_dst	Diff_src
f32	f32	f32
bf16	bf16	bf16
f16	f16	f16

SoftMax

SoftMax operation applies the following formula on every element of src tensor (the variable names follow the standard *Conventions*):

$$\mathrm{dst}_i = \frac{e^{(\mathrm{src}_i)}}{\sum_{j=1}^{C} e^{\mathrm{src}_j}}$$

where $ C $ is a size of tensor along axis dimension.

Operation Attributes

Attribute Name	Description	Value Type	**Supported** Values	**Required or** Optional
axis	Represents the axis from which the SoftMax is calculated.	s64	Arbitrary s64 value (1 in default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

SoftMax operation supports the following data type combinations.

Src	Dst
f32	f32
bf16	bf16
f16	f16

SoftMaxBackward

SoftMaxBackward operation computes gradient for SoftMax.

Operation Attributes

Attribute Name	Description	Value Type	**Supported** Values	**Required or** Optional
axis	Represents the axis from which the SoftMax is calculated.	s64	Arbitrary s64 value (1 in default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`diff_dst`	Required
1	`dst`	Required

Outputs

Index	Argument Name	Required or Optional
0	`diff_src`	Required

Supported Data Types

SoftMaxBackward operation supports the following data type combinations.

Dst	Diff_dst	Diff_src
f32	f32	f32
bf16	bf16	bf16
f16	f16	f16

Sqrt

Sqrt operation performs element-wise square root operation with given tensor.

Operation Attributes

Sqrt operation does not support any attribute.

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

Sqrt operation supports the following data type combinations.

Src	Dst
f32	f32
f16	f16
bf16	bf16

SqrtBackward

SqrtBackward operation computes gradient for Sqrt.

Operation Attributes

Attribute Name	Description	Value Type	**Supported** Values	**Required or** Optional
use_dst	If true, use `dst` of Sqrt operation to calculate the gradient. Otherwise, use `src`.	bool	`true` (default), `false`	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src` / `dst`	Required
1	`diff_dst`	Required

Outputs

Index	Argument Name	Required or Optional
0	`diff_src`	Required

Supported Data Types

SqrtBackward operation supports the following data type combinations.

Src / Dst	Diff_dst	Diff_src
f32	f32	f32
f16	f16	f16
bf16	bf16	bf16

Square

Square operation performs element-wise square operation with given tensor.

Operation Attributes

Square operation does not support any attribute.

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

Square operation supports the following data type combinations.

Src	Dst
f32	f32
f16	f16
bf16	bf16

SquaredDifference

SquaredDifference operation performs element-wise subtraction operation with two given tensors applying multi-directional broadcast rules, after that each result of the subtraction is squared.

Before performing arithmetic operation, src_0 and src_1 are broadcasted if their shapes are different and `auto_broadcast` attributes is not `none`. Broadcasting is performed according to `auto_broadcast` value. After broadcasting SquaredDifference does the following with the input tensors:

$$\mathrm{dst}_i = (\mathrm{src}\_0_i - \mathrm{src}\_1_i)^2$$

Operation Attributes

Attribute Name	Description	Value Type	**Supported** Values	**Required or** Optional
auto_broadcast	Specifies rules used for auto-broadcasting of src tensors	string	`none`, `numpy`, (default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src_0`	Required
1	`src_1`	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

SquaredDifference operation supports the following data type combinations.

Src_0	Src_1	Dst
f32	f32	f32
bf16	bf16	bf16
f16	f16	f16

StaticReshape

StaticReshape operation changes dimensions of src tensor according to the specified shape. The volume of src is equal to dst, where volume is the product of dimensions. dst may have a different memory layout from src. StaticReshape operation is not guaranteed to return a view or a copy of src when dst is in-placed with the src. StaticReshape can be used where if shape is stored in a constant node or available during graph building stage. Then shape can be passed via `shape` attribute.

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
shape	Specifies rules used for auto-broadcasting of src tensors	string	`none`, `numpy` (default)	Required
special_zero	Controls how zero values in shape are interpreted	bool	`true`, `false`	Required

@note *shape*: dimension `-1` means that this dimension is calculated to keep the same overall elements count as the src tensor. That case that more than one `-1` in the shape is not supported.

@note *special_zero*: if false, `0` in the shape is interpreted as-is (for example a zero-dimension tensor); if true, then all `0`s in shape implies the copying of corresponding dimensions from src into dst.

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

StaticReshape operation supports the following data type combinations.

Src	Dst
f32	f32
bf16	bf16
f16	f16

StaticTranspose

StaticTranspose operation rearranges the dimensions of src with respect to the permutation described by `order`. dst may have a different memory layout from src. StaticTranspose operation is not guaranteed to return a view or a copy of src when dst is in-placed with the src.

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
order	Specifies permutation to be applied on src	s64	A s64 list containing the element in the range of [-N, N-1], negative value means counting from last axis	Required

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

StaticTranspose operation supports the following data type combinations.

Src	Dst
f32	f32
bf16	bf16
f16	f16

Subtract

Subtract operation performs element-wise subtraction operation with two given tensors applying multi-directional broadcast rules.

$$\mathrm{dst}(\overline{x}) = \mathrm{src\_0}(\overline{x}) - \mathrm{src\_1}(\overline{x})$$

Operation Attributes

Attribute Name	Description	Value Type	**Supported** Values	**Required or** Optional
auto_broadcast	Specifies rules used for auto-broadcasting of src tensors	string	`none`, `numpy`, (default)	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src_0`	Required
1	`src_1`	Required

@note Both src shapes should match and no auto-broadcasting is allowed if `auto_broadcast` attributes is `none`. `src_0` and `src_1` shapes can be different and auto-broadcasting is allowed if `auto_broadcast` attributes is `numpy`. Broadcasting is performed according to auto_broadcast value.

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

Substract operation supports the following data type combinations.

Src_0 / Src_1	Dst
f32	f32
bf16	bf16
f16	f16

Tanh

Tanh operation applies following formula on every element of src tensor (the variable names follow the standard *Conventions*):

$$\text{dst} = \tanh(\text{src})$$

Operation Attributes

Tanh operation does not support any attribute.

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

Tanh operation supports the following data type combinations.

Src	Dst
f32	f32
bf16	bf16
f16	f16

TanhBackward

TanhBackward operation computes gradient for Tanh.

Operation Attributes

Attribute Name	Description	Value Type	Supported Values	Required or Optional
use_dst	If true, use dst of Tanh operation to calculate the gradient. Otherwise, use src.	bool	`true` (default), `false`	Optional

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src` / `dst`	Required
1	`diff_dst`	Required

Outputs

Index	Argument Name	Required or Optional
0	`diff_src`	Required

Supported Data Types

TanhBackward operation supports the following data type combinations.

Src / Dst	Diff_dst	Diff_src
f32	f32	f32
f16	f16	f16
bf16	bf16	bf16

TypeCast

TypeCast operation performs element-wise cast from input data type to the data type given by output tensor. It requires that src and dst have the same shape and layout. Rounding to nearest even will be used during cast.

Operation Attributes

TypeCast operation does not support any attribute.

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Required

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Required

Supported Data Types

TypeCast operation supports the following data type combinations.

Src	Dst
bf16, f16	f32
f32	bf16, f16

@note This operation is to support *mixed precision* computation.

Wildcard

Wildcard operation is used to represent any compute logic and help construct graph. Typically the operation can support mapping any framework ops which are not supported by the library implementation. It's useful to make the graph completed or connected.

Operation Attributes

Wildcard operation does not support any attribute.

Execution Arguments

The inputs and outputs must be provided according to the below index order when constructing an operation.

Inputs

Index	Argument Name	Required or Optional
0	`src`	Optional

Outputs

Index	Argument Name	Required or Optional
0	`dst`	Optional

@note WildCard operation can accept arbitrary number of inputs or outputs.

Supported Data Types

Wildcard operation supports arbitrary data type combinations.

4.7 Open Source Implementation

Intel has published an open source implementation with the Apache license.

4.8 Implementation Notes

This specification provides high-level descriptions for oneDNN operations and does not cover all the implementation-specific details of the open source implementation. Specifically, it does not cover highly-optimized memory formats and integration with profiling tools, etc. This is done intentionally to improve specification portability. Code that uses API defined in this specification is expected to be portable across open source implementation and any potential other implementations of this specification to a reasonable extent.

In the future this section will be extended with more details on how different implementations of this specification should cooperate and co-exist.

4.9 Testing

Intel's binary distribution of oneDNN contains example code that you can be used to test library functionality.

The open source implementation includes a comprehensive test suite. Consult the README for directions.

CHAPTER

FIVE

ONECCL

5.1 Introduction

The oneAPI Collective Communications Library (oneCCL) provides primitives for the communication patterns that occur in deep learning applications. oneCCL supports both scale-up for platforms with multiple oneAPI devices and scale-out for clusters with multiple compute nodes.

oneCCL supports the following communication patterns used in deep learning (DL) algorithms:

- allgatherv
- allreduce
- alltoallv
- broadcast
- reduce
- reduce_scatter

oneCCL exposes controls over additional optimizations and capabilities such as:

- Prioritization for communication operations
- Persistent communication operations (enables decoupling one-time initialization and repetitive execution)

5.2 Namespaces

This section describes the oneCCL namespace conventions.

5.2.1 oneapi::ccl namespace

The `oneapi::ccl` namespace shall contains public identifiers defined by the library.

5.2.2 ccl namespace

The alternative `ccl` namespace shall be considered an alias for the `oneapi::ccl` namespace.

5.3 Current Version of this oneCCL Specification

This is the oneCCL specification version 1.0.

5.4 Definitions

5.4.1 oneCCL Concepts

oneCCL specification defines the following list of concepts:

- *Device*
- *Context*
- *Key-Value Store*
- *Communicator*
- *Stream*
- *Event*
- *Operation Attributes*

Device

Note: Here and below, a native device/context/stream/event are defined in the scope of SYCL device runtime

```
using native_device_type = sycl::device;
using native_context_type = sycl::context;
using native_stream_type = sycl::queue;
using native_event_type = sycl::event;
```

oneCCL specification defines `device` as an abstraction of a computational device: a CPU, a specific GPU card in the system, or any other device participating in a communication operation. `device` corresponds to the communicator's rank (addressable entity in a communication operation).

oneCCL specification defines the way to create an instance of the `device` class with a native object (`native_device_type`) and without a native object (corresponds to the host).

Creating a new device object:

```
device ccl::create_device(native_device_type& native_device);

device ccl::create_device();
```

native_device
: the existing native device object

return device
a device object

`device` class shall provide ability to retrieve a native object.

Retrieving a native device object:

```
native_device_type device::get_native();
```

return native_device_type

a native device object
shall throw exception if a `device` object does not wrap the native object

Context

oneCCL specification defines `context` as an abstraction of a computational devices context that is responsible for managing resources and for executing of communication operations on one or more devices specified in the context.

oneCCL specification defines the way to create an instance of the `context` class with a native object (`native_context_type`) and without a native object.

Creating a new context object:

```
context ccl::create_context(native_context_type& native_context);

context ccl::create_context();
```

native_context
the existing native context object

return context
a context object

`context` class shall provide ability to retrieve a native object.

Retrieving a native context object:

```
native_context_type context::get_native();
```

return native_context_type

a native context object
shall throw exception if a `context` object does not wrap the native object

Key-Value Store

`kvs_interface` defines the key-value store (KVS) interface to be used to establish connection between ranks during the creation of oneCCL communicator. The interface shall include blocking `get` and `set` methods.

Getting a record from the key-value store:

```
virtual vector_class<char> kvs_interface::get(
    const string_class& key) = 0;
```

key
the key of value to be retrieved

return vector_class<char>
the value associated with the given key

Note: `get` operation with a non-existing key shall return empty result

---

Saving a record in the key-value store:

```
void kvs_interface::set(
    const string_class& key,
    const vector_class<char>& data) = 0;
```

key
the key at which the value should be stored

data
the value that should be associated with the given key

Note: `set` operation with empty `data` shall remove a record from the key-value store

---

oneCCL specification defines `kvs` class as a built-in KVS provided by oneCCL.

```
class kvs : public kvs_interface {

public:

static constexpr size_t address_max_size = 256;
using address_type = array_class<char, address_max_size>;

~kvs() override;

address_type get_address() const;

vector_class<char> get(
    const string_class& key) override;

void set(
    const string_class& key,
    const vector_class<char>& data) override;

}
```

Retrieving an address of built-in key-value store:

```
kvs::address_type kvs::get_address() const;
```

return kvs::address_type

the address of the key-value store
should be retrieved from the main built-in KVS and distributed to other processes for the built-in KVS creation

Creating a main built-in key-value store. Its address should be distributed using an out-of-band communication mechanism and be used to create key-value stores on other ranks:

```
shared_ptr_class<kvs> ccl::create_main_kvs();
```

return shared_ptr_class<kvs>
the main key-value store object

Creating a new key-value store from main kvs address:

```
shared_ptr_class<kvs> ccl::create_kvs(const kvs::address_type& addr);
```

addr
the address of the main kvs

return shared_ptr_class<kvs>
key-value store object

Communicator

oneCCL specification defines `communicator` class that describes a group of communicating ranks, where a rank is an addressable entity in a communication operation and corresponds to single oneCCL device.

`communicator` defines communication operations on memory buffers between homogeneous oneCCL devices, that is, all oneCCL devices either wrap native device objects of the same type (for example CPUs only or GPUs only) or do not wrap native objects.

Each process may correspond to multiple ranks.

Note: Support for multiple ranks per process is optional

Creating a new communicator(s) with user-supplied communicator size, rank-to-device mapping/rank, context and kvs:

Note: If `device` and `context` objects are omitted, then they are created with `ccl::create_device()` and `ccl::create_context()` functions without native objects

---

```
vector_class<communicator> ccl::create_communicators(
    int size,
    const map_class<int, device>& rank_device_map,
    const context& context,
    shared_ptr_class<kvs_interface> kvs);

communicator ccl::create_communicator(
    int size,
    int rank,
    shared_ptr_class<kvs_interface> kvs);
```

size
user-supplied total number of ranks

rank_device_map
user-supplied mapping of local ranks on devices

rank
user-supplied local rank

context
device context

kvs
key-value store for ranks wire-up

return vector_class<communicator> / communicator
a vector of communicator objects / a communicator object

`communicator` shall provide methods to retrieve the rank, the device, and the context that correspond to the communicator object as well as the total number of ranks in the communicator.

Retrieving the rank in a communicator:

```
int communicator::rank() const;
```

return int
the rank that corresponds to the communicator object

Retrieving the total number of ranks in a communicator:

```
int communicator::size() const;
```

return int
the total number of the ranks

Retrieving an underlying device, which was used as communicator construction argument:

```
device communicator::get_device() const;
```

return device
the device that corresponds to the communicator object

Retrieving an underlying context, which was used as communicator construction argument:

```
context communicator::get_context() const;
```

return context
the context that corresponds to the communicator object

Note: See also: *Collective Operations*

Stream

oneCCL specification defines `stream` as an abstraction that encapsulates execution context for `communicator` communication operations.

Stream shall be passed to `communicator` communication operation.

oneCCL specification defines the way to create an instance of the `stream` class with a native object (`native_stream_type`) and without a native object.

Creating a new stream object:

```
stream ccl::create_stream(native_stream_type& native_stream);

stream ccl::create_stream();
```

native_stream
the existing native stream object

return `stream`
a stream object

`stream` class shall provide ability to retrieve a native object.

Retrieving a native stream object:

```
native_stream_type stream::get_native();
```

return `native_stream_type`

a native stream object

shall throw exception if a `stream` object does not wrap the native object

Event

oneCCL specification defines `event` as an abstraction that encapsulates synchronization context for `communicator` communication operations.

Each communication operation of oneCCL shall return an event object for tracking the operation's progress. A vector of events may be passed to the `communicator` communication operation to designate input dependencies for the operation.

Note: Support for handling of input events is optional

---

oneCCL specification defines the way to create an instance of the `event` class with a native object (`native_event_type`).

Creating a new event object:

```
event ccl::create_event(native_event_type& native_event);
```

native_event
the existing native event object

return `event`
an event object

`event` class shall provide ability to retrieve a native object.

Retrieving a native event object:

```
native_event_type event::get_native();
```

return `native_event_type`

a native event object

shall throw exception if an `event` object does not wrap the native object

---

Note: See also: *Operation Progress Tracking*

Operation Attributes

Communication operation behavior may be controlled through operation attributes.

Operation Attributes

5.4.2 Communication Operations

This section covers communication operations defined by oneCCL specification.

Datatypes

oneCCL specification defines the following datatypes that may be used for communication operations:

```
enum class datatype : int
{
    int8            = /* unspecified */,
    uint8           = /* unspecified */,
    int16           = /* unspecified */,
    uint16          = /* unspecified */,
    int32           = /* unspecified */,
    uint32          = /* unspecified */,
    int64           = /* unspecified */,
    uint64          = /* unspecified */,

    float16         = /* unspecified */,
    float32         = /* unspecified */,
    float64         = /* unspecified */,
    bfloat16        = /* unspecified */,

    last_predefined = /* unspecified, equal to the largest of all the values above */
};
```

datatype::int8
: 8 bits signed integer

datatype::uint8
: 8 bits unsigned integer

datatype::int16
: 16 bits signed integer

datatype::uint16
: 16 bits unsigned integer

datatype::int32
: 32 bits signed integer

datatype::uint32
: 32 bits unsigned integer

datatype::int64
: 64 bits signed integer

datatype::uint64
: 64 bits unsigned integer

datatype::float16
16-bit/half-precision floating point

datatype::float32
32-bit/single-precision floating point

datatype::float64
64-bit/double-precision floating point

datatype::bfloat16
non-standard 16-bit floating point with 7-bit mantissa

Note: Support for `datatype::float16` is optional

Custom Datatypes

oneCCL specification defines the way to register and deregister a custom datatype using the `datatype_attr` attribute object.

The list of identifiers that may be used to fill an attribute object:

```
enum class datatype_attr_id {
    size = /* unspecified */
};
```

datatype_attr_id::size
the size of the datatype in bytes

Creating a datatype attribute object, which may used to register custom datatype:

```
datatype_attr ccl::create_datatype_attr();
```

return datatype_attr
an object containing attributes for the custom datatype

Registering a custom datatype to be used in communication operations:

```
datatype ccl::register_datatype(const datatype_attr& attr);
```

attr
the datatype's attributes

return datatype
the handle for the custom datatype

Deregistering a custom datatype:

```
void ccl::deregister_datatype(datatype dtype);
```

dtype
the handle for the custom datatype

Retrieving a datatype size in bytes:

```
size_t ccl::get_datatype_size(datatype dtype);
```

dtype
: the datatype's handle

return `size_t`
: datatype size in bytes

Reductions

oneCCL specification defines the following reduction operations for *Allreduce*, *Reduce* and *ReduceScatter* collective operations:

```
enum class reduction
{
    sum    = /* unspecified */,
    prod   = /* unspecified */,
    min    = /* unspecified */,
    max    = /* unspecified */,
    custom = /* unspecified */
};
```

reduction::sum
: elementwise summation

reduction::prod
: elementwise multiplication

reduction::min
: elementwise min

reduction::max
: elementwise max

reduction::custom

: specify user-defined reduction operation

: the actual reduction function must be passed through `reduction_fn` operation attribute

Operation Attributes

Collective Operations

oneCCL specification defines the following collective communication operations:

- *Allgatherv*
- *Allreduce*
- *Alltoallv*
- *Barrier*
- *Broadcast*
- *Reduce*
- *ReduceScatter*
- *PointToPoint*

These operations are collective, meaning that all participants (ranks) of oneCCL communicator should make a call. The order of collective operation calls should be the same across all ranks.

`communicator` shall provide the ability to perform communication operations either on host or device memory buffers depending on the `device` used to create the communicator. Additionally, communication operations shall accept an execution context (stream) and may accept a vector of events that the communication operation should depend on, that is, input dependencies. The output `event` object shall provide the ability to track the progress of the operation.

Note: Support for handling of input events is optional

`BufferType` is used below to define the C++ type of elements in data buffers (`buf`, `send_buf` and `recv_buf`) of communication operations. At least the following types shall be supported: `[u]int{8/16/32/64}_t`, `float`, `double`. The explicit `datatype` parameter shall be used to enable data types which cannot be inferred from the function arguments.

Note: See also: *Custom Datatypes*

The communication operation accepts a `stream` object. If a communicator is created from `native_device_type`, then the stream shall translate to `native_stream_type` created from the corresponding device.

The communication operation may accept attribute object. If that parameter is missed, then the default attribute object is used (default_<operation_name>_attr). The default attribute object shall be provided by the library.

Note: See also: *Operation Attributes*

If the arguments provided to a communication operation call do not comply to the requirements of the operation, the behavior is undefined unless it is specified otherwise.

Allgatherv

Allgatherv is a collective communication operation that collects data from all the ranks within a communicator into a single buffer. Different ranks may contribute segments of different sizes. The resulting data in the output buffer must be the same for each rank.

Allgatherv is in place when `send_buf == recv_buf + rank_offset`, where `rank_offset = sum (recv_counts[i])`, for all `i < rank`.

```
template<class BufferType>
event ccl::allgatherv(const BufferType* send_buf,
                      size_t send_count,
                      BufferType* recv_buf,
                      const vector_class<size_t>& recv_counts,
                      const communicator& comm,
                      const stream& stream,
                      const allgatherv_attr& attr = default_allgatherv_attr,
                      const vector_class<event>& deps = {});

event ccl::allgatherv(const void* send_buf,
                      size_t send_count,
                      void* recv_buf,
                      const vector_class<size_t>& recv_counts,
```

(continues on next page)

(continued from previous page)

```
                        datatype dtype,
                        const communicator& comm,
                        const stream& stream,
                        const allgatherv_attr& attr = default_allgatherv_attr,
                        const vector_class<event>& deps = {});
```

send_buf
: the buffer with `send_count` elements of `BufferType` that stores local data to be gathered

send_count
: the number of elements of type `BufferType` in `send_buf`

recv_buf [out]
: the buffer to store the gathered result, must be large enough to hold values from all ranks

recv_counts
: an array with the number of elements of type `BufferType` to be received from each rank
: the array's size must be equal to the number of ranks
: the values in the array are expected to be the same for all ranks
: the value at the position of the caller's rank must be equal to `send_count`

dtype
: the datatype of elements in `send_buf` and `recv_buf`
: must be skipped if `BufferType` can be inferred
: otherwise must be passed explicitly

comm
: the communicator that defines a group of ranks for the operation

stream
: the stream associated with the operation

attr
: optional attributes to customize the operation

deps
: an optional vector of the events that the operation should depend on

return event
: an object to track the progress of the operation

Allreduce

Allreduce is a collective communication operation that performs the global reduction operation on values from all ranks of communicator and distributes the result back to all ranks.

Allreduce is in-place when `send_buf == recv_buf`.

```
template <class BufferType>
event ccl::allreduce(const BufferType* send_buf,
                     BufferType* recv_buf,
                     size_t count,
                     reduction rtype,
                     const communicator& comm,
```

(continues on next page)

(continued from previous page)

```
                      const stream& stream,
                      const allreduce_attr& attr = default_allreduce_attr,
                      const vector_class<event>& deps = {});

event ccl::allreduce(const void* send_buf,
                      void* recv_buf,
                      size_t count,
                      reduction rtype,
                      datatype dtype,
                      const communicator& comm,
                      const stream& stream,
                      const allreduce_attr& attr = default_allreduce_attr,
                      const vector_class<event>& deps = {});
```

send_buf
: the buffer with `count` elements of `BufferType` that stores local data to be reduced

recv_buf [out]
: the buffer to store the reduced result, must have the same dimension as `send_buf`

count
: the number of elements of type `BufferType` in `send_buf` and `recv_buf`

rtype
: the type of the reduction operation to be applied

dtype
: the datatype of elements in `send_buf` and `recv_buf`
: must be skipped if `BufferType` can be inferred
: otherwise must be passed explicitly

comm
: the communicator that defines a group of ranks for the operation

stream
: the stream associated with the operation

attr
: optional attributes to customize the operation

deps
: an optional vector of the events that the operation should depend on

return `event`
: an object to track the progress of the operation

Alltoallv

Alltoall is a collective communication operation in which each rank sends separate blocks of data to each rank. Block sizes may differ. The j-th block of send buffer sent from the i-th rank is received by the j-th rank and is placed in the i-th block of receive buffer.

```
template <class BufferType>
event ccl::alltoallv(const BufferType* send_buf,
                     const vector_class<size_t>& send_counts,
                     BufferType* recv_buf,
                     const vector_class<size_t>& recv_counts,
                     const communicator& comm,
                     const stream& stream,
                     const alltoallv_attr& attr = default_alltoallv_attr,
                     const vector_class<event>& deps = {});

event ccl::alltoallv(const void* send_buf,
                     const vector_class<size_t>& send_counts,
                     void* recv_buf,
                     const vector_class<size_t>& recv_counts,
                     datatype dtype,
                     const communicator& comm,
                     const stream& stream,
                     const alltoallv_attr& attr = default_alltoallv_attr,
                     const vector_class<event>& deps = {});
```

send_buf
: the buffer with elements of `BufferType` that stores local blocks to be sent to each rank

send_counts

: an array with number of elements of type `BufferType` in the blocks sent for each rank
: the array's size must be equal to the number of ranks
: the values at the position of the caller's rank in `send_counts` and `recv_counts` must be equal

recv_buf [out]
: the buffer to store the received result, must be large enough to hold blocks from all ranks

recv_counts

: an array with number of elements of type `BufferType` in the blocks received from each rank
: the array's size must be equal to the number of ranks
: the values at the position of the caller's rank in `send_counts` and `recv_counts` must be equal

dtype

: the datatype of elements in `send_buf` and `recv_buf`
: must be skipped if `BufferType` can be inferred
: otherwise must be passed explicitly

comm
: the communicator that defines a group of ranks for the operation

stream
: the stream associated with the operation

attr
: optional attributes to customize the operation

deps
an optional vector of the events that the operation should depend on

return event
an object to track the progress of the operation

Barrier

Barrier synchronization is performed across all ranks of the communicator and it is completed only after all the ranks in the communicator have called it.

```
event ccl::barrier(const communicator& comm,
                   const stream& stream,
                   const barrier_attr& attr = default_barrier_attr,
                   const vector_class<event>& deps = {});
```

comm
the communicator that defines a group of ranks for the operation

stream
the stream associated with the operation

attr
optional attributes to customize the operation

deps
an optional vector of the events that the operation should depend on

return event
an object to track the progress of the operation

Broadcast

Broadcast is a collective communication operation that broadcasts data from one rank of communicator (denoted as root) to all other ranks.

```
template <class BufferType>
event ccl::broadcast(BufferType* buf,
                     size_t count,
                     int root,
                     const communicator& comm,
                     const stream& stream,
                     const broadcast_attr& attr = default_broadcast_attr,
                     const vector_class<event>& deps = {});

event ccl::broadcast(void* buf,
                     size_t count,
                     datatype dtype,
                     int root,
                     const communicator& comm,
                     const stream& stream,
                     const broadcast_attr& attr = default_broadcast_attr,
                     const vector_class<event>& deps = {});
```

buf [in,out]

the buffer with `count` elements of `BufferType`
serves as `send_buf` for root and as `recv_buf` for other ranks

count
the number of elements of type `BufferType` in `buf`

root
the rank that broadcasts `buf`

dtype

the datatype of elements in `buf`
must be skipped if `BufferType` can be inferred
otherwise must be passed explicitly

comm
the communicator that defines a group of ranks for the operation

stream
the stream associated with the operation

attr
optional attributes to customize the operation

deps
an optional vector of the events that the operation should depend on

return event
an object to track the progress of the operation

Reduce

Reduce is a collective communication operation that performs the global reduction operation on values from all ranks of the communicator and returns the result to the root rank.

Reduce is in-place when `send_buf == recv_buf`.

```
template <class BufferType>
event ccl::reduce(const BufferType* send_buf,
                  BufferType* recv_buf,
                  size_t count,
                  reduction rtype,
                  int root,
                  const communicator& comm,
                  const stream& stream,
                  const reduce_attr& attr = default_reduce_attr,
                  const vector_class<event>& deps = {});

event ccl::reduce(const void* send_buf,
                  void* recv_buf,
                  size_t count,
                  datatype dtype,
                  reduction rtype,
                  int root,
                  const communicator& comm,
                  const stream& stream,
```

(continues on next page)

(continued from previous page)

```
                    const reduce_attr& attr = default_reduce_attr,
                    const vector_class<event>& deps = {});
```

send_buf
: the buffer with `count` elements of `BufferType` that stores local data to be reduced

recv_buf [out]
: the buffer to store the reduced result, must have the same dimension as `send_buf`.
Used by the `root` rank only, ignored by other ranks.

count
: the number of elements of type `BufferType` in `send_buf` and `recv_buf`

rtype
: the type of the reduction operation to be applied

root
: the rank that gets the result of the reduction

dtype
: the datatype of elements in `send_buf` and `recv_buf`
must be skipped if `BufferType` can be inferred
otherwise must be passed explicitly

comm
: the communicator that defines a group of ranks for the operation

stream
: the stream associated with the operation

attr
: optional attributes to customize the operation

deps
: an optional vector of the events that the operation should depend on

return event
: an object to track the progress of the operation

ReduceScatter

Reduce-scatter is a collective communication operation that performs the global reduction operation on values from all ranks of the communicator and scatters the result in blocks back to all ranks.

ReduceScatter is in-place when `recv_buf == send_buf + rank * recv_count`

```
template <class BufferType>
event ccl::reduce_scatter(const BufferType* send_buf,
                          BufferType* recv_buf,
                          size_t recv_count,
                          reduction rtype,
                          const communicator& comm,
                          const stream& stream,
                          const reduce_scatter_attr& attr = default_reduce_scatter_attr,
                          const vector_class<event>& deps = {});
```

(continues on next page)

(continued from previous page)

```
event ccl::reduce_scatter(const void* send_buf,
                          void* recv_buf,
                          size_t recv_count,
                          datatype dtype,
                          reduction rtype,
                          const communicator& comm,
                          const stream& stream,
                          const reduce_scatter_attr& attr = default_reduce_scatter_attr,
                          const vector_class<event>& deps = {});
```

send_buf
: the buffer with `comm_size` * `count` elements of `BufferType` that stores local data to be reduced

recv_buf [out]
: the buffer to store the result block containing `recv_count` elements of type `BufferType`

recv_count
: the number of elements of type `BufferType` in the received block

rtype
: the type of the reduction operation to be applied

dtype
: the datatype of elements in `send_buf` and `recv_buf`
: must be skipped if `BufferType` can be inferred
: otherwise must be passed explicitly

comm
: the communicator that defines a group of ranks for the operation

stream
: the stream associated with the operation

attr
: optional attributes to customize the operation

deps
: an optional vector of the events that the operation should depend on

return `event`
: an object to track the progress of the operation

Point-To-Point Operations

OneCCL specification defines the following point-to-point operations:

- Send
- Recv

In point-to-point communication, two ranks participate in the communication so when a process sends data to a peer rank, the peer rank needs to post a `recv` call with the same datatype and count as the sending rank.

The current specification only supports blocking `send` and `recv` and does not support for multiple `send` and `receive` operations to proceed concurrently.

In the `send` operation, the peer specifies the destination process, while in the recv operation peer specifies the source process.

As with the collective operations, the communicator can perform communication operations on host or device memory buffers depending on the device used to create the communicator. Additionally, communication operations accept an execution context (stream) and may accept a vector of events on which the communication operation should depend, that is, input dependencies. The output event object provides the ability to track the operation's progress.

Note: Support for the handling of input events is optional.

BufferType is used below to define the C++ type of elements in communication operations' data buffers (`buf`, `send_buf`, and `recv_buf`). At least the following types should be supported: `[u]int{8/16/32/64}_t, float, double`. The explicit datatype parameter enable data types that cannot be inferred from the function arguments. For more information, see Custom Datatypes.

The communication operation accepts a stream object. If a communicator is created from `native_device_type`, then the stream translates to `native_stream_type` created from the corresponding device.

The communication operation may accept attribute objects. If that parameter is missed, then the default attribute object is used (`default_<operation_name>_attr`). The default attribute object is provided by the library. For more information, see Operation Attributes.

If the arguments provided to a communication operation call do not comply with the requirements of the operation, the behavior is undefined, unless otherwise specified.

Send

A blocking point-to-point communication operation that sends the data in a buf to a peer rank.

```
template <class BufferType,
event CCL_API send(BufferType *buf,
                   size_t count,
                   int peer,
                   const communicator &comm,
                   const stream &stream,
                   const pt2pt_attr &attr = default_pt2pt_attr,
                   const vector_class<event>& deps = {});

event CCL_API send(void *buf,
                   size_t count,
                   datatype dtype,
                   int peer,
                   const communicator &comm,
                   const stream &stream,
                   const pt2pt_attr &attr = default_pt2pt_attr,
                   const vector_class<event> &deps = {});
```

buf
: the buffer with count elements of `dtype` that contains the data to be sent

count
: the number of elements of type dtype in buf

dtype

the datatype of elements in buf must be skipped if `BufferType` can be inferred otherwise must be passed explicitly

peer
the destination rank

comm
the communicator that defines a group of ranks for the operation

stream
the stream associated with the operation

attr
optional attributes to customize the operation

deps
an optional vector of the events that the operation should depend on

return event
an object to track the progress of the operation

Recv

A blocking point-to-point communication operation that receives the data in a buf from a peer rank.

```
template <class BufferType,
          event CCL_API recv(BufferType *buf,
                    size_t count,
                    int peer,
                    const communicator &comm,
                    const stream &stream,
                    const pt2pt_attr &attr = default_pt2pt_attr,
                    const vector_class<event> &deps = {});

event CCL_API send(void *buf,
                    size_t count,
                    datatype dtype,
                    int peer,
                    const communicator &comm,
                    const stream &stream,
                    const pt2pt_attr &attr = default_pt2pt_attr,
                    const vector_class<event> &deps = {});
```

buf [out]
the buffer with count elements of dtype that contains the data to be sent

count
the number of elements of type dtype in buf

dtype
the datatype of elements in buf must be skipped if `BufferType` can be inferred otherwise must be passed explicitly

peer
the destination rank

comm
the communicator that defines a group of ranks for the operation

stream
The stream associated with the operation

attr
optional attributes to customize the operation

deps
an optional vector of the events that the operation should depend on

return event
object to track the progress of the operation

Note: See also:

- *Communicator*
- *Stream*
- *Event*
- *Operation Progress Tracking*

Operation Attributes

oneCCL specification defines communication operation attributes that serve as modifiers of an operation's behavior. Optionally, they may be passed to the corresponding communication operations.

oneCCL specification defines the following operation attribute classes:

- `allgatherv_attr`
- `allreduce_attr`
- `alltoallv_attr`
- `barrier_attr`
- `broadcast_attr`
- `reduce_attr`
- `reduce_scatter_attr`

oneCCL specification defines attribute identifiers that may be used to fill operation attribute objects.

The list of common attribute identifiers that may be used for any communication operation:

```
enum class operation_attr_id {
    priority    = /* unspecified */,
    to_cache    = /* unspecified */,
    synchronous = /* unspecified */,
    match_id    = /* unspecified */

    last_value  = /* unspecified, equal to the largest of all the values above */
};
```

operation_attr_id::priority
the priority of the communication operation

operation_attr_id::to_cache

persistent/non-persistent communication operation
should be used in conjunction with `match_id`

operation_attr_id::synchronous
synchronous/asynchronous communication operation

operation_attr_id::match_id

the unique identifier of the operation
in conjunction with `to_cache`, it enables the caching of the communication operation

The communication operation specific attribute identifiers may extend the list of common identifiers.

The list of attribute identifiers that may be used for *Allreduce*, *Reduce* and *ReduceScatter* collective operations:

```
enum class allreduce_attr_id {
    reduction_fn = /* unspecified */
};

enum class reduce_attr_id {
    reduction_fn = /* unspecified */
};

enum class reduce_scatter_attr_id {
    reduction_fn = /* unspecified */
};
```

allreduce_attr_id::reduction_fn / reduce_attr_id::reduction_fn / reduce_scatter_attr_id::reduction_fn
a function pointer for the custom reduction operation that follows the signature:

```
typedef void (*reduction_fn)
(
    const void*,      /* in_buf    */
    size_t,           /* in_count  */
    void*,            /* inout_buf */
    size_t*,          /* out_count */
    datatype,         /* datatype  */
    const fn_context* /* context   */
);

typedef struct {
    const char* match_id;
    const size_t offset;
} fn_context;
```

Creating an operation attribute object, which may be used in a corresponding communication operation:

```
template <class OpAttrType>
OpAttrType ccl::create_operation_attr();
```

return `OpAttrType`
an object to hold attributes for a specific communication operation

The operation attribute classes shall provide `get` and `set` methods for getting and setting of values with specific attribute identifiers.

Operation Progress Tracking

oneCCL communication operation shall return an event object to be used for tracking the operation's progress.

The `event` class shall provide the ability to wait for completion of an operation in a blocking manner, the ability to check the completion status in a non-blocking manner, and the ability to retrieve the underlying native object that is signaled when the operation completes.

Event

Waiting for the completion of an operation in a blocking manner:

```
void event::wait();
```

Checking for the completion of an operation in a non-blocking manner:

```
bool event::test();
```

return `bool`
: `true` if the operation has been completed `false` if the operation has not been completed

Retrieving a native object that is signaled when the operation completes:

```
native_event_type event::get_native();
```

return `native_event_type`

: a native object that is signaled when the operation completes
shall throw an exception if an `event` object does not wrap the native object

5.4.3 Error handling

oneCCL error handling relies on the mechanism of C++ exceptions. If an error occurs, it shall be propagated at the point of a function call where it is caught using standard C++ error handling mechanism.

Exception classification

Exception classification in oneCCL is aligned with C++ Standard Library classification. oneCCL introduces class that defines the base class in the hierarchy of oneCCL exception classes. All oneCCL routines throw exceptions inherited from this base class.

In the hierarchy of oneCCL exceptions, `ccl::exception` is the base class inherited from `std::exception` class. All other oneCCL exception classes are derived from this base class.

This specification does not require implementations to perform error-checking. However, if an implementation does provide error-checking, it shall use the following exception classes. Additional implementation-specific exception classes can be used for exceptional conditions not fitting any of these classes.

Common exceptions

Exception class	Description
`ccl::exception`	Reports general unspecified error
`ccl::invalid_argument`	Reports an error when arguments to the operation were rejected
`ccl::host_bad_alloc`	Reports an error that occurred during memory allocation on the host
`ccl::unimplemented`	Reports an error when the requested operation is not implemented
`ccl::unsupported`	Reports an error when the requested operation is not supported

5.5 Programming Model

5.5.1 Generic Workflow

Below is a generic workflow with oneCCL API

1. Create a main built-in key-value store. Its address should be distributed using an out-of-band communication mechanism and be used to create key-value stores on other processes:

```
using namespace std;

/* for example use MPI as an out-of-band communication mechanism */

int mpi_rank, mpi_size;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);

ccl::shared_ptr_class<ccl::kvs> kvs;
ccl::kvs::address_type kvs_addr;

if (mpi_rank == 0) {
    kvs = ccl::create_main_kvs();
    kvs_addr = kvs->get_address();
    MPI_Bcast((void*)kvs_addr.data(), ccl::kvs::address_max_size, MPI_BYTE, 0, MPI_COMM_
↪WORLD);
}
else {
    MPI_Bcast((void*)kvs_addr.data(), ccl::kvs::address_max_size, MPI_BYTE, 0, MPI_COMM_
↪WORLD);
    kvs = ccl::create_kvs(kvs_addr);
}
```

2. Create communicator(s):

```
/* host communications */
auto comm = ccl::create_communicator(mpi_size, mpi_rank, kvs);
```

```
/* SYCL devices communications, for example with multiple devices per process */

/* sycl_context -> sycl::context */
/* sycl_devices -> vector<sycl::device> */
/* sycl_queues  -> vector<sycl::queue> */

/* create ccl::context object from sycl::context object */
auto ccl_context = ccl::create_context(sycl_context);

/* create ccl::device objects from sycl::device objects */
vector<ccl::device> ccl_devices;
for (size_t idx = 0; idx < sycl_devices.size(); idx++) {
    ccl_devices.push_back(ccl::create_device(sycl_devices[idx]));
}

map<int, ccl::device> r2d_map;
for (auto& dev : ccl_devices) {
    int rank = /* generate a globally unique rank for a specific device */
    r2d_map[rank] = dev;
}

/* create ccl::stream objects from sycl::queue objects */
vector<ccl::stream> ccl_streams;
for (size_t idx = 0; idx < sycl_queues.size(); idx++) {
    ccl_streams.push_back(ccl::create_stream(sycl_queues[idx]));
}

auto comms = ccl::create_communicators(mpi_size * r2d_map.size(),
                                       r2d_map,
                                       ccl_context,
                                       kvs);
```

3. Execute a communication operation of choice on the communicator(s):

```
/* host communications */
allreduce(..., comm).wait();
```

```
/* SYCL devices communications */
vector<ccl::event> events;
for (auto& comm : comms) {
    events.push_back(allreduce(..., comm, ccl_streams[comm.rank()]));
}

for (auto& e : events) {
    e.wait();
}
```

CHAPTER

SIX

LEVEL ZERO

The oneAPI Level Zero (Level Zero) provides low-level direct-to-metal interfaces that are tailored to the devices in a oneAPI platform. Level Zero supports broader language features such as function pointers, virtual functions, unified memory, and I/O capabilities while also providing fine-grain explicit controls needed by higher-level runtime APIs including:

- Device discovery
- Memory allocation
- Peer-to-peer communication
- Inter-process sharing
- Kernel submission
- Asynchronous execution and scheduling
- Synchronization primitives
- Metrics reporting

The API architecture exposes both physical and logical abstractions of the underlying oneAPI platform devices and their capabilities. The device, sub-device, and memory are exposed at a physical level while command queues, events, and synchronization methods are defined as logical entities. All logical entities are bound to device-level physical capabilities. The API provides a scheduling model that is tailored to multiple uses including a low-latency submission model to the devices as well as one that is tailored to the construction and submission of work across simultaneous host threads. While heavily influenced by other low-level APIs, such as OpenCL, Level Zero is designed to evolve independently. While heavily influenced by GPU architecture, Level Zero is supportable across different oneAPI compute device architectures, such as FPGAs.

6.1 Detailed API Descriptions

The detailed specification can be found online in the specification.

CHAPTER

SEVEN

ONEDAL

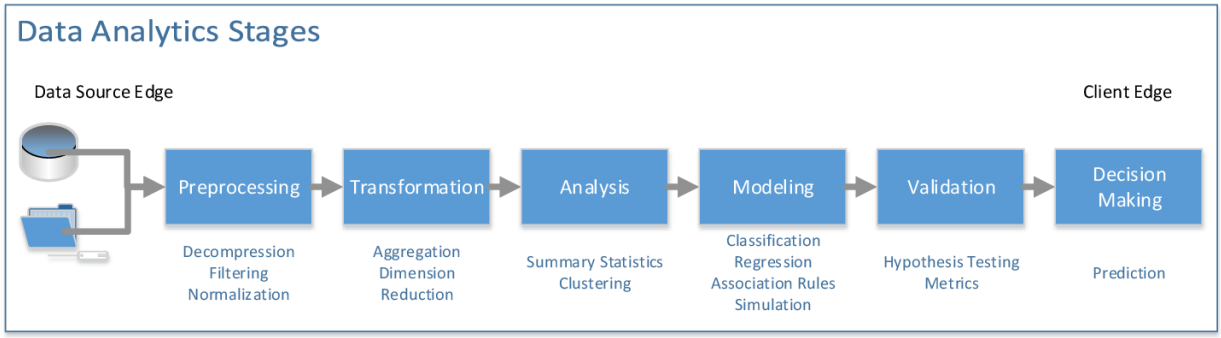
This document specifies requirements for implementations of oneAPI Data Analytics Library (oneDAL).

oneDAL is a library that helps speed up big data analysis by providing highly optimized algorithmic building blocks for all stages of data analytics (preprocessing, transformation, analysis, modeling, validation, and decision making) in batch, online, and distributed processing modes of computation. The current version of oneDAL provides Data Parallel C++ (DPC++) API extensions to the traditional C++ interface.

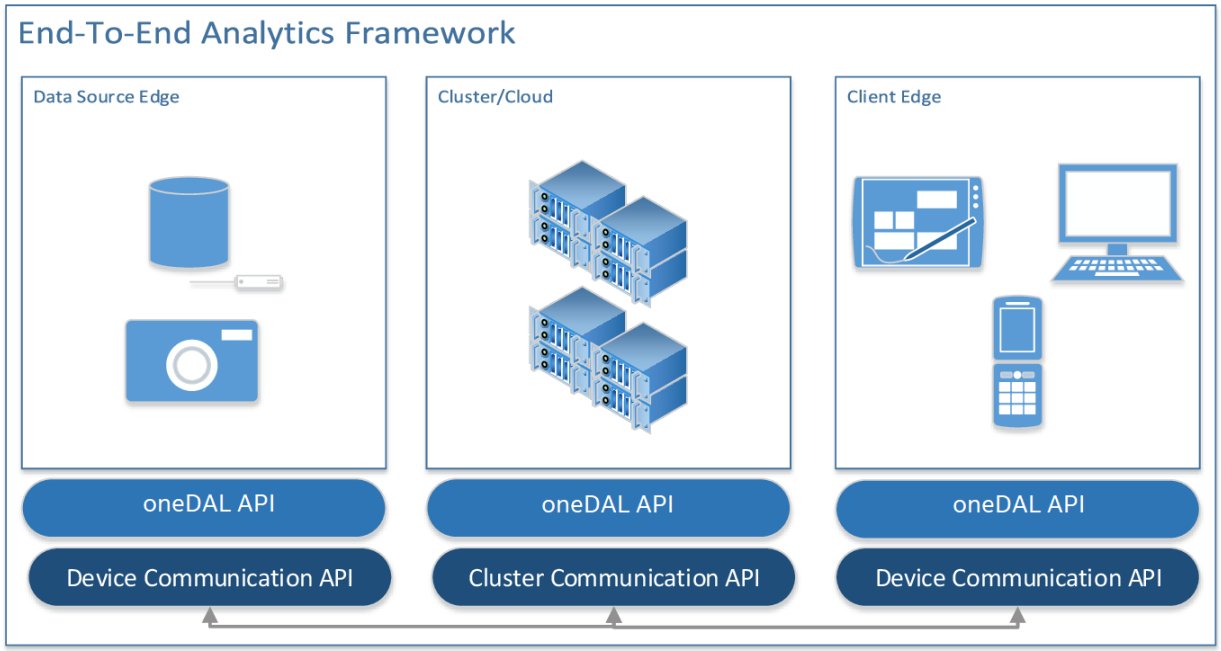
For general information, visit oneDAL GitHub* page.

7.1 Introduction

oneAPI Data Analytics Library (oneDAL) is a library that provides building blocks covering all stages of data analytics: data acquisition from a data source, preprocessing, transformation, data mining, modeling, validation, and decision making.



oneDAL supports the concept of the end-to-end analytics when some of data analytics stages are performed on the edge devices (close to where the data is generated and where it is finally consumed). Specifically, oneDAL Application Programming Interfaces (APIs) are agnostic about a particular cross-device communication technology and, therefore, can be used within different end-to-end analytics frameworks.



oneDAL consists of the following major components:

- The *Data Management* component includes classes and utilities for data acquisition, initial preprocessing and normalization, for data conversion into numeric formats (performed by one of supported Data Sources), and for model representation.
- The *Algorithms* component consists of classes that implement algorithms for data analysis (data mining) and data modeling (training and prediction). These algorithms include clustering, classification, regression, and recommendation algorithms. Algorithms support the following computation modes:
  - *Batch processing*: algorithms work with the entire data set to produce the final result
  - *Online processing*: algorithms process a data set in blocks streamed into the device's memory
  - *Distributed processing*: algorithms operate on a data set distributed across several devices (compute nodes)

    Distributed algorithms in oneDAL are abstracted from underlying cross-device communication technology, which enables use of the library in a variety of multi-device computing and data transfer scenarios.

  Depending on the usage, algorithms operate both on actual data (data set) and data models:
  - Analysis algorithms typically operate on data sets.
  - Training algorithms typically operate on a data set to train an appropriate data model.
  - Prediction algorithms typically work with the trained data model and with a working data set.
- The **Utilities** component includes auxiliary functionality intended to be used for design of classes and implementation of methods such as memory allocators or type traits.
- The **Miscellaneous** component includes functionality intended to be used by oneDAL algorithms and applications for algorithm customization and optimization on various stages of the analytical pipeline. Examples of such algorithms include solvers and random number generators.

Classes in Data Management, Algorithms, Utilities, and Miscellaneous components cover the most important usage scenarios and allow seamless implementation of complex data analytics workflows through direct API calls. At the same time, the library is an object-oriented framework that helps customize the API by redefining particular classes and methods of the library.

7.2 Glossary

7.2.1 Machine learning terms

Categorical feature
A *feature* with a discrete domain. Can be *nominal* or *ordinal*.

Synonyms: discrete feature, qualitative feature

Classification
A *supervised machine learning problem* of assigning *labels* to *feature vectors*.

Examples: predict what type of object is on the picture (a dog or a cat?), predict whether or not an email is spam

Clustering
An *unsupervised machine learning problem* of grouping *feature vectors* into bunches, which are usually encoded as *nominal* values.

Example: find big star clusters in the space images

Continuous feature
A *feature* with values in a domain of real numbers. Can be *interval* or *ratio*

Synonyms: quantitative feature, numerical feature

Examples: a person's height, the price of the house

CSV file
A comma-separated values file (csv) is a type of a text file. Each line in a CSV file is a record containing fields that are separated by the delimiter. Fields can be of a numerical or a text format. Text usually refers to categorical values. By default, the delimiter is a comma, but, generally, it can be any character. For more details, see.

Dataset
A collection of *observations*.

Dimensionality reduction
A problem of transforming a set of *feature vectors* from a high-dimensional space into a low-dimensional space while retaining meaningful properties of the original feature vectors.

Feature
A particular property or quality of a real object or an event. Has a defined type and domain. In machine learning problems, features are considered as input variable that are independent from each other.

Synonyms: attribute, variable, input variable

Feature vector
A vector that encodes information about real object, an event or a group of objects or events. Contains at least one *feature*.

Example: A rectangle can be described by two features: its width and height

Inference
A process of applying a *trained model* to the *dataset* in order to predict *response* values based on input *feature vectors*.

Synonym: prediction

Inference set
A *dataset* used at the *inference* stage. Usually without *responses*.

Interval feature
A *continuous feature* with values that can be compared, added or subtracted, but cannot be multiplied or divided.

Examples: a time frame scale, a temperature in Celsius or Fahrenheit

Label
A *response* with *categorical* or *ordinal* values. This is an output in *classification* and *clustering* problems.

Example: the spam-detection problem has a binary label indicating whether the email is spam or not

Model
An entity that stores information necessary to run *inference* on a new *dataset*. Typically a result of a *training* process.

Example: in linear regression algorithm, the model contains weight values for each input feature and a single bias value

Nominal feature
A *categorical feature* without ordering between values. Only equality operation is defined for nominal features.

Examples: a person's gender, color of a car

Observation
A *feature vector* and zero or more *responses*.

Synonyms: instance, sample

Ordinal feature
A *categorical feature* with defined operations of equality and ordering between values.

Example: student's grade

Outlier
Observation which is significantly different from the other observations.

Ratio feature
A *continuous feature* with defined operations of equality, comparison, addition, subtraction, multiplication, and division. Zero value element means the absence of any value.

Example: the height of a tower

Regression
A *supervised machine learning problem* of assigning *continuous responses* for *feature vectors*.

Example: predict temperature based on weather conditions

Response
A property of some real object or event which dependency from *feature vector* need to be defined in *supervised learning* problem. While a *feature* is an input in the machine learning problem, the response is one of the outputs can be made by the *model* on the *inference* stage.

Synonym: dependent variable

Supervised learning
Training process that uses a *dataset* with information about dependencies between *features* and *responses*. The goal is to get a *model* of dependencies between input *feature vector* and *responses*.

Training
A process of creating a *model* based on information extracted from a *training set*. Resulting *model* is selected in accordance with some quality criteria.

Training set
A *dataset* used at the *training* stage to create a *model*.

Unsupervised learning
Training process that uses a *training set* with no *responses*. The goal is to find hidden patters inside *feature vectors* and dependencies between them.

7.2.2 oneDAL terms

Accessor
A oneDAL concept for an object that provides access to the data of another object in the special *data format*. It abstracts data access from interface of an object and provides uniform access to the data stored in objects of different types.

Batch mode
The computation mode for an algorithm in oneDAL, where all the data needed for computation is available at the start and fits the memory of the device on which the computations are performed.

Builder
A oneDAL concept for an object that encapsulates the creation process of another object and enables its iterative creation.

Contiguous data
Data that are stored as one contiguous memory block. One of the characteristics of a *data format*.

Data format
Representation of the internal structure of the data.

Examples: data can be stored in array-of-structures or compressed-sparse-row format

Data layout
A characteristic of *data format* which describes the order of elements in a *contiguous data* block.

Example: row-major format, where elements are stored row by row

Data type
An attribute of data used by a compiler to store and access them. Includes size in bytes, encoding principles, and available operations (in terms of a programming language).

Examples: `int32_t`, `float`, `double`

Flat data
A block of *contiguous homogeneous* data.

Getter
A method that returns the value of the private member variable.

Example:

```
std::int64_t get_row_count() const;
```

Heterogeneous data
Data which contain values either of different *data types* or different sets of operations defined on them. One of the characteristics of a *data format*.

Example: A *dataset* with 100 *observations* of three *interval features*. The first two features are of float32 *data type*, while the third one is of float64 data type.

Homogeneous data
Data with values of single *data type* and the same set of available operations defined on them. One of the characteristics of a *data format*.

Example: A *dataset* with 100 *observations* of three *interval features*, each of type float32

Immutability
The object is immutable if it is not possible to change its state after creation.

Metadata
Information about logical and physical structure of an object. All possible combinations of metadata values

present the full set of possible objects of a given type. Metadata do not expose information that is not a part of a type definition, e.g. implementation details.

Example: *table* object can contain three *nominal features* with 100 *observations* (logical part of metadata). This object can store data as sparse csr array and provides direct access to them (physical part)

Online mode
The computation mode for an algorithm in oneDAL, where the data needed for computation becomes available in parts over time.

Reference-counted object
A copy-constructible and copy-assignable oneDAL object which stores the number of references to the unique implementation. Both copy operations defined for this object are lightweight, which means that each time a new object is created, only the number of references is increased. An implementation is automatically freed when the number of references becomes equal to zero.

Setter
A method that accepts the only parameter and assigns its value to the private member variable.

Example:

```
void set_row_count(std::int64_t row_count);
```

Table
A oneDAL concept for a *dataset* that contains only numerical data, *categorical* or *continuous*. Serves as a transfer of data between user's application and computations inside oneDAL. Hides details of *data format* and generalizes access to the data.

Workload
A problem of applying a oneDAL algorithm to a *dataset*.

7.2.3 Common oneAPI terms

API
Application Programming Interface

DPC++
Data Parallel C++ (DPC++) is a high-level language designed for data parallel programming productivity. DPC++ is based on *SYCL** from the Khronos* Group to support data parallelism and heterogeneous programming.

Host/Device
OpenCL [OpenCLSpec] refers to CPU that controls the connected GPU executing kernels.

JIT
Just in Time Compilation — compilation during execution of a program.

Kernel
Code written in OpenCL [OpenCLSpec] or *SYCL* and executed on a GPU device.

SPIR-V
Standard Portable Intermediate Representation - V is a language for intermediate representation of compute kernels.

SYCL
SYCL(TM) [SYCLSpec] — high-level programming model for OpenCL(TM) that enables code for heterogeneous processors to be written in a "single-source" style using completely standard C++.

7.3 Mathematical Notations

Notation	Definition
n or m	The number of *observations* in a *dataset*. Typically n is used, but sometimes m is required to distinguish two datasets, e.g., the *training set* and the *inference set*.
p or r	The number of features in a dataset. Typically p is used, but sometimes r is required to distinguish two datasets.
$a \times b$	The dimensionality of a matrix (dataset) has a rows (observations) and b columns (features).
$\|A\|$	Depending on the context may be interpreted as follows: • If A is a set, this denotes its cardinality, i.e., the number of elements in the set A. • If A is a real number, this denotes an absolute value of A.
$\|\|x\|\|$	The L_2-norm of a vector $x \in \mathbb{R}^d$, $$\|\|x\|\| = \sqrt{x_1^2 + x_2^2 + \cdots + x_d^2}.$$
$\mathrm{sgn}(x)$	Sign function for $x \in \mathbb{R}$, $$\mathrm{sgn}(x) = \begin{cases} -1, x < 0, \\ 0, x = 0, \\ 1, x > 0. \end{cases}$$
x_i	In the description of an algorithm, this typically denotes the i-th *feature vector* in the training set.
x'_i	In the description of an algorithm, this typically denotes the i-th feature vector in the inference set.
y_i	In the description of an algorithm, this typically denotes the i-th *response* in the training set.
y'_i	In the description of an algorithm, this typically denotes the i-th response that needs to be predicted by the inference algorithm given the feature vector x'_i from the inference set.

7.4 Programming model

oneDAL primarily targets algorithms that are extensively used in data analytics. These algorithms typically have many parameters, i.e. knobs to control its internal behavior and produced result. In machine learning, those parameters are often referred as *meta-parameters* to distinguish them from the model parameters learned during the training. Some algorithms define a dozen meta-parameters, while others depend on another algorithm as, for example, the logistic regression training procedure depends on an optimization algorithm.

Besides meta-parameters, machine learning algorithms may have different *stages*, such as *training* and *inference*. Moreover, the stages of an algorithm may be implemented in a variety of *computational methods*. For instance, a linear regression model could be trained by solving a system of linear equations [Friedman17] or by applying an iterative optimization solver directly to the empirical risk function [Zhang04].

The same machine learning techniques are often applied for solving problems of different types. In the example with linear regression, the same mathematical model used for solving *regression* problem is generalized for solving a *classification* problem, for example, logistic regression. Such techniques differ only in few problem-specific aspects, but share the same subset of meta-parameters and have a common computational flow. oneDAL does not distinguish these techniques into different algorithms. Instead, from oneDAL perspective, the same algorithm may perform different *computational tasks*.

From computational perspective, algorithm implementation may rely on different *floating-point types*, such as `float`,

`double` or `bfloat16`. Having a capability to specify what type is needed is important for the end user as their precision requirements vary depending on a workload.

To best tackle the mentioned challenges, each algorithm is decomposed into *descriptors* and *operations*.

7.4.1 End-to-end example

Below you can find a typical workflow of using oneDAL algorithm on GPU. The example is provided for *Principal Component Analysis algorithm (PCA)*.

The following steps depict how to:

- Read the data from CSV file
- Run the training and inference operations for PCA
- Access intermediate results obtained at the training stage

1. Include the following header that makes all oneDAL declarations available.

```
#include "oneapi/dal.hpp"

/* Standard library headers required by this example */
#include <cassert>
#include <iostream>
```

2. Create a SYCL* queue with the desired device selector. In this case, GPU selector is used:

```
const auto queue = sycl::queue{ sycl::gpu_selector{} };
```

3. Since all oneDAL declarations are in the `oneapi::dal` namespace, import all declarations from the `oneapi` namespace to use `dal` instead of `oneapi::dal` for brevity:

```
using namespace oneapi;
```

4. Use *CSV data source* to read the data from the CSV file into a *table*:

```
const auto data = dal::read<dal::table>(queue, dal::csv::data_source{"data.csv"});
```

5. Create a *PCA* descriptor, configure its parameters, and run the training algorithm on the data loaded from CSV.

```
const auto pca_desc = dal::pca::descriptor<float>
   .set_component_count(3)
   .set_deterministic(true);

const dal::pca::train_result train_res = dal::train(queue, pca_desc, data);
```

6. Print the learned eigenvectors:

```
const dal::table eigenvectors = train_res.get_eigenvectors();

const auto acc = dal::row_accessor<const float>{eigenvectors};
for (std::int64_t i = 0; i < eigenvectors.row_count(); i++) {

   /* Get i-th row from the table, the eigenvector stores pointer to USM */
   const dal::array<float> eigenvector = acc.pull(queue, {i, i + 1});
```

(continues on next page)

(continued from previous page)

```
   assert(eigenvector.get_count() == eigenvectors.get_column_count());

   std::cout << i << "-th eigenvector: ";
   for (std::int64_t j = 0; j < eigenvector.get_count(); j++) {
      std::cout << eigenvector[j] << " ";
   }
   std::cout << std::endl;
}
```

7. Use the trained model for inference to reduce dimensionality of the data:

```
const dal::pca::model model = train_res.get_model();

const dal::table data_transformed =
   dal::infer(queue, pca_desc, data).get_transformed_data();

assert(data_transformed.column_count() == 3);
```

7.4.2 Descriptors

A descriptor is an object that represents an algorithm including all its meta-parameters, dependencies on other algorithms, floating-point types, computational methods and tasks. A descriptor serves as:

- A dispatching mechanism for *operations*. Based on a descriptor type, an operation executes a particular algorithm implementation.
- An aggregator of meta-parameters. It provides an interface for setting up meta-parameters at either compile-time or run-time.
- An object that stores the state of the algorithm. In the general case, a descriptor is a stateful object whose state changes after an operation is applied.

Each oneDAL algorithm has its own dedicated namespace, where the corresponding descriptor is defined (for more details, see *Namespaces*). Descriptor, in its turn, defines the following:

- **Template parameters.** A descriptor is allowed to have any number of template parameters, but shall support at least three:
  - `Float` is a *floating-point type* that the algorithm uses for computations. This parameter is defined first and has the `oneapi::dal::default_float_t` default value.
  - `Method` is a tag-type that specifies the *computational method*. This parameter is defined second and has the `method::by_default` default value.
  - `Task` is a tag-type that specifies the *computational task*. This parameter is defined third and has the `task::by_default` default value.
- **Properties.** A property is a run-time parameter that can be accessed by means of the corresponding *getter* and *setter* methods.

The following code sample shows the common structure of a descriptor's definition for an abstract algorithm. To define a particular algorithm, the following strings shall be substituted:

- `%ALGORITHM%` is the name of an algorithm and its namespace. All classes and structures related to that algorithm are defined within the namespace.
- `%PROPERTY_NAME%` and `%PROPERTY_TYPE%` are the name and the type of one of the algorithm's properties.

```
namespace oneapi::dal::%ALGORITHM% {

template <typename Float  = default_float_t,
          typename Method = method::by_default,
          typename Task   = task::by_default,
          /* more template parameters */>
class descriptor {
public:
   /* Constructor */
   descriptor(const %PROPERTY_TYPE%& %PROPERTY_NAME%,
              /* more properties */)

   /* Getter & Setter for the property called `%PROPERTY_NAME%` */
   descriptor& set_%PROPERTY_NAME%(%PROPERTY_TYPE% value);
   %PROPERTY_TYPE% get_%PROPERTY_NAME%() const;

   /* more properties */
};

} // namespace oneapi::dal::%ALGORITHM%
```

Each meta-parameter of an algorithm is mapped to a property that shall satisfy the following requirements:

- Properties are defined with getter and setter methods. The underlying class member variable that stores the property's value is never exposed in the descriptor interface.
- The getter returns the value of the underlying class member variable.
- The setter accepts only one parameter of the property's type and assigns it to the underlying class member variable.
- Most of the properties are preset with default values, others are initialized by passing the required parameters to the constructor.
- The setter returns a reference to the descriptor object to allow chaining calls as shown in the example below.

```
auto desc = descriptor{}
   .set_property_name_1(value_1)
   .set_property_name_2(value_2)
   .set_property_name_3(value_3);
```

Floating-point Types

It is required for each algorithm to support at least one implementation-defined floating-point type. Other floating-point types are optional, for example `float`, `double`, `float16`, and `bfloat16`. It is up to a specific oneDAL implementation whether or not to support these types.

The floating-point type used as a default in descriptors is implementation-defined and shall be declared within the top-level namespace.

```
namespace oneapi::dal {
   using default_float_t = /* implementation defined */;
} // namespace oneapi::dal
```

Computational Methods

The supported computational methods are declared within the `%ALGORITHM%::method` namespace using tag-types. Algorithm shall support at least one method and declare the `by_default` type alias that refers to one of the methods as shown in the example below.

```
namespace oneapi::dal::%ALGORITHM% {
   namespace method {
      struct x {};
      struct y {};
      using by_default = x;
   } // namespace method
} // namespace oneapi::dal::%ALGORITHM%
```

Computational Tasks

The supported computational tasks are declared within the `%ALGORITHM%::task` namespace using tag-types. Algorithm shall support at least one task and declare the `by_default` type alias that refers to one of the tasks as shown in the example below.

If an algorithm assumes both `classification` and `regression` tasks, the default task shall be `classification`. In some cases where an algorithm does not have the well-defined training and inference stages an algorithm may define only one task.

```
namespace oneapi::dal::%ALGORITHM% {
   namespace task {
      struct classification {};
      struct regression {};
      using by_default = classification;
   } // namespace task
} // namespace oneapi::dal::%ALGORITHM%
```

7.4.3 Operations

An operation is a function that transforms *a descriptor* and other arguments represented via *an input* object to *a result* object. An operation is responsible for:

- Executing all of an algorithm's computational routines represented by the descriptor.
- Passing SYCL* queue to computational routines.
- Verifying preconditions and postconditions before and after the execution of computational routines.

General operation definition

The following code sample shows the declaration of an abstract operation. To declare a particular operation, the `%OPERATION%` shall be substituted with the name of the operation.

```
namespace oneapi::dal {

template <typename Descriptor>
using %OPERATION%_input_t = /* implementation defined */;
```

(continues on next page)

(continued from previous page)

```
template <typename Descriptor>
using %OPERATION%_result_t = /* implementation defined */;

template <typename Descriptor>
%OPERATION%_result_t<Descriptor> %OPERATION%(
   sycl::queue& queue,
   const Descriptor& desc,
   const %OPERATION%_input_t<Descriptor>& input);

} // namespace oneapi::dal
```

Each operation shall satisfy the following requirements:

- An operation shall accept three parameters in the following order:
  - The SYCL* queue object
  - The descriptor of the algorithm
  - The *input object*
- An operation shall return the *result object*.
- The `%OPERATION%_input_t` and `%OPERATION%_result_t` alias templates shall be used for inference of the input and return types.
- If a precondition is violated, an operation shall throw an exception derived from `oneapi::dal::logic_error`.
- If a postcondition is violated, an operation shall throw an exception derived from `oneapi::dal::runtime_error`.
- If the descriptor is incompatible with some operation, an error shall be reported at compile-time.
- The exact list of compatible operations and pre-/post- conditions shall be defined by *a particular algorithm specification*.

Operation shortcuts

In order to make the code on user side less verbose, oneDAL defines the following overloaded functions called *shortcuts* for each operation in addition to the general one described in section *General operation definition*.

- A shortcut for execution on *host* that performs the same operation as the general function on host, but does not require the queue to be passed explicitly.

```
template <typename Descriptor>
%OPERATION%_result_t<Descriptor> %OPERATION%(
   const Descriptor& desc,
   const %OPERATION%_input_t<Descriptor>& input);
```

- A shortcut that allows omitting explicit input creation.

```
template <typename Descriptor, typename... Args>
%OPERATION%_result_t<Descriptor> %OPERATION%(
   sycl::queue& queue,
   const Descriptor& desc,
   Args&&... args);
```

- A shortcut that allows omitting explicit queue and input creation. This is a combination of two previous shortcuts.

```
template <typename Descriptor, typename... Args>
%OPERATION%_result_t<Descriptor> %OPERATION%(
   const Descriptor& desc,
   Args&&... args);
```

Input

An input object aggregates all the data that the algorithm requires for performing a specific operation. The data is represented via *tables*, so, typically, an input is a collection of tables, but not limited to them and can aggregate objects of an arbitrary type.

In general, input class definition is similar to *descriptor*. An input defines properties that can be accessed by means of the corresponding *getter* and *setter* methods. Requirements to the input's properties are the same as *requirements for descriptor's properties*.

The following code sample shows the common structure of a inputs's definition. To define an input for particular algorithm and operation, the following strings shall be substituted:

- `%ALGORITHM%` is the name of an algorithm and its namespace.
- `%OPERATION%` is the name of operation.
- `%PROPERTY_NAME%` and `%PROPERTY_TYPE%` are the name and the type of one of the input's properties.

```
namespace oneapi::dal::%ALGORITHM% {

template <typename Task = task::by_default>
class OPERATION_input {
public:
   /* Constructor */
   %OPERATION%_input(const %PROPERTY_TYPE%& %PROPERTY_NAME%,
                     /* more properties */)

   /* Getter & Setter for the property called `%PROPERTY_NAME%` */
   descriptor& set_%PROPERTY_NAME%(%PROPERTY_TYPE% value);
   %PROPERTY_TYPE% get_%PROPERTY_NAME%() const;

   /* more properties */
};

} // namespace oneapi::dal::%ALGORITHM%
```

Note: An input is specific to algorithm and operation, so each `%ALGORITHM%-%OPERATION%` pair shall define its own set of the properties.

Result

A result object aggregates all output values computed by the algorithm. All assumptions about *an input* are applied to a result as well.

```
namespace oneapi::dal::%ALGORITHM% {

template <typename Task = task::by_default>
class OPERATION_result {
public:
   /* Constructor */
   %OPERATION%_result(const %PROPERTY_TYPE%& %PROPERTY_NAME%,
                      /* more properties */)

   /* Getter & Setter for the property called `%PROPERTY_NAME%` */
   descriptor& set_%PROPERTY_NAME%(%PROPERTY_TYPE% value);
   %PROPERTY_TYPE% get_%PROPERTY_NAME%() const;

   /* more properties */
};

} // namespace oneapi::dal::%ALGORITHM%
```

Supported operation

Refer to the *Supported operations* section for more information about particular operations.

Supported operations

This section describes all operations supported by oneDAL. For more information about general operation definition, refer to *Operations* section.

The table bellow specifies whether an algorithm's descriptor can be used together with each operation.

Algorithm	Operations		
	Train	*Infer*	*Compute*
K-Means	Yes	Yes	No
K-Means Initialization	No	No	Yes
k-NN	Yes	Yes	No
PCA	Yes	Yes	No

Train

The `train` operation performs *training* procedure of a machine learning algorithm. The result obtained after the training contains a *model* that can be passed to the `infer` operation.

```
namespace oneapi::dal {

template <typename Descriptor>
using train_input_t = /* implementation defined */;

template <typename Descriptor>
using train_result_t = /* implementation defined */;

template <typename Descriptor>
train_result_t<Descriptor> train(
   sycl::queue& queue,
   const Descriptor& desc,
   const train_input_t<Descriptor>& input);

} // namespace oneapi::dal
```

Infer

The `infer` operation performs *inference* procedure of a machine learning algorithm based on the model obtained as a result of training.

```
namespace oneapi::dal {

template <typename Descriptor>
using infer_input_t = /* implementation defined */;

template <typename Descriptor>
using infer_result_t = /* implementation defined */;

template <typename Descriptor>
infer_result_t<Descriptor> infer(
   sycl::queue& queue,
   const Descriptor& desc,
   const infer_input_t<Descriptor>& input);

} // namespace oneapi::dal
```

Compute

The `compute` operation is used if an algorithm does not have the well-defined training and inference stages.

```
namespace oneapi::dal {

template <typename Descriptor>
using compute_input_t = /* implementation defined */;

template <typename Descriptor>
using compute_result_t = /* implementation defined */;

template <typename Descriptor>
compute_result_t<Descriptor> compute(
   sycl::queue& queue,
   const Descriptor& desc,
   const compute_input_t<Descriptor>& input);

} // namespace oneapi::dal
```

7.4.4 Computational modes

Batch

In the batch processing mode, the algorithm works with the entire data set to produce the final result. A more complex scenario occurs when the entire data set is not available at the moment or the data set does not fit into the device memory.

Online

In the online processing mode, the algorithm processes a data set in blocks streamed into the device's memory. Partial results are updated incrementally and finalized when the last data block is processed.

Distributed

In the distributed processing mode, the algorithm operates on a data set distributed across several devices (compute nodes). On each node, the algorithm produces partial results that are later merged into the final result on the main node.

7.5 Common Interface

7.5.1 Current Version of this oneDAL Specification

This is the oneDAL specification which is part of the oneAPI specification version 1.0.

7.5.2 Header files

oneDAL public identifiers are represented in the following header files:

Header file	Description
`oneapi/dal.hpp`	The main header file of oneDAL library.
`oneapi/dal/%FILE%.hpp`	The common type definitions used in other oneDAL layers. For example, `data_type` or `range`.
`oneapi/dal/algo/%ALGO%.hpp`	A header file for a particular algorithm. The folder for the algorithm itself is `oneapi/dal/algo/%ALGO%/`. The string %ALGO% should be substituted with the name of the algorithm, for example, `kmeans` or `knn`.
`oneapi/dal/algo/misc/%FUNC%.hpp`	A header file for miscellaneous data types and functionality that is intended to be used by oneDAL algorithms and applications of the analytical pipeline. The string %FUNC% should be substituted with the name of the functionality, for example, `mt19937` or `cross_enthropy_loss`.
`oneapi/dal/table/%FILE%.hpp`	A header file for the types related to the *table* concept. The string %FILE% should be substituted with the name of the functionality, for example, `common` for key concepts related to table types (e.g., `table`, `table_metadata`, `data_layout` classes). For entities that have the `_table` suffix in their names, the related header file shall not contain this suffix in its name, for example, `homogen` for `homogen_table` class.
`oneapi/dal/io/%FILE%.hpp`	A header file for the types and entities of input-output functionality. The string %FILE% should be substituted with the name of the functionality, for example, `csv` for reading and writing csv files.
`oneapi/dal/util/%UTIL%.hpp``	A header file for auxiliary functionality, such as memory allocators or type traits, that is intended to be used for the design of classes and implementation of various methods. The string %UTIL% should be substituted with the name of the auxiliary functionality, for example, `usm_allocator` or `type_traits`.

7.5.3 Namespaces

oneDAL functionality is represented with a system of C++ namespaces described below:

Namespace	oneDAL content
`oneapi::dal`	The namespace of the library that contains externally visible data types, data management entities, processing and service functionality of oneDAL.
`oneapi::dal::%`	The namespace of the algorithm. All classes and structures related to that algorithm shall be defined within this particular namespace. To define a namespace for a specific algorithm, the string %ALGORITHM% should be substituted with its name, for example, `oneapi::dal::kmeans` or `oneapi::dal::knn`.
`oneapi::dal::%`	The namespace of the data source. All classes and structures related to that data source shall be defined within a particular namespace. To define a specific data source, the string %DATA_SOURCE% should be substituted with its name, for example, `oneapi::dal::csv`.
`oneapi::dal::m`	The namespace that contains miscellaneous data types and functionality intended to be used by oneDAL algorithms and applications for algorithm customization and optimization on various stages of the analytical pipeline.
`%PARENT%::deta`	The namespace that contains implementation details of the data types and functionality for the parent namespace. The namespace can be on any level in the namespace hierarchy. To define a specific namespace, the string %PARENT% should be substituted with the namespace for which the details are provided, for example, `oneapi::dal::detail` or `oneapi::dal::kmeans::detail`. The application shall not use any data types nor call any functionality located in the `detail` namespaces.

7.5.4 Error handling

oneDAL error handling relies on the mechanism of C++ exceptions. If an error occurs, it shall be propagated at the point of a function call where it is caught using standard C++ error handling mechanism.

Exception classification

Exception classification in oneDAL is aligned with C++ Standard Library classification. oneDAL shall introduce abstract classes that define the base class in the hierarchy of exception classes. Non-abstract exception classes are derived from the respective C++ Standard Library exception classes. oneDAL shall throw exceptions represented with non-abstract classes.

In the hierarchy of oneDAL exceptions, `oneapi::dal::exception` is the base abstract class that all other exception classes are derived from.

```
class oneapi::dal::exception;
```

Exception	Description	Abstract
`oneapi::dal::exception`	The base class of oneDAL exception hierarchy.	Yes

All oneDAL exceptions shall be divided into three groups:

- logic errors
- runtime errors
- errors with allocation

```
class oneapi::dal::logic_error : public oneapi::dal::exception;
class oneapi::dal::runtime_error : public oneapi::dal::exception;
class oneapi::dal::bad_alloc :  public oneapi::dal::exception, public std::bad_alloc;
```

Exception	Description	Abstract
`oneapi::dal::logic_error`	Reports violations of preconditions and invariants.	Yes
`oneapi::dal::runtime_error`	Reports violations of postconditions and other errors happened during the execution of oneDAL functionality.	Yes
`oneapi::dal::bad_alloc`	Reports failure to allocate storage.	Yes

All precondition and invariant errors represented by `oneapi::dal::logic_error` shall be divided into the following groups:

- invalid argument errors
- domain errors
- out of range errors
- errors with an unimplemented method or algorithm
- unsupported device

```
class oneapi::dal::invalid_argument :  public oneapi::dal::logic_error, public␣
↪std::invalid_argument;
class oneapi::dal::domain_error :  public oneapi::dal::logic_error, public std::domain_
↪error;
class oneapi::dal::out_of_range :  public oneapi::dal::logic_error, public std::out_of_
↪range;
class oneapi::dal::unimplemented :  public oneapi::dal::logic_error, public std::logic_
↪error;
class oneapi::dal::unsupported_device :  public oneapi::dal::logic_error, public␣
↪std::logic_error;
```

Exception	Description	Abstract
`oneapi::dal::invalid_argum`	Reports situations when the argument was not accepted.	No
`oneapi::dal::domain_error`	Reports situations when the argument is outside of the domain on which the operation is defined. Higher priority than `oneapi::dal::invalid_argument`.	No
`oneapi::dal::out_of_range`	Reports situations when the index is out of range. Higher priority than `oneapi::dal::invalid_argument`.	No
`oneapi::dal::unimplemented`	Reports errors that arise because an algorithm or a method is not implemented.	No
`oneapi::dal::unsupported_d`	Reports situations when a device is not supported.	No

Errors that occur during the execution of oneDAL functionality are represented with `oneapi::dal::runtime_error`. Two main groups of errors shall be distinguished:

- errors in the destination type range
- errors in the OS facilities interaction

All other errors are reported via `oneapi::dal::internal_error`.

```
class oneapi::dal::range_error :  public oneapi::dal::runtime_error, public std::range_
↪error;
class oneapi::dal::system_error :  public oneapi::dal::runtime_error, public std::system_
↪error;
class oneapi::dal::internal_error :  public oneapi::dal::runtime_error, public␣
↪std::runtime_error;
```

Exception	Description	Abstract
`oneapi::dal::range_error`	Reports situations where a result of a computation cannot be represented by the destination type.	No
`oneapi::dal::system_error`	Reports errors occurred during interaction with OS facilities.	No
`oneapi::dal::internal_errc`	Reports all runtime errors that could not be assigned to other inheritors.	No

All memory allocation errors are represented by `oneapi::dal::bad_alloc`. They shall be divided into two groups based on where they occur:

- Host memory allocation error
- Device memory allocation error

```
class oneapi::dal::host_bad_alloc :  public oneapi::dal::bad_alloc;
class oneapi::dal::device_bad_alloc :  public oneapi::dal::bad_alloc;
```

Exception	Description	Abstract
`oneapi::dal::host_bad_allc`	Reports failure to allocate storage on the host.	No
`oneapi::dal::device_bad_al`	Reports failure to allocate storage on the device.	No

7.5.5 Common type definitions

This section describes common types used in oneDAL.

Programming interface

All types and functions in this section shall be declared in the `oneapi::dal` namespace and be available via inclusion of the `oneapi/dal/common.hpp` header file.

Scalar types

oneDAL relies on the use of integral types defined in <cstdint>. This file shall be included in `oneapi/dal/common.hpp` and all oneDAL types shall use these data types.

The interfaces of the library shall use `std::int64_t` data type to represent dimensionality (for example, the number of rows and columns in the table).

It is recommended to use standard C++ types for applications as well.

Enum classes

Which base type to use when defining `enum` or `enum class` representing a oneDAL concept is up to the implementer unless specification requires a specific base type.

Data type

The implementation of *data type* concept. It shall enumerate all the data types supported by oneDAL to perform computations. The `data_type` class shall contain all the base *scalar types* and can also extend them. Base scalar types include the types whose names follow the pattern `std::int_XX_t` or `std::uint_XX_t`, where XX is 8, 16, 32, or 64.

```
enum class data_type {
    int8,
    int16,
    int32,
    int64,
    uint8,
    uint16,
    uint32,
    uint64,
    float32,
    float64,
    bfloat16
};
```

enum class **data_type**

data_type::int8
8-bit signed integer value type.

data_type::int16
16-bit signed integer value type.

data_type::int32
32-bit signed integer value type.

data_type::int64
64-bit signed integer value type.

data_type::uint8
8-bit unsigned integer value type.

data_type::uint16
16-bit unsigned integer value type.

data_type::uint32
32-bit unsigned integer value type.

data_type::uint64
64-bit unsigned integer value type.

data_type::float32
32-bit floating-point value type.

data_type::float64
64-bit floating-point value type.

data_type::bfloat16
bi-float value type.

Range

A range `[start_index, end_index)` in an array or any other container that supports value indexing.

```
struct range {
public:
    range(std::int64_t start, std::int64_t end);

    std::int64_t get_element_count(std::int64_t max_end_index) const noexcept;

    std::int64_t start_idx;

    std::int64_t end_idx;
};
```

struct **range**

Constructors

range(std::int64_t start, std::int64_t end)

Constructs a range of elements from the given start and end indices.

Parameters

- **start** – The first index in the range. The value shall be greater than or equal to 0.
- **end** – The relative end index in the range. Indicates the next index after the last one in the range. If positive, shall be greater than *start*. If negative, indicates the offset of the last element from the end of the range. For example, *start* = 1 and *end* = -2 specify the range of elements [1, 2, 3] in the set [0, 1, 2, 3, 4].

Public Methods

std::int64_t **get_element_count**(std::int64_t max_end_index) const noexcept

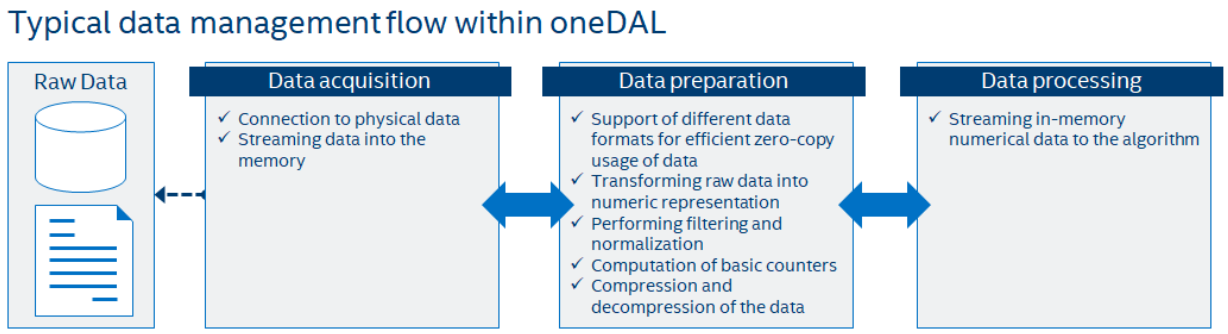
The number of elements in the range. The *max_end_index* value specifies the last maximal index in the sequence.

7.6 Data management

This section includes concepts and objects that operate on data. For oneDAL, such set of operations, or **data management**, is distributed between different stages of the *data analytics pipeline*. From a perspective of data management, this pipeline contains three main steps of data acquisition, preparation, and computation (see *the picture below*):

1. Raw data acquisition
   - Transfer out-of-memory data from various sources (databases, files, remote storage) into an in-memory representation.
2. Data preparation
   - Support different in-memory *data formats*.
   - Compress and decompress the data.
   - Convert the data into numeric representation.
   - Recover missing values.
   - Filter the data and perform data normalization.
   - Compute various statistical metrics for numerical data, such as mean, variance, and covariance.
3. Algorithm computation
   - Stream in-memory numerical data to the algorithm.

In complex usage scenarios, data flow goes through these three stages back and forth. For example, when the data are not fully available at the start of the computation, it can be done step-by-step using blocks of data. After the computation on the current block is completed, the next block should be obtained and prepared.

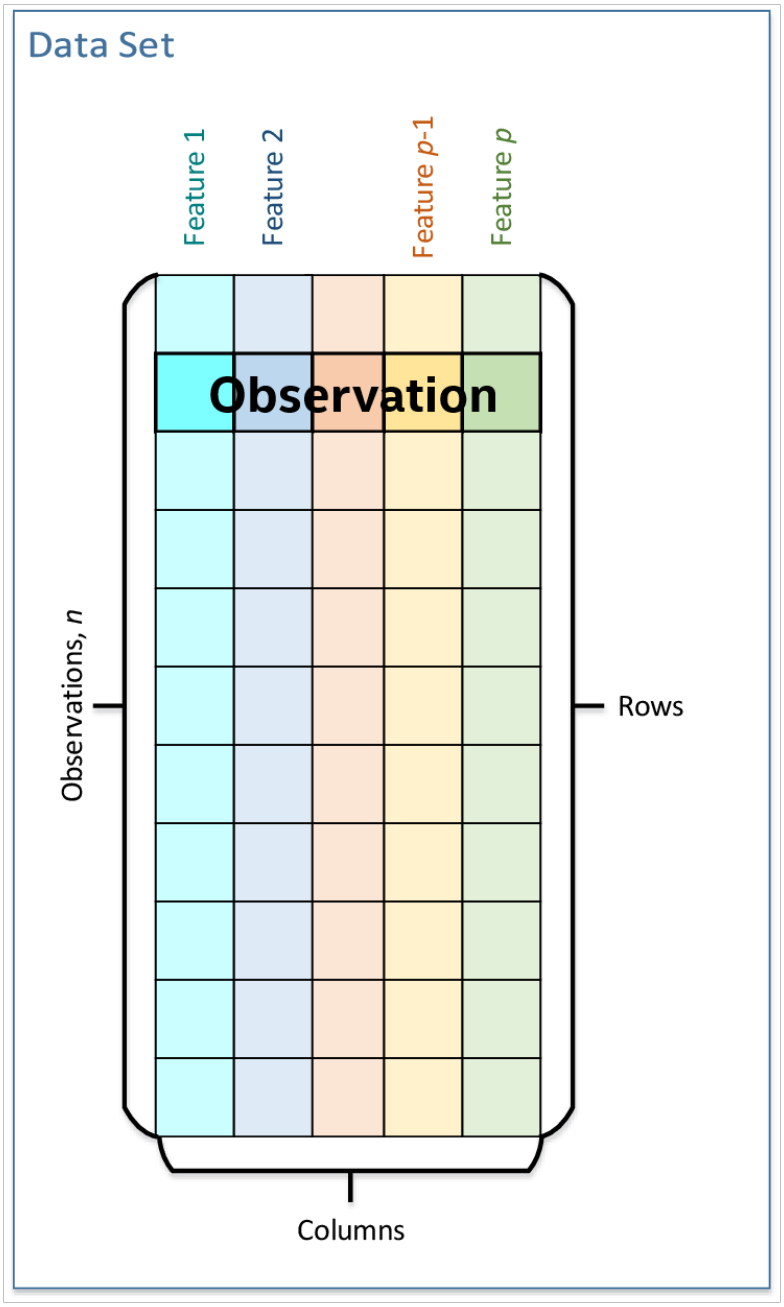


7.6.1 Key concepts

oneDAL provides a set of concepts to operate on out-of-memory and in-memory data during different stages of the *data analytics pipeline*.

Dataset

The main data-related concept that oneDAL works with is a *dataset*. It is a tabular view of data, where table rows represent the *observations* and columns represent the *features*.



The dataset is used across all stages of the data analytics pipeline. For example:

1. At the acquisition stage, it is downloaded into the local memory.
2. At the preparation stage, it is converted into a numerical representation.
3. At the computation stage, it is used as one of the *inputs* or *results* of an algorithm or a *descriptor* properties.

Data source

Data source is a concept of an out-of-memory storage for a *dataset*. It is used at the data acquisition and data preparation stages to:

- Extract datasets from external sources such as databases, files, remote storage.
- Load datasets into the device's local memory. Data do not always fit the local memory, especially when processing with accelerators. A data source provides the ability to load data by batches and extracts it directly into the device's local memory. Therefore, a data source enables complex data analytics scenarios, such as *online computations*.
- Transform datasets into their numerical representation. Data source shall automatically transform non-numeric *categorical* and *continuous* data values into one of the numeric *data formats*.

For details, see *data sources* section.

Table

Table is a concept of in-memory numerical data that are organized in a tabular view with several rows and columns. It is used at the data preparation and data processing stages to:

- Be an in-memory representation of a *dataset* or another tabular data (for example, matrices, vectors, and scalars).
- Store heterogeneous data in various *data formats*, such as dense, sparse, chunked, contiguous.
- Avoid unnecessary data copies during conversion from external data representations.
- Transfer memory ownership of the data from user application to the table, or share it between them.
- Connect with the *data source* to convert data from an out-of-memory into an in-memory representation.
- Support streaming of the data to the algorithm.
- Access the underlying data on a device in a required *data format*, e.g. by blocks with the defined *data layout*.

Note: For thread-safety reasons and better integration with external entities, a table provides a read-only access to the data within it, thus, table object shall be *immutable*.

This concept has different logical organization and physical *format of the data*:

- Logically, a table contains n rows and p columns. Every column may have its own type of data values and a set of allowed operations.
- Physically, a table can be organized in different ways: as a *homogeneous*, *contiguous* array of bytes, as a *heterogeneous* list of arrays of different *data types*, in a compressed-sparse-row format. The number of bytes needed to store the data differs from the number of elements $n \times p$ within a table.

For details, see *tables* section.

Table metadata

Table metadata concept provides an additional information about data in the table:

1. The *data types* of the columns.
2. The logical types of data in the columns: *nominal*, *ordinal*, *interval*, or *ratio*.

Only the properties of data that do not affect table concept definition shall be the part of metadata concept.

> **Warning:** While extending the table concept, specification implementer shall distinguish whether a new property they are adding is a property of a particular `table` sub-type or a property of table metadata.
>
> For example, *data layout* and *data format* are properties of table objects since they affect the structure of a table, its contract, and behavior. The list of names of features or columns inside the table is the example of metadata property.

Accessor

Accessor is a concept that defines a single way to extract the data from a *table*. It allows to:

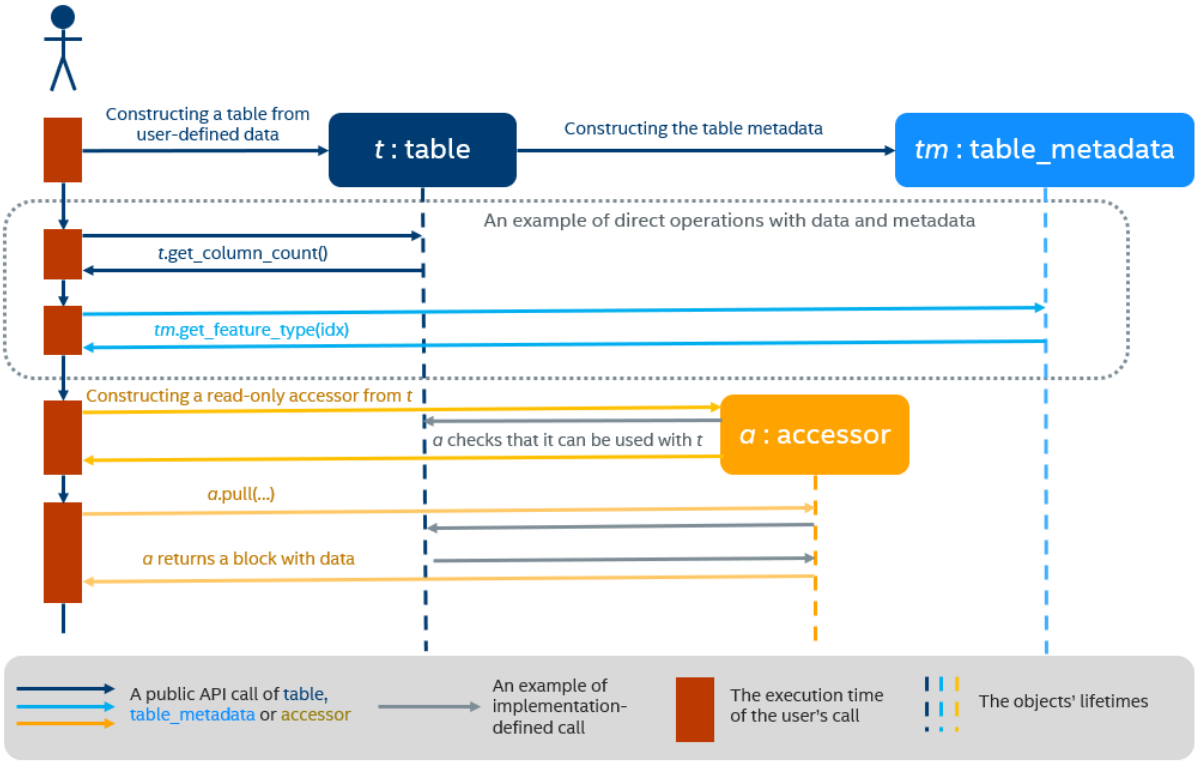
- Have unified access to the data from *table* objects of different types, without exposing their implementation details.
- Provide a *flat* view on the data blocks of a *table* for better data locality. For example, the accessor returns a column of the table stored in row-major format as a contiguous array.
- Acquire data in a desired *data format* for which a specific set of operations is defined.
- Have read-only access to the data.

For details, see *accessors* section.

Example of interaction between table and accessor objects

This section provides a basic usage scenario of the *table* and *accessor* concepts and demonstrates the relations between them. *The following diagram* shows objects of these concepts, which are highlighted by colors:

- *table* object is dark blue
- *accessor* is orange
- *table metadata* is light blue



To perform computations on a dataset, one shall create a *table* object first. It can be done either using a *data source* or directly from user-defined memory. The diagram shows the creation of a *table* object *t* from the data provided by user (not shown on the diagram). During a table creation, an object *tm* of table metadata is constructed and initialized using the data.

Once a table object is created, it can be used as an input in computations or as a parameter of some algorithm. The data in the table can be accessed via its own interface or via read-only accessor as shown on the diagram.

7.6.2 Details

This section includes the detailed descriptions of all data management objects in oneDAL.

Array

The array is a simple concept over the data in oneDAL. It represents a storage that:

1. Holds the data allocated inside it or references to the external data. The data are organized as one *homogeneous* and *contiguous* memory block.
2. Contains information about the memory block's size.
3. Represents either *immutable* or mutable data.
4. Provides an ability to change the data state from immutable to mutable one.
5. Holds ownership information on the data (see the *data ownership requirements* section).
6. Ownership information on the data can be shared between several arrays. It is possible to create a new array from another one without any data copies.

Usage example

The following listing provides a brief introduction to the array API and an example of basic usage scenario:

```
#include <CL/sycl.hpp>
#include <iostream>
#include <string>
#include "oneapi/dal/array.hpp"

using namespace oneapi;

void print_property(const std::string& description, const auto& property) {
   std::cout << description << ": " << property << std::endl;
}

int main() {
   sycl::queue queue { sycl::default_selector() };

   constexpr std::int64_t data_count = 4;
   const float data[] = { 1.0f, 2.0f, 3.0f, 4.0f };

   // Creating an array from immutable user-defined memory
   auto arr_data = dal::array<float>::wrap(data, data_count);

   // Creating an array from internally allocated memory filled by ones
   auto arr_ones = dal::array<float>::full(queue, data_count, 1.0f);

   print_property("Is arr_data mutable", arr_data.has_mutable_data()); // false
   print_property("Is arr_ones mutable", arr_ones.has_mutable_data()); // true

   // Creating new array from arr_data without data copy - they share ownership␣
↪information.
   dal::array<float> arr_mdata = arr_data;

   print_property("arr_mdata elements count", arr_mdata.get_count()); // equal to data_␣
↪count
   print_property("Is arr_mdata mutable", arr_mdata.has_mutable_data()); // false

   /// Copying data inside arr_mdata to new mutable memory block.
   /// arr_data still refers to the original data pointer.
   arr_mdata.need_mutable_data(queue);

   print_property("Is arr_data mutable", arr_data.has_mutable_data()); // false
   print_property("Is arr_mdata mutable", arr_mdata.has_mutable_data()); // true

   queue.submit([&](sycl::handler& cgh){
      auto mdata = arr_mdata.get_mutable_data();
      auto cones = arr_ones.get_data();
      cgh.parallel_for<class array_addition>(sycl::range<1>(data_count), [=](sycl::id<1>␣
↪idx) {
         mdata[idx[0]] += cones[idx[0]];
      });
   }).wait();
```

(continues on next page)

(continued from previous page)

```
   std::cout << "arr_mdata values: ";
   for(std::int64_t i = 0; i < arr_mdata.get_count(); i++) {
      std::cout << arr_mdata[i] << ", ";
   }
   std::cout << std::endl;

   return 0;
}
```

Data ownership requirements

The array shall satisfy the following requirements on managing the memory blocks:

1. An array shall retain:
   - A pointer to the immutable data block of size `count`;
   - A pointer to the mutable data block of size `count`.
2. If an array represents mutable data, both pointers shall point to the mutable data block.
3. If an array represents immutable data, pointer to the mutable data block shall be `nullptr`.
4. An array shall use shared ownership semantics to manage the lifetime of the stored data block:
   - Several array objects may own the same data block;
   - An array releases the ownership when one of the following happens:
     - The array owning the data block is destroyed;
     - The array owning the data block is assigned another memory block via `operator=` or `reset()`;
   - If the array that releases the ownership is the last remaining object owning the data block, the release of ownership is followed by the data block deallocation.
   - The data block is deallocated using the deleter object that is provided to array during construction. If no deleter object provided, an array calls the default deallocating function that corresponds to the internal memory allocation mechanism.
5. If a managed pointer to the data block is replaced by another pointer via `reset()`, the array that managed the pointer releases the ownership of it and starts managing the lifetime of the data block represented by the other pointer.
6. If an array changes its state from immutable to mutable via `need_mutable_data()`, it releases the ownership of immutable data block and start managing lifetime of the mutable data block.
7. An array object may own no data. An array like this is called **zero-sized**:
   - Pointers to the immutable and mutable data of the zero-sized array shall be `nullptr`;
   - The data block size `count` shall be `0`.

Implementation notes

A typical array implementation may be organized in the following way:

1. An array class has the following member variables:
    - A pointer to the immutable data block;
    - A pointer to the mutable data block;
    - A pointer to the ownership structure that implements the shared ownership semantics;
    - The data block size `count`;
2. An ownership structure is an object that stores:
    - A pointer to either immutable or mutable data block;
    - The deleter object;
    - The reference count (the number of array objects that own the associated data block);
3. If an array starts managing the lifetime of the data block represented by the pointer `p` and deleter `d`, it creates the ownership structure object and initialize it with `p` and `d`. The reference count of the ownership structure is assigned one.
4. If an array object releases the ownership, the reference count of the ownership structure is decremented.
    - If that count reaches zero, the ownership structure deallocates the memory block and the array destroys the ownership structure.
    - If that count is greater than zero, the ownership structure is not destroyed.
5. If a copy of the array object is created, the reference count of the ownership structure is incremented and a pointer to the same ownership structure is assigned to the created copy. The other member variables of an array class are copied as is.

Note: You may choose an arbitrary implementation strategy that satisfies array requirements.

Programming interface

All types and functions in this section shall be declared in the `oneapi::dal` namespace and be available via inclusion of the `oneapi/dal/array.hpp` header file.

All the `array` class methods can be divided into several groups:

1. Constructors that are used to create an array from external, mutable or immutable memory.
2. Constructors and assignment operators that are used to create an array that shares its data with another one.
3. The group of `reset()` methods that are used to re-assign an array to another external memory block.
4. The group of `reset()` methods that are used to re-assign an array to an internally allocated memory block.
5. The methods that are used to access the data.
6. Static methods that provide simplified ways to create an array either from external memory or by allocating it within a new object.

```
template <typename Data>
class array {
public:
    using data_t = Data;

    static array<Data> empty(const sycl::queue& queue,
                             std::int64_t count,
                             const sycl::usm::alloc& alloc = sycl::usm::alloc::shared);

    template <typename Element>
    static array<Data> full(sycl::queue& queue,
                            std::int64_t count,
                            Element&& element,
                            const sycl::usm::alloc& alloc = sycl::usm::alloc::shared);

    static array<Data> zeros(sycl::queue& queue,
                             std::int64_t count,
                             const sycl::usm::alloc& alloc = sycl::usm::alloc::shared);

    template <typename ExtData>
    static array<Data> wrap(ExtData* data,
                            std::int64_t count,
                            const std::vector<sycl::event>& dependencies = {});

    array();

    array(const array<Data>& other);

    array(array<Data>&& other);

    template <typename ExtData, typename Deleter>
    explicit array(const sycl::queue& queue,
                   ExtData* data,
                   std::int64_t count,
                   Deleter&& deleter,
                   const std::vector<sycl::event>& dependencies = {});

    template <typename RefData, typename ExtData>
    explicit array(const array<RefData>& ref, ExtData* data, std::int64_t count);

    array<Data> operator=(const array<Data>& other);

    array<Data> operator=(array<Data>&& other);

    const Data* get_data() const noexcept;

    bool has_mutable_data() const noexcept;

    Data* get_mutable_data() const;

    array& need_mutable_data(sycl::queue& queue,
                             const sycl::usm::alloc& alloc = sycl::usm::alloc::shared);
```

(continues on next page)

(continued from previous page)

```
    std::int64_t get_count() const noexcept;

    std::int64_t get_size() const noexcept;

    const Data& operator[](std::int64_t index) const noexcept;

    void reset();

    void reset(const sycl::queue& queue,
               std::int64_t count,
               const sycl::usm::alloc& alloc = sycl::usm::alloc::shared);

    template <typename ExtData, typename Deleter>
    void reset(ExtData* data,
               std::int64_t count,
               Deleter&& deleter,
               const std::vector<sycl::event>& dependencies = {});

    template <typename RefData, typename ExtData>
    void reset(const array<RefData>& ref, ExtData* data, std::int64_t count);
};
```

template<typename **Data**>
class **array**

Template Parameters

Data – The type of the memory block elements within the array. *Data* can represent any *data type*.

Public Static Methods

static array<Data> **empty**(const sycl::queue &queue, std::int64_t count, const sycl::usm::alloc &alloc = sycl::usm::alloc::shared)

Creates a new array instance by allocating a mutable memory block. The created array manages the lifetime of the allocated memory block. The function is not required to initialize the values of the allocated memory block.

Parameters

- **queue** – The SYCL* queue object.
- **count** – The number of elements of type *Data* to allocate memory for.
- **alloc** – The kind of USM to be allocated.

Preconditions

`count > 0`

Postconditions

```
get_count() == count
has_mutable_data() == true
```

template<typename **Element**>

static array<Data> **full**(sycl::queue &queue, std::int64_t count, *Element* &&element, const sycl::usm::alloc &alloc = sycl::usm::alloc::shared)

Creates a new array instance by allocating a mutable memory block and filling its content with a scalar value. The created array manages the lifetime of the allocated memory block.

Template Parameters

Element – The type from which array elements of type *Data* can be constructed.

Parameters

- **queue** – The SYCL* queue object.
- **count** – The number of elements of type *Data* to allocate memory for.
- **element** – The value that is used to fill a memory block.
- **alloc** – The kind of USM to be allocated.

Preconditions

```
count > 0
```

Postconditions

```
get_count() == count
has_mutable_data() == true
get_data()[i] == element, 0 <= i < count
```

static array<Data> **zeros**(sycl::queue &queue, std::int64_t count, const sycl::usm::alloc &alloc = sycl::usm::alloc::shared)

Creates a new array instance by allocating a mutable memory block and filling its content with zeros. The created array manages the lifetime of the allocated memory block.

Parameters

- **queue** – The SYCL* queue object.
- **count** – The number of elements of type *Data* to allocate memory for.
- **alloc** – The kind of USM to be allocated.

Preconditions

```
count > 0
```

Postconditions

```
get_count() == count
has_mutable_data() == true
get_data()[i] == 0, 0 <= i < count
```

template<typename **ExtData**>
static array<Data> **wrap**(*ExtData* *data, std::int64_t count, const std::vector<sycl::event> &dependencies = {})

Creates a new array instance from a pointer to externally-allocated memory block. The created array does not manage the lifetime of the user-provided memory block. It is the responsibility of the programmer to make sure that *data* pointer remains valid as long as this array object exists.

Template Parameters

ExtData – Either *Data* or *const Data* type.

Parameters

- **data** – The pointer to the mutable or immutable externally-allocated memory block.
- **count** – The number of elements of type *Data* in the memory block.
- **dependencies** – Events indicating the availability of the *data* for reading or writing.

Preconditions

```
data != nullptr
count > 0
```

Postconditions

```
get_count() == count
get_data() == data
has_mutable_data() == false
```

Constructors

array()

Creates a zero-sized array without memory allocation.

Postconditions

```
get_count() == 0
get_data() == nullptr
has_mutable_data() == false
```

array(const *array*<Data> &other)

Creates a new array instance that shares an ownership with the *other* array.

array(*array*<Data> &&other)

Creates a new array instance that transfers the ownership from the *other* array. After the construction of a new instance, the behaviour of the *other* is defined by the implementation.

Postconditions

```
other.get_count() == 0
other.get_data() == nullptr
has_mutable_data() == false
```

template<typename **ExtData**, typename **Deleter**>
array(const sycl::queue &queue, *ExtData* *data, std::int64_t count, *Deleter* &&deleter, const std::vector<sycl::event> &dependencies = {})

Creates a new array instance from a pointer to externally-allocated memory block. The created array manages the lifetime of the user-provided memory block. The memory block is deallocated using a custom deleter object provided by the user.

Template Parameters

- **ExtData** – Either *Data* or *const Data* type.
- **Deleter** – The type of a deleter used to deallocate the *data*. The expression `deleter(data)` must be well-formed (can be compiled) and not throw any exceptions.

Parameters

- **queue** – The SYCL* queue object.
- **data** – The pointer to the mutable or immutable externally-allocated mutable data.

- **count** – The number of elements of type *Data* in the memory block.
- **deleter** – The object used to deallocate *data*.
- **dependencies** – Events that indicate when *data* becomes ready to be read or written.

Preconditions

```
data != nullptr
count > 0
```

Postconditions

```
get_count() == count
get_data() == data
has_mutable_data() == true
get_mutable_data() == data
```

template<typename **RefData**, typename **ExtData**>
array(const *array*<*RefData*> &ref, *ExtData* *data, std::int64_t count)

Creates a new array instance that shares the ownership with the reference array while storing the pointer to another memory block provided by the user. The lifetime of the user-provided memory block is not managed by the created array. One of the use cases of this constructor is the creation of an array with an offset, for example, `array{ other, other.get_data() + offset }`. The array created this way shares the ownership with the *other*, but points to its data with an offset. It is the responsibility of the programmer to make sure that *data* pointer remains valid as long as this array object exists.

Template Parameters

- **RefData** – The type of elements in the reference array.
- **ExtData** – Either *Data* or *const Data* type.

Parameters

- **ref** – The reference array which shares the ownership with the created one.
- **data** – The unmanaged pointer to the mutable or immutable externally-allocated memory block.
- **count** – The number of elements of type *Data* in the *data*.

Preconditions

```
data != nullptr
count > 0
```

Postconditions

```
get_count() == count
get_data() == data
```

Public Methods

array<Data> **operator=**(const array<Data> &other)

Replaces the immutable and mutable data pointers and the number of elements by the values stored in the *other* array.

Postconditions

```
get_data() == other.get_data()
```

```
get_count() == other.get_count()
get_mutable_data() == other.get_mutable_data()
```

array<Data> **operator=**(array<Data> &&other)

Replaces the immutable and mutable data pointers and the number of elements by the values stored in the *other* array.

Postconditions

```
get_data() == other.get_data()
get_count() == other.get_count()
get_mutable_data() == other.get_mutable_data()
```

const Data ***get_data**() const noexcept

The pointer to the immutable memory block.

bool **has_mutable_data**() const noexcept

Returns whether an array contains mutable data or not.

Data ***get_mutable_data**() const

The pointer to the mutable memory block.

Preconditions

`has_mutable_data() == true`, othewise throws `domain_error`

array &**need_mutable_data**(sycl::queue &queue, const sycl::usm::alloc &alloc = sycl::usm::alloc::shared)

Does nothing if an array contains mutable data. Otherwise, allocates a mutable memory block and copies the content of the immutable memory block into it. The array manages the lifetime of the allocated mutable memory block. Returns the reference to the same array instance.

Parameters

- **queue** – The SYCL* queue object.
- **alloc** – The kind of USM to be allocated.

Postconditions

```
has_mutable_data() == true
```

std::int64_t **get_count**() const noexcept

The number of elements of type *Data* in a memory block.

std::int64_t **get_size**() const noexcept

The size of memory block in bytes.

const Data &**operator[]**(std::int64_t index) const noexcept

Provides a read-only access to the elements of an array. No bounds checking is performed.

void **reset**()

Releases the ownership of the managed memory block.

Preconditions

```
count > 0
```

Postconditions

```
get_count() == count
has_mutable_data() == true
```

void **reset**(const sycl::queue &queue, std::int64_t count, const sycl::usm::alloc &alloc = sycl::usm::alloc::shared)

Releases the ownership of the managed memory block and replaces it by a newly allocated mutable memory block. The lifetime of the allocated memory block is managed by the array.

Parameters

- **queue** – The SYCL* queue object.
- **count** – The number of elements of type *Data* to allocate memory for.
- **alloc** – The kind of USM to be allocated.

Preconditions

```
count > 0
```

Postconditions

```
get_count() == count
```

template<typename **ExtData**, typename **Deleter**>
void **reset**(*ExtData* *data, std::int64_t count, *Deleter* &&deleter, const std::vector<sycl::event> &dependencies = {})

Releases the ownership of the managed memory block and replace it by a pointer to externally-allocated memory block. The lifetime of the memory block is managed by the array. The memory block is deallocated using a custom deleter object provided by the user.

Template Parameters

- **ExtData** – Either *Data* or *const Data* type.
- **Deleter** – The type of a deleter used to deallocate the *data*. The expression `deleter(data)` must be well-formed (can be compiled) and not throw any exceptions.

Parameters

- **data** – The pointer to the to the mutable or immutable externally-allocated memory block.
- **count** – The number of elements of type *Data* in the *data*.
- **deleter** – The object used to deallocate *data*.
- **dependencies** – Events indicating the availability of the *data* for reading or writing.

Preconditions

```
data != nullptr
count > 0
```

Postconditions

```
get_count() == count
get_data() == data
has_mutable_data() == true
get_mutable_data() == data
```

template<typename **RefData**, typename **ExtData**>

void **reset**(const array<*RefData*> &ref, *ExtData* *data, std::int64_t count)

Releases the ownership of the managed memory block and starts managing the lifetime of the reference array while storing the pointer to another memory block provided by the user. The lifetime of the user-provided memory block is not managed. It is the responsibility of the programmer to make sure that *data* pointer remains valid as long as this array object exists.

Template Parameters

- **RefData** – The type of elements in the reference array.
- **ExtData** – Either *Data* or *const Data* type.

Parameters

- **ref** – The reference array which shares the ownership with the created one.
- **data** – The unmanaged pointer to the mutable or immutable externally-allocated memory block.
- **count** – The number of elements of type *Data* in the *data*.

Preconditions

```
data != nullptr
count > 0
```

Postconditions

```
get_count() == count
get_data() == data
```

Accessors

This section defines *requirements* to an *accessor* implementation and introduces several *accessor types*.

Requirements

Each accessor implementation shall:

1. Define a single *format of the data* for the access. Every accessor type shall return and use only one data format.
2. Provide read-only access to the data in the *table* types.
3. Provide the `pull()` method for obtaining the values from the table.
4. Be lightweight. Its constructors shall not have computationally intensive operations such data copy, reading, or conversion. These operations shall be performed by method `pull()`. Support of copy- and move- constructors by the accessor is not required since it shall be designed for use in a local scope - directly in a place when it is created.
5. The `pull()` method shall avoid data copy and conversion when it is possible to return the pointer to the memory block in the table. This is applicable for cases such as when the *data format* and *data types* of the data within the table are the same as the *data format* and *data type* for the access.

Accessor Types

oneDAL defines a set of accessor classes. Each class supports one specific way of obtaining data from the *table*.

All accessor classes in oneDAL are listed below:

Accessor type	Description	List of supported types
row accessor	Provides access to the range of rows as one *contiguous homogeneous* block of memory.	*homogen table*
column accessor	Provides access to the range of values within a single column as one *contiguous homogeneous* block of memory.	*homogen table*

Details

Column accessor

The `column_accessor` class provides a read-only access to the column values of the *table* as *contiguous homogeneous* array.

Usage example

```
#include <CL/sycl.hpp>
#include <iostream>

#include "oneapi/dal/table/homogen.hpp"
#include "oneapi/dal/table/column_accessor.hpp"

using namespace oneapi;

int main() {
   sycl::queue queue { sycl::default_selector() };

   constexpr float host_data[] = {
      1.0f, 1.5f, 2.0f,
      2.1f, 3.2f, 3.7f,
      4.0f, 4.9f, 5.0f,
      5.2f, 6.1f, 6.2f
   };

   constexpr std::int64_t row_count = 4;
   constexpr std::int64_t column_count = 3;

   auto shared_data = sycl::malloc_shared<float>(row_count * column_count, queue);
   auto event = queue.memcpy(shared_data, host_data, sizeof(float) * row_count * column_
↪count);
   auto t = dal::homogen_table::wrap(queue, data, row_count, column_count, { event });

   // Accessing whole elements in a first column
   dal::column_accessor<const float> acc { t };
```

(continues on next page)

(continued from previous page)

```
   auto block = acc.pull(queue, 0);
   for(std::int64_t i = 0; i < block.get_count(); i++) {
      std::cout << block[i] << ", ";
   }
   std::cout << std::endl;

   sycl::free(shared_data, queue);
   return 0;
}
```

Programming interface

All types and functions in this section shall be declared in the `oneapi::dal` namespace and be available via inclusion of the `oneapi/dal/table/column_accessor.hpp` header file.

```
template <typename Data>
class column_accessor {
public:
    using data_t = std::remove_const_t<Data>;

public:
    column_accessor(const table& obj);

    array<data_t> pull(sycl::queue& queue,
                       std::int64_t column_index,
                       const range& rows              = { 0, -1 },
                       const sycl::usm::alloc& alloc = sycl::usm::alloc::shared) const;

    Data* pull(sycl::queue& queue,
               array<data_t>& block,
               std::int64_t column_index,
               const range& rows             = { 0, -1 },
               const sycl::usm::alloc& alloc = sycl::usm::alloc::shared) const;
};
```

template<typename **Data**>
class **column_accessor**

Template Parameters

Data – The type of data values in blocks returned by the accessor. Shall be const-qualified for read-only access. An accessor shall support at least `float`, `double`, and `std::int32_t` types of *Data*.

Constructors

column_accessor(const *table* &obj)

Creates a new read-only accessor object from the table. The check that the accessor supports the table kind of *obj* shall be performed. The reference to the *obj* table shall be stored within the accessor to obtain data from the table.

Public Methods

array<data_t> **pull**(sycl::queue &queue, std::int64_t column_index, const *range* &rows = {0, -1}, const sycl::usm::alloc &alloc = sycl::usm::alloc::shared) const

Provides access to the column values of the table. The method shall return an array that directly points to the memory within the table if it is possible. In that case, the array shall refer to the memory as to immutable data. Otherwise, the new memory block shall be allocated, the data from the table rows shall be converted and copied into this block. The array shall refer to the block as to mutable data.

Parameters

- **queue** – The SYCL* queue object.
- **column_index** – The index of the column from which the data shall be returned by the accessor.
- **rows** – The range of rows that should be read in the $column_index$ block.
- **alloc** – The requested kind of USM in the returned block.

Preconditions

rows are within the range of *[0, obj.row_count)*.
column_index is within the range of *[0, obj.column_count)*.

Data ***pull**(sycl::queue &queue, *array*<data_t> &block, std::int64_t column_index, const *range* &rows = {0, -1}, const sycl::usm::alloc &alloc = sycl::usm::alloc::shared) const

Provides access to the column values of the table. The method shall return the *block*.data pointer.

Parameters

- **queue** – The SYCL* queue object.
- **block** – The block which memory is reused (if it is possible) to obtain the data from the table. The block memory shall be reset either when its size is not big enough, or when it contains immutable data, or when direct memory from the table can be used. If the block is reset to use a direct memory pointer from the object, it shall refer to this pointer as to immutable memory block.
- **column_index** – The index of the column from which the data shall be returned by the accessor.
- **rows** – The range of rows that should be read in the $column_index$ block.
- **alloc** – The requested kind of USM in the returned block.

Preconditions

rows are within the range of *[0, obj.row_count)*.
column_index is within the range of *[0, obj.column_count)*.

Row accessor

The `row_accessor` class provides a read-only access to the rows of the *table* as *contiguous homogeneous* array.

Usage example

```
#include <CL/sycl.hpp>
#include <iostream>

#include "oneapi/dal/table/homogen.hpp"
#include "oneapi/dal/table/row_accessor.hpp"

using namespace oneapi;

int main() {
   sycl::queue queue { sycl::default_selector() };

   constexpr float host_data[] = {
      1.0f, 1.5f, 2.0f,
      2.1f, 3.2f, 3.7f,
      4.0f, 4.9f, 5.0f,
      5.2f, 6.1f, 6.2f
   };

   constexpr std::int64_t row_count = 4;
   constexpr std::int64_t column_count = 3;

   auto shared_data = sycl::malloc_shared<float>(row_count * column_count, queue);
   auto event = queue.memcpy(shared_data, host_data, sizeof(float) * row_count * column_
↪count);
   auto t = dal::homogen_table::wrap(queue, data, row_count, column_count, { event });

   // Accessing second and third rows of the table
   dal::row_accessor<const float> acc { t };

   auto block = acc.pull(queue, {1, 3});
   for(std::int64_t i = 0; i < block.get_count(); i++) {
      std::cout << block[i] << ", ";
   }
   std::cout << std::endl;

   sycl::free(shared_data, queue);
   return 0;
}
```

Programming interface

All types and functions in this section shall be declared in the `oneapi::dal` namespace and be available via inclusion of the `oneapi/dal/table/row_accessor.hpp` header file.

```
template <typename Data>
class row_accessor {
public:
    using data_t = std::remove_const_t<Data>;

public:
    row_accessor(const table& obj);

    array<data_t> pull(sycl::queue& queue,
                       const range& rows             = { 0, -1 },
                       const sycl::usm::alloc& alloc = sycl::usm::alloc::shared) const;

    Data* pull(sycl::queue& queue,
               array<data_t>& block,
               const range& rows             = { 0, -1 },
               const sycl::usm::alloc& alloc = sycl::usm::alloc::shared) const;
};
```

template<typename **Data**>
class **row_accessor**

Template Parameters

Data – The type of data values in blocks returned by the accessor. Shall be const-qualified for read-only access. An accessor shall support at least `float`, `double`, and `std::int32_t` types of $Data$.

Constructors

row_accessor(const *table* &obj)

Creates a new read-only accessor object from the table. The check that the accessor supports the table kind of obj shall be performed. The reference to the obj table shall be stored within the accessor to obtain data from the table.

Public Methods

array<data_t> **pull**(sycl::queue &queue, const *range* &rows = {0, -1}, const sycl::usm::alloc &alloc = sycl::usm::alloc::shared) const

Provides access to the rows of the table. The method shall return an array that directly points to the memory within the table if it is possible. In that case, the array shall refer to the memory as to immutable data. Otherwise, the new memory block shall be allocated, the data from the table rows shall be converted and copied into this block. The array shall refer to the block as to mutable data.

Parameters

- **queue** – The SYCL* queue object.
- **rows** – The range of rows that data shall be returned from the accessor.
- **alloc** – The requested kind of USM in the returned block.

Preconditions

rows are within the range of *[0, obj.row_count)*.

Data *`pull`(sycl::queue &queue, *array*<data_t> &block, const *range* &rows = {0, -1}, const sycl::usm::alloc &alloc = sycl::usm::alloc::shared) const

Provides access to the rows of the table. The method shall return the *`block`*`.data` pointer.

Parameters

- **`queue`** – The SYCL* queue object.
- **`block`** – The block which memory is reused (if it is possible) to obtain the data from the table. The block memory shall be reset either when its size is not big enough, or when it contains immutable data, or when direct memory from the table can be used. If the block is reset to use a direct memory pointer from the object, it shall refer to this pointer as to immutable memory block.
- **`rows`** – The range of rows that data shall be returned from the accessor.
- **`alloc`** – The requested kind of USM in the returned block.

Preconditions

rows are within the range of *[0, obj.row_count)*.

Data Sources

This section describes the types related to the *data source* concept.

Read

Read operation is a function that transforms a data source and other arguments represented via *an args* object to *a result* object. The operation is responsible for:

- Executing all of the data retrieval and transformation routines of the data source.
- Passing a SYCL* queue to the data retrieval and transformation routines.

Read operation definition

The following code sample shows the declaration for a read operation.

```
namespace oneapi::dal {

template <typename Object, typename DataSource>
using read_args_t = /* implementation defined */;

template <typename Object, typename DataSource>
using read_result_t = Object;

template <typename Object, typename DataSource>
read_result_t<Object, DataSource> read(
   sycl::queue& queue,
   const DataSource& data_source,
   const read_args_t<Object, DataSource>& args);

} // namespace oneapi::dal
```

Each operation shall satisfy the following requirements:

- An operation shall accept three parameters in the following order:
  - The SYCL* queue object.
  - The data source.
  - The *args object*.
- An operation shall return the *result object*.
- The `read_args_t` and `read_result_t` alias templates shall be used for inference of the args and return types.

Read operation shortcuts

In order to make the code on user side less verbose, oneDAL defines the following overloaded functions called *shortcuts* for a read operation in addition to the general one described in section *Read operation definition*.

- A shortcut for execution on *host*. Performs the same operation as the general function on host, but does not require passing the queue explicitly.

```
template <typename Object, typename DataSource>
read_result_t<Object, DataSource> read(
   const DataSource& data_source,
   const read_args_t<Object, DataSource>& args);
```

- A shortcut that allows omitting explicit args creation.

```
template <typename Object, typename DataSource, typename... Args>
read_result_t<Object, DataSource> read(
   sycl::queue& queue,
   const DataSource& data_source,
   Args&&... args);
```

- A shortcut that allows omitting explicit queue and args creation. This is a combination of two previous shortcuts.

```
template <typename Object, typename DataSource, typename... Args>
read_result_t<Object, DataSource> read(
   const DataSource& data_source,
   Args&&... args);
```

Args

- The string `%DATA_SOURCE%` should be substituted with the name of the data source, for example, `csv`.
- `%PROPERTY_NAME%` and `%PROPERTY_TYPE%` should be substituted with the name and the type of one of the data source args properties.

```
namespace oneapi::dal::%DATA_SOURCE% {

template <typename Object, typename DataSource>
class read_args {
public:
   read_args(
```

(continues on next page)

(continued from previous page)

```
    const %PROPERTY_TYPE_1%& property_name_1,
    const %PROPERTY_TYPE_2%& property_name_2,
    /* more properties */
  )
  /* Getter & Setter for the property called `%PROPERTY_NAME_1%` */
  descriptor& set_%PROPERTY_NAME_1%(%PROPERTY_TYPE_1% value);
  %PROPERTY_TYPE_1% get_%PROPERTY_NAME_1%() const;
  /* Getter & Setter for the property called `%PROPERTY_NAME_2%` */
  descriptor& set_%PROPERTY_NAME_2%(%PROPERTY_TYPE_2% value);
  %PROPERTY_TYPE_2% get_%PROPERTY_NAME_2%() const;
  /* more properties */
};
} // namespace oneapi::dal::%DATA_SOURCE%
```

Result

The result of a `read` operation is an instance of an in-memory object with `Object` type.

Data Source Types

oneDAL defines a set of classes.

Data source type	Description
CSV data source	Data source that allows reading data from a text file into a *table*.

Details

CSV data source

Class `csv::data_source` is an API for accessing the data source represented as a *csv file*. CSV data source shall be used with `read` operation to extract data in text format from the given input file, process it using provided parameters (such as delimiter and read options), transform it into numerical representation, and store it as an in-memory *dataset* of a chosen type.

Supported type of in-memory object for `read` operation with CSV data source is `oneapi::dal::table`.

CSV data source requires input file name to be set in the constructor, while the other parameters of the constructor such as delimiter and read options rely on default values.

Usage example

```
using namespace oneapi;

const auto data_source = dal::csv::data_source("data.csv", ',');

const auto table = dal::read<dal::table>(data_source);
```

Programming Interface

All types and functions in this section shall be declared in the `oneapi::dal::csv` namespace and be available via inclusion of the `oneapi/dal/io/csv.hpp` header file.

```
enum class read_options : std::uint64_t {
   none = 0,
   parse_header = 1 << 0
};

constexpr char default_delimiter = ',';
constexpr read_options default_read_options = read_options::none;

class data_source {
public:
   data_source(const char *file_name,
               char delimiter = default_delimiter,
               read_options opts = default_read_options);

   data_source(const std::string &file_name,
               char delimiter = default_delimiter,
               read_options opts = default_read_options);

   std::string get_file_name() const;
   char get_delimiter() const;
   read_options get_read_options() const;
};
```

class **data_source**

data_source(const char *file_name, char delimiter = default_delimiter, read_options opts = default_read_options)

Creates a new instance of a CSV data source with the given *file_name*, *delimiter* and read options *opts* flag.

data_source(const std::string &file_name, char delimiter = default_delimiter, read_options opts = default_read_options)

Creates a new instance of a CSV data source with the given *file_name*, *delimiter* and read options *opts* flag.

std::string **file_name** = ""

A string that contains the name of the file with the dataset to read.

Getter

```
std::string get_filename() const
```

char **delimiter** = default_delimiter

A character that represents the delimiter between separate features in the input file.

Getter

```
char get_delimter() const
```

read_options **options** = default_read_options

Value that stores read options to be applied during reading of the input file. Enabled `parse_header` option indicates that the first line in the input file shall be processed as a header record with features names.

Getter

```
read_options get_read_options() const
```

Reading `oneapi::dal::read<Object>(...)`

Args

```
template <typename Object>
class read_args {
public:
   read_args();
};
```

template<typename **Object**>
class **read_args**

read_args()

Creates args for the read operation with the default attribute values.

Operation

`oneapi::dal::table` is the only supported value of the `Object` template parameter for *read* operation with CSV data source.

template<typename **Object**, typename **DataSource**>
Object **read**(const *DataSource* &ds)

Template Parameters

- **Object** – oneDAL object type that shall be produced as a result of reading from the data source.
- **DataSource** – CSV data source `csv::data_source`.

Tables

This section describes the types related to the *table* concept.

Type	Description
table	A common implementation of the table concept. Base class for other table types.
table_metadata	An implementation of *table metadata* concept.
data_layout	An enumeration of *data layouts* used to store contiguous data blocks inside the table.
feature_type	An enumeration of *feature* types used in oneDAL to define set of available operations onto the data.

Requirements on table types

Each implementation of *table* concept shall:

1. Follow the definition of the *table* concept and its restrictions (e.g., *immutability*).
2. Be derived from the `oneapi::dal::table` class. The behavior of this class can be extended, but cannot be weaken.
3. Be *reference-counted*.
4. Every new `oneapi::dal::table` sub-type shall define a unique id number - the "kind" that represents objects of that type in runtime.

The following listing provides an example of table API to illustrate table kinds and copy-assignment operation:

```
using namespace onedal;

// Creating homogen_table sub-type.
dal::homogen_table table1 = homogen_table::wrap(queue, data_ptr, row_count, column_
↪count);

// table1 and table2 share the same data (no data copy is performed)
dal::table table2 = table1;

// Creating an empty table
dal::table table3;

std::cout << table1.get_kind()     == table2.get_kind() << std::endl; // true
std::cout << homogen_table::kind() == table2.get_kind() << std::endl; // true
std::cout << table2.get_kind()     == table3.get_kind() << std::endl; // false

// Referring table3 to the table2.
table3 = table2;
std::cout << table2.get_kind() == table3.get_kind() << std::endl; // true
```

Table types

oneDAL defines a set of classes that implement the *table* concept for a specific data format:

Table type	Description
homogen table	A dense table that contains *contiguous homogeneous* data.

Programming interface

All types and functions in this section shall be declared in the `oneapi::dal` namespace and be available via inclusion of the `oneapi/dal/table/common.hpp` header file.

Table

A base implementation of the *table* concept. The `table` type and all of its subtypes shall be *reference-counted*:

1. The instance shall store a pointer to table implementation that holds all property values and data
2. The reference count indicating how many table objects refer to the same implementation.
3. The table shall increment the reference count for it to be equal to the number of table objects sharing the same implementation.
4. The table shall decrement the reference count when the table goes out of the scope. If the reference count is zero, the table shall free its implementation.

```
class table {
public:
    table();

    table(const table& other);

    table(table&& other);

    table& operator=(const table& other);

    table& operator=(table&& other);

    bool has_data() const noexcept;

    std::int64_t get_column_count() const;

    std::int64_t get_row_count() const;

    const table_metadata& get_metadata() const;

    std::int64_t get_kind() const;

    data_layout get_data_layout() const;
};
```

class **table**

Constructors

table()

An empty table constructor: creates the table instance with zero number of rows and columns. Implementation shall be set to the special "empty" object that returns all the property values set to default (see Properties section).

table(const *table* &other)

Creates a new table instance which shares implementation with $other$.

table(*table* &&other)

Creates a new table instance and moves implementation from $other$ into it.

Public Methods

table &**operator=**(const *table* &other)

Replaces the implementation by another one from $other$.

table &**operator=**(*table* &&other)

Swaps the implementation of this object and $other$.

bool **has_data**() const noexcept

Indicates whether a table contains non-zero number of rows and columns.

std::int64_t **get_column_count**() const

The number of columns in the table.

std::int64_t **get_row_count**() const

The number of rows in the table.

const *table_metadata* &**get_metadata**() const

The metadata object that holds additional information about the data within the table.

std::int64_t **get_kind**() const

The runtime id of the table type. Each table sub-type shall have its unique *kind*. An empty table (see the default constructor) shall have a unique *kind* value as well.

data_layout **get_data_layout**() const

The layout of the data within the table.

Table metadata

An implementation of the *table metadata* concept. Holds additional information about data within the table. The objects of `table_metadata` shall be *reference-counted*.

```
class table_metadata {
public:
    table_metadata();

    table_metadata(const array<data_type>& dtypes, const array<feature_type>& ftypes);

    std::int64_t get_feature_count() const;

    const feature_type& get_feature_type(std::int64_t feature_index) const;
```

(continues on next page)

(continued from previous page)

```
    const data_type& get_data_type(std::int64_t feature_index) const;
};
```

class **table_metadata**

Constructors

table_metadata()

Creates the metadata instance without information about the features. The `feature_count` shall be set to zero. The *data_type* and *feature_type* properties shall not be initialized.

table_metadata(const *array*<*data_type*> &dtypes, const *array*<*feature_type*> &ftypes)

Creates the metadata instance from external information about the data types and the feature types.

Parameters

- **dtypes** – The data types of the features. Shall be assigned into the *data_type* property.
- **ftypes** – The feature types. Shall be assigned into the *feature_type* property.

Preconditions

dtypes.get_count() == *ftypes*.get_count()

Public Methods

std::int64_t **get_feature_count**() const

The number of features that metadata contains information about.

Preconditions

feature_count >= 0

const *feature_type* &**get_feature_type**(std::int64_t feature_index) const

Feature types in the metadata object. Shall be within the range *[0, feature_count)*.

const *data_type* &**get_data_type**(std::int64_t feature_index) const

Data types of the features in the metadata object. Shall be within the range *[0, feature_count)*.

Data layout

An implementation of the *data layout* concept.

```
enum class data_layout { unknown, row_major, column_major };
```

enum class **data_layout**

data_layout::unknown
Represents the *data layout* that is undefined or unknown at this moment.

data_layout::row_major
The data block elements are stored in raw-major layout.

data_layout::column_major
The data block elements are stored in column_major layout.

Feature type

An implementation of the logical data types.

```
enum class feature_type { nominal, ordinal, interval, ratio };
```

enum class **feature_type**

feature_type::nominal
Represents the type of *Nominal feature*.

feature_type::ordinal
Represents the type of *Ordinal feature*.

feature_type::interval
Represents the type of *Interval feature*.

feature_type::ratio
Represents the type of *Ratio feature*.

Homogeneous table

Class `homogen_table` is an implementation of a table type for which the following is true:

- The data within the table are dense and stored as one contiguous memory block.
- All the columns have the same *data type*.

Programming interface

All types and functions in this section shall be declared in the `oneapi::dal` namespace and be available via inclusion of the `oneapi/dal/table/homogen.hpp` header file.

```
class homogen_table : public table {
public:
    static std::int64_t kind();

    template <typename Data>
    static homogen_table wrap(const sycl::queue& queue,
                              const Data* data_pointer,
                              std::int64_t row_count,
                              std::int64_t column_count,
                              const sycl::vector_class<sycl::event>& dependencies = {},
                              data_layout layout = data_layout::row_major);

public:
    homogen_table();

    template <typename Data, typename ConstDeleter>
    homogen_table(const sycl::queue& queue,
                  const Data* data_pointer,
                  std::int64_t row_count,
                  std::int64_t column_count,
                  ConstDeleter&& data_deleter,
```

(continues on next page)

(continued from previous page)

```
                const sycl::vector_class<sycl::event>& dependencies = {},
                data_layout layout                                  = data_layout::row_
↪major);

    template <typename Data>
    const Data* get_data() const {
        return reinterpret_cast<const Data*>(this->get_data());
    }

    const void* get_data() const;

    std::int64_t get_kind() const {
        return kind();
    }
};
```

class **homogen_table**

Public Static Methods

static std::int64_t **kind**()

Returns the unique id of *homogen_table* class.

template<typename **Data**>
static *homogen_table* **wrap**(const sycl::queue &queue, const *Data* *data_pointer, std::int64_t row_count, std::int64_t column_count, const sycl::vector_class<sycl::event> &dependencies = {}, *data_layout* layout = *data_layout*::row_major)

Creates a new *homogen_table* instance from externally-defined data block. Table object refers to the data but does not own it. The responsibility to free the data remains on the user side. The `data` shall point to the *data_pointer* memory block.

Template Parameters

Data – The type of elements in the data block that will be stored into the table. The table shall initialize data types of metadata with this data type. The feature types shall be set to default values for *Data* type: contiguous for floating-point, ordinal for integer types. The *Data* type shall be at least `float`, `double` or `std::int32_t`.

Parameters

- **queue** – The SYCL* queue object.
- **data_pointer** – The pointer to a homogeneous data block.
- **row_count** – The number of rows in the table.
- **column_count** – The number of columns in the table.
- **dependencies** – Events indicating availability of the *data* for reading or writing.
- **layout** – The layout of the data. Shall be *data_layout::row_major* or *data_layout::column_major*.

Constructors

homogen_table()

Creates a new *homogen_table* instance with zero number of rows and columns. The *kind* shall be set to`homogen_table::kind()`. All the properties shall be set to default value (see the Properties section).

template<typename **Data**, typename **ConstDeleter**>

homogen_table(const sycl::queue &queue, const *Data* *data_pointer, std::int64_t row_count, std::int64_t column_count, *ConstDeleter* &&data_deleter, const sycl::vector_class<sycl::event> &dependencies = {}, *data_layout* layout = *data_layout*::row_major)

Creates a new *homogen_table* instance from externally-defined data block. Table object owns the data pointer. The `data` shall point to the *data_pointer* memory block.

Template Parameters

- **Data** – The type of elements in the data block that will be stored into the table. The $Data$ type shall be at least `float`, `double` or `std::int32_t`.
- **ConstDeleter** – The type of a deleter called on *data_pointer* when the last table that refers it is out of the scope.

Parameters

- **queue** – The SYCL* queue object.
- **data_pointer** – The pointer to a homogeneous data block.
- **row_count** – The number of rows in the table.
- **column_count** – The number of columns in the table.
- **data_deleter** – The deleter that is called on the *data_pointer* when the last table that refers it is out of the scope.
- **dependencies** – Events indicating availability of the $data$ for reading or writing.
- **layout** – The layout of the data. Shall be *data_layout::row_major* or *data_layout::column_major*.

Public Methods

template<typename **Data**>
const *Data* ***get_data**() const

Returns the `data` pointer cast to the $Data$ type. No checks are performed that this type is the actual type of the data within the table.

const void ***get_data**() const

The pointer to the data block within the table. Shall be equal to *nullptr* when `row_count == 0` and `column_count == 0`.

std::int64_t **get_kind**() const

The unique id of the homogen table type.

7.7 Algorithms

The Algorithms component consists of classes that implement algorithms for data analysis (data mining) and data modeling (training and prediction). These algorithms include matrix decompositions, clustering, classification, and regression algorithms, as well as association rules discovery.

7.7.1 Clustering

K-Means

The K-Means algorithm solves *clustering* problem by partitioning n feature vectors into k clusters minimizing some criterion. Each cluster is characterized by a representative point, called *a centroid*.

Operation	Computational methods	Programming Interface		
Training	*Lloyd's*	*train(...)*	*train_input*	*train_result*
Inference	*Lloyd's*	*infer(...)*	*infer_input*	*infer_result*

Mathematical formulation

Training

Given the training set $X = \{x_1, \ldots, x_n\}$ of p-dimensional feature vectors and a positive integer k, the problem is to find a set $C = \{c_1, \ldots, c_k\}$ of p-dimensional centroids that minimize the objective function

$$\Phi_X(C) = \sum_{i=1}^{n} d^2(x_i, C),$$

where $d^2(x_i, C)$ is the squared Euclidean distance from x_i to the closest centroid in C,

$$d^2(x_i, C) = \min_{1 \le j \le k} \|x_i - c_j\|^2, \quad 1 \le i \le n.$$

Expression $\|\cdot\|$ denotes L_2 norm.

Note: In the general case, d may be an arbitrary distance function. Current version of the oneDAL spec defines only Euclidean distance case.

Training method: *Lloyd's*

The Lloyd's method [Lloyd82] consists in iterative updates of centroids by applying the alternating *Assignment* and *Update* steps, where t denotes a index of the current iteration, e.g., $C^{(t)} = \{c_1^{(t)}, \ldots, c_k^{(t)}\}$ is the set of centroids at the t-th iteration. The method requires the initial centroids $C^{(1)}$ to be specified at the beginning of the algorithm ($t = 1$).

(1) Assignment step: Assign each feature vector x_i to the nearest centroid. $y_i^{(t)}$ denotes the assigned label (cluster index) to the feature vector x_i.

$$y_i^{(t)} = \arg \min_{1 \le j \le k} \|x_i - c_j^{(t)}\|^2, \quad 1 \le i \le n.$$

Each feature vector from the training set X is assigned to exactly one centroid so that X is partitioned to k disjoint sets (clusters)

$$S_j^{(t)} = \left\{ x_i \in X : y_i^{(t)} = j \right\}, \quad 1 \le j \le k.$$

(2) Update step: Recalculate centroids by averaging feature vectors assigned to each cluster.

$$c_j^{(t+1)} = \frac{1}{|S_j^{(t)}|} \sum_{x \in S_j^{(t)}} x, \quad 1 \le j \le k.$$

The steps (1) and (2) are performed until the following **stop condition**,

$$\sum_{j=1}^{k} \left\| c_j^{(t)} - c_j^{(t+1)} \right\|^2 < \varepsilon,$$

is satisfied or number of iterations exceeds the maximal value T defined by the user.

Inference

Given the inference set $X' = \{x'_1, \ldots, x'_m\}$ of p-dimensional feature vectors and the set $C = \{c_1, \ldots, c_k\}$ of centroids produced at the training stage, the problem is to predict the index $y'_j \in \{0, \ldots, k-1\}$, $1 \le j \le m$, of the centroid in accordance with a method-defined rule.

Inference method: *Lloyd's*

Lloyd's inference method computes the y'_j as an index of the centroid closest to the feature vector x'_j,

$$y'_j = \arg\min_{1 \le l \le k} \|x'_j - c_l\|^2, \quad 1 \le j \le m.$$

Usage example

Training

```
kmeans::model<> run_training(const table& data,
                             const table& initial_centroids) {
   const auto kmeans_desc = kmeans::descriptor<float>{}
      .set_cluster_count(10)
      .set_max_iteration_count(50)
      .set_accuracy_threshold(1e-4);

   const auto result = train(kmeans_desc, data, initial_centroids);

   print_table("labels", result.get_labels());
   print_table("centroids", result.get_model().get_centroids());
   print_value("objective", result.get_objective_function_value());

   return result.get_model();
}
```

Inference

```
table run_inference(const kmeans::model<>& model,
                    const table& new_data) {
   const auto kmeans_desc = kmeans::descriptor<float>{}
      .set_cluster_count(model.get_cluster_count());

   const auto result = infer(kmeans_desc, model, new_data);
```

(continues on next page)

(continued from previous page)

```
   print_table("labels", result.get_labels());
}
```

Programming Interface

All types and functions in this section shall be declared in the `oneapi::dal::kmeans` namespace and be available via inclusion of the `oneapi/dal/algo/kmeans.hpp` header file.

Descriptor

```
template <typename Float = float,
          typename Method = method::by_default,
          typename Task = task::by_default>
class descriptor {
public:
   explicit descriptor(std::int64_t cluster_count = 2);

   int64_t get_cluster_count() const;
   descriptor& set_cluster_count(int64_t);

   int64_t get_max_iteration_count() const;
   descriptor& set_max_iteration_count(int64_t);

   double get_accuracy_threshold() const;
   descriptor& set_accuracy_threshold(double);
};
```

template<typename **Float** = float, typename **Method** = method::*by_default*, typename **Task** = task::*by_default*>
class **descriptor**

Template Parameters

- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
- **Method** – Tag-type that specifies an implementation of algorithm. Can be `method::`*`lloyd`*.
- **Task** – Tag-type that specifies the type of the problem to solve. Can be `task::`*`clustering`*.

Constructors

descriptor(std::int64_t cluster_count = 2)

Creates a new instance of the class with the given `cluster_count`.

Properties

int64_t **max_iteration_count**

The maximum number of iterations T. **Default value**: 100.

Getter & Setter

```
int64_t get_max_iteration_count() const
descriptor & set_max_iteration_count(int64_t)
```

Invariants

max_iteration_count >= 0

int64_t **cluster_count**

The number of clusters k. **Default value**: 2.

Getter & Setter

```
int64_t get_cluster_count() const
descriptor & set_cluster_count(int64_t)
```

Invariants

cluster_count > 0

double **accuracy_threshold**

The threshold ε for the stop condition. **Default value**: 0.0.

Getter & Setter

```
double get_accuracy_threshold() const
descriptor & set_accuracy_threshold(double)
```

Invariants

accuracy_threshold >= 0.0

Method tags

```
namespace method {
   struct lloyd {};
   using by_default = lloyd;
} // namespace method
```

struct **lloyd**

Tag-type that denotes *Lloyd's* computational method.

using **by_default** = *lloyd*

Alias tag-type for *Lloyd's* computational method.

Task tags

```
namespace task {
   struct clustering {};
   using by_default = clustering;
} // namespace task
```

struct **clustering**

Tag-type that parameterizes entities used for solving *clustering problem*.

using **by_default** = *clustering*

Alias tag-type for the clustering task.

Model

```
template <typename Task = task::by_default>
class model {
public:
   model();

   const table& get_centroids() const;

   int64_t get_cluster_count() const;
};
```

template<typename **Task** = task::*by_default*>
class **model**

Template Parameters
Task – Tag-type that specifies type of the problem to solve. Can be `task::`*clustering*.

Constructors

model()
Creates a new instance of the class with the default property values.

Public Methods

const *table* &**get_centroids**() const
A $k \times p$ table with the cluster centroids. Each row of the table stores one centroid.

int64_t **get_cluster_count**() const
Number of clusters k in the trained model.

Training `train(...)`

Input

```
template <typename Task = task::by_default>
class train_input {
public:
   train_input(const table& data = table{},
               const table& initial_centroids = table{});

   const table& get_data() const;
   train_input& set_data(const table&);

   const table& get_initial_centroids() const;
   train_input& set_initial_centroids(const table&);
};
```

template<typename **Task** = task::*by_default*>
class **train_input**

Template Parameters
Task – Tag-type that specifies type of the problem to solve. Can be `task::`*clustering*.

Constructors

train_input(const *table* &data = *table*{}, const *table* &initial_centroids = *table*{})

Creates a new instance of the class with the given `data` and `initial_centroids`.

Properties

const *table* &**data**

An $n \times p$ table with the data to be clustered, where each row stores one feature vector.

Getter & Setter

```
const table & get_data() const
train_input & set_data(const table &)
```

const *table* &**initial_centroids**

A $k \times p$ table with the initial centroids, where each row stores one centroid.

Getter & Setter

```
const table & get_initial_centroids() const
train_input & set_initial_centroids(const table &)
```

Result

```
template <typename Task = task::by_default>
class train_result {
public:
   train_result();

   const model<Task>& get_model() const;

   const table& get_labels() const;

   int64_t get_iteration_count() const;

   double get_objective_function_value() const;
};
```

template<typename **Task** = task::*by_default*>
class **train_result**

Template Parameters

Task – Tag-type that specifies type of the problem to solve. Can be `task::`*clustering*.

Constructors

train_result()

Creates a new instance of the class with the default property values.

Public Methods

const *model*<Task> &**get_model**() const

The trained K-means model.

const *table* &**get_labels**() const

An $n \times 1$ table with the labels y_i assigned to the samples x_i in the input data, $1 \leq 1 \leq n$.

int64_t **get_iteration_count**() const

The number of iterations performed by the algorithm.

double **get_objective_function_value**() const

The value of the objective function $\Phi_X(C)$, where C is *model*.centroids (see kmeans::*model*::centroids).

Operation

```
template <typename Float, typename Method, typename Task>
train_result<Task> train(const descriptor<Float, Method, Task>& desc,
                         const train_input<Task>& input);
```

template<typename **Float**, typename **Method**, typename **Task**>
train_result<*Task*> **train**(const *descriptor*<*Float*, *Method*, *Task*> &desc, const *train_input*<*Task*> &input)

Runs the training operation for K-Means clustering. For more details see oneapi::dal::train.

Template Parameters

- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
- **Method** – Tag-type that specifies an implementation of algorithm. Can be method::*lloyd*.
- **Task** – Tag-type that specifies type of the problem to solve. Can be task::*clustering*.

Parameters

- **desc** – Descriptor of the algorithm.
- **input** – Input data for the training operation.

Preconditions

input.data.has_data == true
input.initial_centroids.row_count == *desc*.cluster_count
input.initial_centroids.column_count == *input*.data.column_count

Postconditions

result.labels.row_count == *input*.data.row_count
result.labels.column_count == 1
result.labels[i] >= 0
result.labels[i] < *desc*.cluster_count
result.iteration_count <= *desc*.max_iteration_count
result.model.centroids.row_count == *desc*.cluster_count
result.model.centroids.column_count == *input*.data.column_count

Inference `infer(...)`

Input

```
template <typename Task = task::by_default>
class infer_input {
public:
   infer_input(const model<Task>& m = model<Task>{},
               const table& data = table{});

   const model<Task>& get_model() const;
   infer_input& set_model(const model<Task>&);

   const table& get_data() const;
   infer_input& set_data(const table&);
};
```

template<typename **Task** = task::*by_default*>
class **infer_input**

Template Parameters

Task – Tag-type that specifies type of the problem to solve. Can be `task::clustering`.

Constructors

infer_input(const *model*<Task> &m = *model*<Task>{}, const *table* &data = *table*{})

Creates a new instance of the class with the given `model` and `data`.

Properties

const *table* &**data**

The trained K-Means model. **Default value**: table{}.

Getter & Setter

```
const table & get_data() const
infer_input & set_data(const table &)
```

const *model*<Task> &**model**

An $n \times p$ table with the data to be assigned to the clusters, where each row stores one feature vector. **Default value**: model<Task>{}.

Getter & Setter

```
const model< Task > & get_model() const
infer_input & set_model(const model< Task > &)
```

Result

```
template <typename Task = task::by_default>
class infer_result {
public:
   infer_result();

   const table& get_labels() const;

   double get_objective_function_value() const;
};
```

template<typename **Task** = task::*by_default*>
class **infer_result**

Template Parameters

Task – Tag-type that specifies type of the problem to solve. Can be `task::clustering`.

Constructors

infer_result()

Creates a new instance of the class with the default property values.

Public Methods

const *table* &**get_labels**() const

An $n \times 1$ table with assignments labels to feature vectors in the input data.

double **get_objective_function_value**() const

The value of the objective function $\Phi_X(C)$, where C is defined by the corresponding *infer_input*::model::centroids.

Operation

```
template <typename Float, typename Method, typename Task>
infer_result<Task> infer(const descriptor<Float, Method, Task>& desc,
                         const infer_input<Task>& input);
```

template<typename **Float**, typename **Method**, typename **Task**>
infer_result<*Task*> **infer**(const *descriptor*<*Float*, *Method*, *Task*> &desc, const *infer_input*<*Task*> &input)

Runs the inference operation for K-Means clustering. For more details see `oneapi::dal::infer`.

Template Parameters

- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
- **Method** – Tag-type that specifies an implementation of algorithm. Can be `method::lloyd`.
- **Task** – Tag-type that specifies type of the problem to solve. Can be `task::clustering`.

Parameters

- **desc** – Descriptor of the algorithm.
- **input** – Input data for the inference operation.

Preconditions

input.data.has_data == true
input.model.centroids.has_data == true
input.model.centroids.row_count == *desc*.cluster_count
input.model.centroids.column_count == *input*.data.column_count

Postconditions

result.labels.row_count == *input*.data.row_count
result.labels.column_count == 1
result.labels[i] >= 0
result.labels[i] < *desc*.cluster_count

K-Means initialization

The K-Means initialization algorithm receives n feature vectors as input and chooses k initial centroids. After initialization, K-Means algorithm uses the initialization result to partition input data into k clusters.

Operation	Computational methods	Programming Interface		
Computing	*Dense*	*compute(...)*	*compute_input*	*compute_result*

Mathematical formulation

Computing

Given the training set $X = \{x_1, \ldots, x_n\}$ of p-dimensional feature vectors and a positive integer k, the problem is to find a set $C = \{c_1, \ldots, c_k\}$ of p-dimensional initial centroids.

Computing method: *dense*

The method chooses first k feature vectors from the training set X.

Usage example

Computing

```
table run_compute(const table& data) {
   const auto kmeans_desc = kmeans_init::descriptor<float,
                                                    kmeans_init::method::dense>{}
      .set_cluster_count(10)

   const auto result = compute(kmeans_desc, data);

   print_table("centroids", result.get_centroids());

   return result.get_centroids();
}
```

Programming Interface

All types and functions in this section shall be declared in the `oneapi::dal::kmeans_init` namespace and be available via inclusion of the `oneapi/dal/algo/kmeans_init.hpp` header file.

Descriptor

```
template <typename Float = float,
          typename Method = method::by_default,
          typename Task = task::by_default>
class descriptor {
public:

   explicit descriptor(std::int64_t cluster_count = 2);

   std::int64_t get_cluster_count() const;
   descriptor& set_cluster_count(std::int64_t);

};
```

template<typename **Float** = float, typename **Method** = method::*by_default*, typename **Task** = task::*by_default*>
class **descriptor**

Template Parameters

- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
- **Method** – Tag-type that specifies an implementation of K-Means Initialization algorithm.
- **Task** – Tag-type that specifies the type of the problem to solve. Can be `task::`*`init`*.

Constructors

descriptor(std::int64_t cluster_count = 2)

Creates a new instance of the class with the given `cluster_count`.

Properties

std::int64_t **cluster_count**

The number of clusters k. **Default value**: 2.

Getter & Setter

```
std::int64_t get_cluster_count() const
descriptor & set_cluster_count(std::int64_t)
```

Invariants

cluster_count > 0

Method tags

```
namespace method {
   struct dense {};
   using by_default = dense;
} // namespace method
```

struct **dense**

Tag-type that denotes *dense* computational method.

using **by_default** = *dense*

Task tags

```
namespace task {
   struct init {};
   using by_default = init;
} // namespace task
```

struct **init**

Tag-type that parameterizes entities used for obtaining the initial K-Means centroids.

using **by_default** = *init*

Alias tag-type for the initialization task.

Computing `compute(...)`

Input

```
template <typename Task = task::by_default>
class compute_input {
public:

   compute_input(const table& data = table{});

   const table& get_data() const;
   compute_input& set_data(const table&);
};
```

template<typename **Task** = task::*by_default*>
class **compute_input**

Template Parameters

Task – Tag-type that specifies type of the problem to solve. Can be `task::init`.

Constructors

compute_input(const *table* &data = *table*{})

Creates a new instance of the class with the given `data`.

Properties

const *table* &**data**

An $n \times p$ table with the data to be clustered, where each row stores one feature vector. **Default value**: table{}.

Getter & Setter

```
const table & get_data() const
compute_input & set_data(const table &)
```

Result

```
template <typename Task = task::by_default>
class compute_result {
public:

   compute_result();

   const table& get_centroids() const;
};
```

template<typename **Task** = task::*by_default*>
class **compute_result**

Template Parameters

Task – Tag-type that specifies type of the problem to solve. Can be `task::clustering`.

Constructors

compute_result()

Creates a new instance of the class with the default property values.

Public Methods

const *table* &**get_centroids**() const

A $k \times p$ table with the initial centroids. Each row of the table stores one centroid.

Operation

```
template <typename Float, typename Method, typename Task>
compute_result<Task> compute(const descriptor<Float, Method, Task>& desc,
                             const compute_input<Task>& input);
```

template<typename **Float**, typename **Method**, typename **Task**>
compute_result<*Task*> **compute**(const *descriptor*<*Float*, *Method*, *Task*> &desc, const *compute_input*<*Task*> &input)

Runs the computing operation for K-Means initialization. For more details, see `oneapi::dal::compute`.

Template Parameters

- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
- **Method** – Tag-type that specifies an implementation of K-Means Initialization algorithm.
- **Task** – Tag-type that specifies type of the problem to solve. Can be `task::init`.

Parameters

- **desc** – The descriptor of the algorithm.
- **input** – Input data for the computing operation.

Preconditions

```
input.data.has_data == true
input.data.row_count == desc.cluster_count
```

Postconditions

```
result.centroids.has_data == true
result.centroids.row_count == desc.cluster_count
result.centroids.column_count == input.data.column_count
```

7.7.2 Nearest Neighbors (kNN)

k-Nearest Neighbors Classification (k-NN)

k-NN *classification* algorithm infers the class for the new feature vector by computing majority vote of the k nearest observations from the training set.

Operation	Computational methods		Programming Interface		
Training	*Brute-force*	*k-d tree*	*train(...)*	*train_input*	*train_result*
Inference	*Brute-force*	*k-d tree*	*infer(...)*	*infer_input*	*infer_result*

Mathematical formulation

Training

Let $X = \{x_1, \ldots, x_n\}$ be the training set of p-dimensional feature vectors, let $Y = \{y_1, \ldots, y_n\}$ be the set of class labels, where $y_i \in \{0, \ldots, c-1\}$, $1 \leq i \leq n$. Given X, Y and the number of nearest neighbors k, the problem is to build a model that allows distance computation between the feature vectors in training and inference sets at the inference stage.

Training method: *brute-force*

The training operation produces the model that stores all the feature vectors from the initial training set X.

Training method: *k-d tree*

The training operation builds a k-d tree that partitions the training set X (for more details, see *k-d Tree*).

Inference

Let $X' = \{x'_1, \ldots, x'_m\}$ be the inference set of p-dimensional feature vectors. Given X', the model produced at the training stage and the number of nearest neighbors k, the problem is to predict the label y'_j for each x'_j, $1 \le j \le m$, by performing the following steps:

1. Identify the set $N(x'_j) \subseteq X$ of the k feature vectors in the training set that are nearest to x'_j with respect to the Euclidean distance.
2. Estimate the conditional probability for the l-th class as the fraction of vectors in $N(x'_j)$ whose labels y_j are equal to l:

$$P_{jl} = \frac{1}{|N(x'_j)|}\Big|\{x_r \in N(x'_j) : y_r = l\}\Big|, \quad 1 \le j \le m,\ 0 \le l < c. \tag{7.1}$$

3. Predict the class that has the highest probability for the feature vector x'_j:

$$y'_j = \arg\max_{0 \le l < c} P_{jl}, \quad 1 \le j \le m. \tag{7.2}$$

Inference method: *brute-force*

Brute-force inference method determines the set $N(x'_j)$ of the nearest feature vectors by iterating over all the pairs (x'_j, x_i) in the implementation defined order, $1 \le i \le n$, $1 \le j \le m$. The final prediction is computed according to the equations (7.1) and (7.2).

Inference method: *k-d tree*

K-d tree inference method traverses the k-d tree to find feature vectors associated with a leaf node that are closest to x'_j, $1 \le j \le m$. The set $\tilde{n}(x'_j)$ of the currently-known nearest k-th neighbors is progressively updated during tree traversal. The search algorithm limits exploration of the nodes for which the distance between the x'_j and respective part of the feature space is not less than the distance between x'_j and the most distant feature vector from $\tilde{n}(x'_j)$. Once tree traversal is finished, $\tilde{n}(x'_j) \equiv N(x'_j)$. The final prediction is computed according to the equations (7.1) and (7.2).

Usage example

Training

```
knn::model<> run_training(const table& data,
                          const table& labels) {
   const std::int64_t class_count = 10;
   const std::int64_t neighbor_count = 5;
   const auto knn_desc = knn::descriptor<float>{class_count, neighbor_count};

   const auto result = train(knn_desc, data, labels);

   return result.get_model();
}
```

Inference

```
table run_inference(const knn::model<>& model,
                    const table& new_data) {
   const std::int64_t class_count = 10;
   const std::int64_t neighbor_count = 5;
   const auto knn_desc = knn::descriptor<float>{class_count, neighbor_count};

   const auto result = infer(knn_desc, model, new_data);

   print_table("labels", result.get_labels());
}
```

Programming Interface

All types and functions in this section shall be declared in the `oneapi::dal::knn` namespace and be available via inclusion of the `oneapi/dal/algo/knn.hpp` header file.

Descriptor

```
template <typename Float = float,
          typename Method = method::by_default,
          typename Task = task::by_default>
class descriptor {
public:
   explicit descriptor(std::int64_t class_count,
                       std::int64_t neighbor_count);

   std::int64_t get_class_count() const;
   descriptor& set_class_count(std::int64_t);

   std::int64_t get_neighbor_count() const;
   descriptor& set_neighbor_count(std::int64_t);
};
```

template<typename **Float** = float, typename **Method** = method::*by_default*, typename **Task** = task::*by_default*>
class **descriptor**

Template Parameters

- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
- **Method** – Tag-type that specifies an implementation of algorithm. Can be `method::`*`bruteforce`* or `method::`*`kd_tree`*.
- **Task** – Tag-type that specifies type of the problem to solve. Can be `task::`*`classification`*.

Constructors

descriptor(std::int64_t class_count, std::int64_t neighbor_count)

Creates a new instance of the class with the given `class_count` and `neighbor_count` property values.

Properties

std::int64_t **neighbor_count**

The number of neighbors k.

Getter & Setter

```
std::int64_t get_neighbor_count() const
descriptor & set_neighbor_count(std::int64_t)
```

Invariants

neighbor_count > 0

std::int64_t **class_count**

The number of classes c.

Getter & Setter

```
std::int64_t get_class_count() const
descriptor & set_class_count(std::int64_t)
```

Invariants

class_count > 1

Method tags

```
namespace method {
   struct bruteforce {};
   struct kd_tree {};
   using by_default = bruteforce;
} // namespace method
```

struct **bruteforce**

Tag-type that denotes *brute-force* computational method.

struct **kd_tree**

Tag-type that denotes k-d tree computational method.

using **by_default** = *bruteforce*

Alias tag-type for *brute-force* computational method.

Task tags

```
namespace task {
   struct classification {};
   using by_default = classification;
} // namespace task
```

struct **classification**

Tag-type that parameterizes entities used for solving *classification problem*.

using **by_default** = *classification*

Alias tag-type for classification task.

Model

```
template <typename Task = task::by_default>
class model {
public:
   model();
};
```

template<typename **Task** = task::*by_default*>
class **model**

Template Parameters
Task – Tag-type that specifies type of the problem to solve. Can be `task::classification`.

Constructors

model()
Creates a new instance of the class with the default property values.

Training `train(...)`

Input

```
template <typename Task = task::by_default>
class train_input {
public:
   train_input(const table& data = table{},
               const table& labels = table{});

   const table& get_data() const;
   train_input& set_data(const table&);

   const table& get_labels() const;
   train_input& set_labels(const table&);
};
```

template<typename **Task** = task::*by_default*>
class **train_input**

Template Parameters
Task – Tag-type that specifies type of the problem to solve. Can be `task::classification`.

Constructors

train_input(const *table* &data = *table*{}, const *table* &labels = *table*{})
Creates a new instance of the class with the given `data` and `labels` property values.

Properties

const *table* &**data**
The training set X. **Default value**: table{}.

Getter & Setter

```
const table & get_data() const
```

```
train_input & set_data(const table &)
```

const *table* &**labels**

Vector of labels y for the training set X. **Default value**: table{}.

Getter & Setter

```
const table & get_labels() const
train_input & set_labels(const table &)
```

Result

```
template <typename Task = task::by_default>
class train_result {
public:
   train_result();

   const model<Task>& get_model() const;
};
```

template<typename **Task** = task::*by_default*>
class **train_result**

Template Parameters

Task – Tag-type that specifies type of the problem to solve. Can be `task::`*`classification`*.

Constructors

train_result()

Creates a new instance of the class with the default property values.

Public Methods

const *model*<Task> &**get_model**() const

The trained k-NN model.

Operation

```
template <typename Float, typename Method, typename Task>
train_result<Task> train(const descriptor<Float, Method, Task>& desc,
                         const train_input<Task>& input);
```

template<typename **Float**, typename **Method**, typename **Task**>
train_result<*Task*> **train**(const *descriptor*<*Float*, *Method*, *Task*> &desc, const *train_input*<*Task*> &input)

Runs the training operation for k-NN classifier. For more details see `oneapi::dal::train`.

Template Parameters

- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
- **Method** – Tag-type that specifies an implementation of algorithm. Can be `method::`*`bruteforce`* or `method::`*`kd_tree`*.
- **Task** – Tag-type that specifies type of the problem to solve. Can be `task::`*`classification`*.

Parameters

- **desc** – Descriptor of the algorithm.
- **input** – Input data for the training operation.

Preconditions

```
input.data.has_data == true
input.labels.has_data == true
input.data.row_count == input.labels.row_count
input.labels.column_count == 1
input.labels[i] >= 0
input.labels[i] < desc.class_count
```

Inference `infer(...)`

Input

```
template <typename Task = task::by_default>
class infer_input {
public:
   infer_input(const model<Task>& m = model<Task>{},
               const table& data = table{});

   const model<Task>& get_model() const;
   infer_input& set_model(const model&);

   const table& get_data() const;
   infer_input& set_data(const table&);
};
```

template<typename **Task** = task::*by_default*>
class **infer_input**

Template Parameters

Task – Tag-type that specifies type of the problem to solve. Can be `task::classification`.

Constructors

infer_input(const *model*<Task> &m = *model*<Task>{}, const *table* &data = *table*{})

Creates a new instance of the class with the given `model` and `data` property values.

Properties

const *table* &**data**

The dataset for inference X'. **Default value**: table{}.

Getter & Setter

```
const table & get_data() const
infer_input & set_data(const table &)
```

const *model*<Task> &**model**

The trained k-NN model. **Default value**: model<Task>{}.

Getter & Setter

```
const model< Task > & get_model() const
infer_input & set_model(const model &)
```

Result

```
template <typename Task = task::by_default>
class infer_result {
public:
   infer_result();

   const table& get_labels() const;
};
```

template<typename **Task** = task::*by_default*>
class **infer_result**

Template Parameters

Task – Tag-type that specifies type of the problem to solve. Can be `task::`*`classification`*.

Constructors

infer_result()

Creates a new instance of the class with the default property values.

Public Methods

const *table* &**get_labels**() const

The predicted labels.

Operation

```
template <typename Float, typename Method, typename Task>
infer_result<Task> infer(const descriptor<Float, Method, Task>& desc,
                         const infer_input<Task>& input);
```

template<typename **Float**, typename **Method**, typename **Task**>
infer_result<*Task*> **infer**(const *descriptor*<*Float*, *Method*, *Task*> &desc, const *infer_input*<*Task*> &input)

Runs the inference operation for k-NN classifier. For more details see `oneapi::dal::infer`.

Template Parameters

- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
- **Method** – Tag-type that specifies an implementation of algorithm. Can be `method::`*`bruteforce`* or `method::`*`kd_tree`*.
- **Task** – Tag-type that specifies type of the problem to solve. Can be `task::`*`classification`*.

Parameters

- **desc** – Descriptor of the algorithm.
- **input** – Input data for the inference operation.

Preconditions

```
input.data.has_data == true
```

Postconditions

```
result.labels.row_count == input.data.row_count
result.labels.column_count == 1
result.labels[i] >= 0
result.labels[i] < desc.class_count
```

7.7.3 Decomposition

Principal Components Analysis (PCA)

Principal Component Analysis (PCA) is an algorithm for exploratory data analysis and *dimensionality reduction*. PCA transforms a set of feature vectors of possibly correlated features to a new set of uncorrelated features, called principal components. Principal components are the directions of the largest variance, that is, the directions where the data is mostly spread out.

Operation	Computational methods		Programming Interface		
Training	*Covariance*	*SVD*	*train(...)*	*train_input*	*train_result*
Inference	*Covariance*	*SVD*	*infer(...)*	*infer_input*	*infer_result*

Mathematical formulation

Training

Given the training set $X = \{x_1, \ldots, x_n\}$ of p-dimensional feature vectors and the number of principal components r, the problem is to compute r principal directions (p-dimensional eigenvectors [Lang87]) for the training set. The eigenvectors can be grouped into the $r \times p$ matrix T that contains one eigenvector in each row.

Training method: *Covariance*

This method uses eigenvalue decomposition of the covariance matrix to compute the principal components of the datasets. The method relies on the following steps:

1. Computation of the covariance matrix
2. Computation of the eigenvectors and eigenvalues
3. Formation of the matrices storing the results

Covariance matrix computation shall be performed in the following way:

1. Compute the vector-column of sums $s_i = \sum_{j=1}^{n} x_{i,j}, \quad 1 \leq i \leq p$.
2. Compute the cross-product $P = X^T X - s^T s$.
3. Compute the covariance matrix $\Sigma = \frac{1}{n-1} P$.

To compute eigenvalues λ_i and eigenvectors υ_i, the implementer can choose an arbitrary method such as [Ping14].

The final step is to sort the set of pairs (λ_i, υ_i) in the descending order by λ_i and form the resulting matrix $T = (\upsilon_{i,1}, \cdots, \upsilon_{i,r}), \quad 1 \leq i \leq p$. Additionally, the means and variances of the initial dataset shall be returned.

Training method: *SVD*

This method uses singular value decomposition of the dataset to compute its principal components. The method relies on the following steps:

1. Computation of the singular values and singular vectors
2. Formation of the matrices storing the results

To compute singular values λ_i and singular vectors u_i and v_i, the implementer can choose an arbitrary method such as [Demmel90].

The final step is to sort the set of pairs (λ_i, v_i) in the descending order by λ_i and form the resulting matrix $T = (v_{i,1}, \cdots, v_{i,r}), \quad 1 \leq i \leq p$. Additionally, the means and variances of the initial dataset shall be returned.

Sign-flip technique

Eigenvectors computed by some eigenvalue solvers are not uniquely defined due to sign ambiguity. To get the deterministic result, a sign-flip technique should be applied. One of the sign-flip techniques proposed in [Bro07] requires the following modification of matrix T:

$$\hat{T_i} = T_i \cdot \operatorname{sgn}(\max_{1 \leq j \leq p} |T_{ij}|), \quad 1 \leq i \leq r,$$

where T_i is i-th row, T_{ij} is the element in the i-th row and j-th column, $\operatorname{sgn}(\cdot)$ is the signum function,

$$\operatorname{sgn}(x) = \begin{cases} -1, & x < 0, \\ 0, & x = 0, \\ 1, & x > 0. \end{cases}$$

Note: The sign-flip technique described above is an example. oneDAL spec does not require implementation of this sign-flip technique. Implementer can choose an arbitrary technique that modifies the eigenvectors' signs.

Inference

Given the inference set $X' = \{x'_1, \ldots, x'_m\}$ of p-dimensional feature vectors and the $r \times p$ matrix T produced at the training stage, the problem is to transform X' to the set $X'' = \{x''_1, \ldots, x''_m\}$, where x''_j is an r-dimensional feature vector, $1 \leq j \leq m$.

The feature vector x''_j is computed through applying linear transformation [Lang87] defined by the matrix T to the feature vector x'_j,

$$x''_j = T x'_j, \quad 1 \leq j \leq m. \tag{7.3}$$

Inference methods: *Covariance* and *SVD*

Covariance and SVD inference methods compute x''_j according to (7.3).

Usage example

Training

```
pca::model<> run_training(const table& data) {
   const auto pca_desc = pca::descriptor<float>{}
      .set_component_count(5)
      .set_deterministic(true);

   const auto result = train(pca_desc, data);

   print_table("means", result.get_means());
   print_table("variances", result.get_variances());
   print_table("eigenvalues", result.get_eigenvalues());
   print_table("eigenvectors", result.get_eigenvectors());

   return result.get_model();
}
```

Inference

```
table run_inference(const pca::model<>& model,
                    const table& new_data) {
   const auto pca_desc = pca::descriptor<float>{}
      .set_component_count(model.get_component_count());

   const auto result = infer(pca_desc, model, new_data);

   print_table("labels", result.get_transformed_data());
}
```

Programming Interface

All types and functions in this section shall be declared in the `oneapi::dal::pca` namespace and be available via inclusion of the `oneapi/dal/algo/pca.hpp` header file.

Descriptor

```
template <typename Float = float,
          typename Method = method::by_default,
          typename Task = task::by_default>
class descriptor {
public:
   explicit descriptor(std::int64_t component_count = 0);

   int64_t get_component_count() const;
   descriptor& set_component_count(int64_t);
```

(continues on next page)

(continued from previous page)

```
   bool get_deterministic() const;
   descriptor& set_deterministic(bool);
};
```

template<typename **Float** = float, typename **Method** = method::*by_default*, typename **Task** = task::*by_default*>
class **descriptor**

Template Parameters

- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
- **Method** – Tag-type that specifies an implementation of algorithm. Can be `method::cov` or `method::svd`.
- **Task** – Tag-type that specifies type of the problem to solve. Can be `task::dim_reduction`.

Constructors

descriptor(std::int64_t component_count = 0)

Creates a new instance of the class with the given `component_count` property value.

Properties

int64_t **component_count**

The number of principal components r. If it is zero, the algorithm computes the eigenvectors for all features, $r = p$. **Default value**: 0.

Getter & Setter

```
int64_t get_component_count() const
descriptor & set_component_count(int64_t)
```

Invariants

component_count >= 0

bool **deterministic**

Specifies whether the algorithm applies the *Sign-flip technique*. If it is *true*, the directions of the eigenvectors must be deterministic. **Default value**: true.

Getter & Setter

```
bool get_deterministic() const
descriptor & set_deterministic(bool)
```

Method tags

```
namespace method {
   struct cov {};
   struct svd {};
   using by_default = cov;
} // namespace method
```

struct **cov**

Tag-type that denotes *Covariance* computational method.

struct **svd**

Tag-type that denotes *SVD* computational method.

using **by_default** = *cov*

Alias tag-type for *Covariance* computational method.

Task tags

```
namespace task {
   struct dim_reduction {};
   using by_default = dim_reduction;
} // namespace task
```

struct **dim_reduction**

Tag-type that parameterizes entities used for solving *dimensionality reduction problem*.

using **by_default** = *dim_reduction*

Alias tag-type for dimensionality reduction task.

Model

```
template <typename Task = task::by_default>
class model {
public:
   model();

   const table& get_eigenvectors() const;

   int64_t get_component_count() const;
};
```

template<typename **Task** = task::*by_default*>
class **model**

Template Parameters

Task – Tag-type that specifies type of the problem to solve. Can be `task::dim_reduction`.

Constructors

model()

Creates a new instance of the class with the default property values.

Public Methods

const *table* &**get_eigenvectors**() const

An $r \times p$ table with the eigenvectors. Each row contains one eigenvector.

int64_t **get_component_count**() const

The number of components r in the trained model.

Training `train(...)`

Input

```
template <typename Task = task::by_default>
class train_input {
public:
   train_input(const table& data = table{});

   const table& get_data() const;
   train_input& set_data(const table&);
};
```

template<typename **Task** = task::*by_default*>
class **train_input**

Template Parameters

Task – Tag-type that specifies type of the problem to solve. Can be `task::dim_reduction`.

Constructors

train_input(const *table* &data = *table*{})

Creates a new instance of the class with the given `data` property value.

Properties

const *table* &**data**

An $n \times p$ table with the training data, where each row stores one feature vector. **Default value**: table{}.

Getter & Setter

```
const table & get_data() const
train_input & set_data(const table &)
```

Result

```
template <typename Task = task::by_default>
class train_result {
public:
   train_result();

   const model<Task>& get_model() const;

   const table& get_means() const;

   const table& get_variances() const;

   const table& get_eigenvalues() const;

   const table& get_eigenvectors() const;
};
```

template<typename **Task** = task::*by_default*>

class **train_result**

Template Parameters

Task – Tag-type that specifies type of the problem to solve. Can be `task::dim_reduction`.

Constructors

train_result()

Creates a new instance of the class with the default property values.

Public Methods

const *model*<Task> &**get_model**() const

The trained PCA model.

const *table* &**get_means**() const

A $1 \times r$ table that contains the mean values for the first r features.

const *table* &**get_variances**() const

A $1 \times r$ table that contains the variances for the first r features.

const *table* &**get_eigenvalues**() const

A $1 \times r$ table that contains the eigenvalues for for the first r features.

const *table* &**get_eigenvectors**() const

An $r \times p$ table with the eigenvectors. Each row contains one eigenvector.

Operation

```
template <typename Float, typename Method, typename Task>
train_result<Task> train(const descriptor<Float, Method, Task>& desc,
                         const train_input<Task>& input);
```

template<typename **Float**, typename **Method**, typename **Task**>
train_result<*Task*> **train**(const *descriptor*<*Float*, *Method*, *Task*> &desc, const *train_input*<*Task*> &input)

Runs the training operation for PCA. For more details, see `oneapi::dal::train`.

Template Parameters

- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
- **Method** – Tag-type that specifies an implementation of algorithm. Can be `method::cov` or `method::svd`.
- **Task** – Tag-type that specifies type of the problem to solve. Can be `task::dim_reduction`.

Parameters

- **desc** – Descriptor of the algorithm.
- **input** – Input data for the training operation.

Preconditions

input.data.has_data == true
input.data.column_count >= *desc*.component_count

Postconditions

```
result.means.row_count == 1
result.means.column_count == desc.component_count
result.variances.row_count == 1
result.variances.column_count == desc.component_count
result.variances[i] >= 0.0
result.eigenvalues.row_count == 1
result.eigenvalues.column_count == desc.component_count
result.model.eigenvectors.row_count == 1
result.model.eigenvectors.column_count == desc.component_count
```

Inference `infer(...)`

Input

```
template <typename Task = task::by_default>
class infer_input {
public:
   infer_input(const model<Task>& m = model<Task>{},
               const table& data = table{});

   const model<Task>& get_model() const;
   infer_input& set_model(const model&);

   const table& get_data() const;
   infer_input& set_data(const table&);
};
```

template<typename **Task** = task::*by_default*>
class **infer_input**

Template Parameters

Task – Tag-type that specifies type of the problem to solve. Can be `task::dim_reduction`.

Constructors

infer_input(const *model*<Task> &m = *model*<Task>{}, const *table* &data = *table*{})

Creates a new instance of the class with the given `model` and `data` property values.

Properties

const *table* &**data**

The dataset for inference X'. **Default value**: table{}.

Getter & Setter

```
const table & get_data() const
infer_input & set_data(const table &)
```

const *model*<Task> &**model**

The trained PCA model. **Default value**: model<Task>{}.

Getter & Setter

```
const model< Task > & get_model() const
infer_input & set_model(const model &)
```

Result

```
template <typename Task = task::by_default>
class infer_result {
public:
   infer_result();

   const table& get_transformed_data() const;
};
```

template<typename **Task** = task::*by_default*>
class **infer_result**

Template Parameters

Task – Tag-type that specifies type of the problem to solve. Can be `task::dim_reduction`.

Constructors

infer_result()

Creates a new instance of the class with the default property values.

Public Methods

const *table* &**get_transformed_data**() const

An $n \times r$ table that contains data projected to the r principal components.

Operation

```
template <typename Float, typename Method, typename Task>
infer_result<Task> infer(const descriptor<Float, Method, Task>& desc,
                         const infer_input<Task>& input);
```

template<typename **Float**, typename **Method**, typename **Task**>
infer_result<*Task*> **infer**(const *descriptor*<*Float*, *Method*, *Task*> &desc, const *infer_input*<*Task*> &input)

Runs the inference operation for PCA. For more details see `oneapi::dal::infer`.

Template Parameters

- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
- **Method** – Tag-type that specifies an implementation of algorithm. Can be `method::cov` or `method::svd`.
- **Task** – Tag-type that specifies type of the problem to solve. Can be `task::dim_reduction`.

Parameters

- **desc** – Descriptor of the algorithm.
- **input** – Input data for the inference operation.

Preconditions

input.data.has_data == true
input.model.eigenvectors.row_count == *desc*.component_count
input.model.eigenvectors.column_count == *input*.data.column_count

Postconditions

```
result.transformed_data.row_count == input.data.row_count
result.transformed_data.column_count == desc.component_count
```

7.8 Appendix

7.8.1 k-d Tree

k-d tree is a space-partitioning binary tree [Bentley80], where

- Each non-leaf node induces the hyperplane that splits the feature space into two parts. To define the splitting hyperplane explicitly, a non-leaf node stores the identifier of the feature (that defines axis in the feature space) and *a cut-point*
- Each leaf node of the tree has an associated subset (*a bucket*) of elements of the training data set. Feature vectors from a bucket belong to the region of the space defined by tree nodes on the path from the root node to the respective leaf.

Related terms

A cut-point
A feature value that corresponds to a non-leaf node of a k-d tree and defines the splitting hyperplane orthogonal to the axis specified by the given feature.

7.9 Bibliography

For more information about algorithms implemented in oneAPI Data Analytics Library (oneDAL), refer to the following publications:

CHAPTER

EIGHT

ONETBB

8.1 General Information

8.1.1 Introduction

[intro]

This document specifies requirements for implementations of oneAPI Threading Building Blocks (oneTBB).

oneTBB is a programming model for scalable parallel programming using standard ISO C++ code. A program uses oneTBB to specify logical parallelism in algorithms, while a oneTBB implementation maps that parallelism onto execution threads.

oneTBB employs generic programming via C++ templates, with most of its interfaces defined by requirements on types and not specific types. Generic programming makes oneTBB flexible yet efficient through customizing APIs to specific needs of an application.

Here is the list of specific requirements for oneTBB implementations:

- An implementation should use the C++11 version of the standard and should not require newer versions except where explicitly specified; it also should not require any non-standard language extensions.
- An implementation can use platform-specific APIs if they are compatible with the C++ execution and memory models. For example, a platform-specific implementation of threads can be used if that implementation provides the same execution guarantees as C++ threads.
- An implementation should support execution on single-core and multi-core CPUs, including those that provide simultaneous multithreading capabilities.
- On CPU, an implementation should support nested parallelism to enable building larger parallel components from smaller ones.

8.1.2 Notational Conventions

[notational_conventions]

The following conventions are used in this document.

Convention	Explanation	Example
Italic	Used for introducing new terms, denotation of terms, placeholders, or titles of documents.	The filename consists of the *basename* and the *extension*. For more information, refer to the *TBB Developer Guide*.

`Monospace`	Indicates directory paths and filenames, commands and command line options, function names, methods, classes, data structures in body text, source code.	`oneapi/tbb.h`
		`\alt\include`
		Use the `okCreateObjs()` function to…
		`printf("hello, world\n");`

`Monospace italic`	Indicates source code placeholders.	blocked_range<*Type*>
`Monospace bold`	Emphasizes parts of source code.	`x = ( h > 0 ? sizeof(m) : 0xF ) + min;`
[ ]	Square brackets indicate that the items enclosed in brackets are optional.	Fa[c] Indicates `Fa` or `Fac`.
{ | }	Braces and vertical bars indicate the choice of one item from a selection of two or more items.	X{K | W | P} Indicates `XK`, `XW`, or `XP`.
"[" "]" "{"" }" "|"	Writing a metacharacter in quotation marks negates the syntactical meaning stated above; the character is taken as a literal.	"[" X "]" [Y] Denotes the letter X enclosed in brackets, optionally followed by the letter Y.
…	The ellipsis indicates that the previous item can be repeated several times.	**filename** … Indicates that one or more filenames can be specified.
,…	The ellipsis preceded by a comma indicates that the previous item can be repeated several times, separated by commas.	**word** ,… Indicates that one or more words can be specified. If more than one word is specified, the words are comma-separated.

Class members are summarized by informal class declarations that describe the class as it seems to clients, not how it is actually implemented. For example, here is an informal declaration of class `Foo`:

```
class Foo {
public:
     int x();
     int y;
     ~Foo();
};
```

The actual implementation might look like:

```
namespace internal {
     class FooBase  {
     protected:
             int x();
     };

     class Foo_v3: protected FooBase {
     private:
```

(continues on next page)

(continued from previous page)

```
        int internal_stuff;
    public:
        using FooBase::x;
        int y;
    };
}

typedef internal::Foo_v3 Foo;
```

The example shows two cases where the actual implementation departs from the informal declaration:

- `Foo` is actually a typedef to `Foo_v3`.
- Method `x()` is inherited from a protected base class.
- The destructor is an implicit method generated by the compiler.

The informal declarations are intended to show you what you need to know to use the class without the distraction of irrelevant clutter particular to the implementation.

8.1.3 Identifiers

[identifiers]

This section describes the identifier conventions used by oneTBB.

Case

The identifier convention in the library follows the style of the ISO C++ standard library. Identifiers are written in underscore_style, and concepts - in PascalCase.

Reserved Identifier Prefixes

The library reserves the `__TBB` prefix for internal identifiers and macros that should never be directly referenced by your code.

8.1.4 Named Requirements

[named_requirements]

This section describes named requirements used in the oneTBB Specification.

A *named requirement* is a set of requirements on a type. The requirements may be syntactic or semantic. The *named_requirement* term is similar to "Requirements on types and expressions" term which is defined by the ISO C++ Standard (chapter "Library Introduction") or "Named Requirements" section on the cppreference.com site.

For example, the named requirement of *sortable* could be defined as a set of requirements that enable an array to be sorted. A type `T` would be *sortable* if:

- `x < y` returns a boolean value, and represents a total order on items of type `T`.
- `swap(x,y)` swaps items `x` and `y`

You can write a sorting template function in C++ that sorts an array of any type that is *sortable*.

Two approaches for defining named requirements are *valid expressions* and *pseudo-signatures*. The ISO C++ standard follows the valid *expressions* approach, which shows what the usage pattern looks like for a requirement. It has the drawback of relegating important details to notational conventions. This document uses pseudo-signatures because they are concise and can be cut-and-pasted for an initial implementation.

For example, the table below shows pseudo-signatures for a *sortable* type `T`:

Sortable Requirements : Pseudo-Signature, Semantics

bool **`operator<`**`(const T &x, const T &y)`

Compare x and y.

void **`swap`**`(T &x, T &y)`

Swap x and y.

A real signature may differ from the pseudo-signature that it implements in ways where implicit conversions would deal with the difference. For an example type `U`, the real signature that implements `operator<` in the table above can be expressed as `int operator<( U x, U y )`, because C++ permits implicit conversion from `int` to `bool`, and implicit conversion from `U` to (`const U&`). Similarly, the real signature `bool operator<( U& x, U& y )` is acceptable because C++ permits implicit addition of a const qualifier to a reference type.

Algorithms

Range

[req.range]

A *Range* can be recursively subdivided into two parts. Subdivision is done by calling *splitting constructor* of a *Range*. There are two types of splitting constructors:

- Basic splitting constructor. In this constructor, it is recommended that the division is done into nearly equal parts, but it is not required. Splitting as evenly as possible typically yields the best parallelism.
- Proportional splitting constructor. This constructor is optional and can be omitted. When using this type of constructor, for the best results, follow the given proportion with rounding to the nearest integer if necessary.

Ideally, a range is recursively splittable until the parts represent portions of work that are more efficient to execute serially rather than split further. The amount of work represented by *Range* typically depends on higher level context, therefore a typical type that models a *Range* should provide a way to control the degree of splitting. For example, the template class *blocked_range* has the *grainsize* parameter that specifies the biggest range considered indivisible.

If the set of values has a sense of direction, by convention the splitting constructor should construct the second part of the range and update its argument to be the first part of the range. This causes the *parallel_for*, *parallel_reduce*, and *parallel_scan* algorithms, when running sequentially, to work across a range in the increasing order, which is typical of an ordinary sequential loop.

Because a *Range* declares splitting and copy constructors, the default constructor for it is not generated automatically. You need to explicitly define the default constructor or add any other constructor to create an instance of a *Range* type in the program.

A type *R* meets *Range* if it satisfies the following requirements:

Range Requirements: Pseudo-Signature, Semantics

R`::R`(const *R*&)

Copy constructor.

`R::~R`()

Destructor.

bool `R::empty`() const

True if range is empty.

bool `R::is_divisible`() const

True if range can be partitioned into two subranges.

R`::R`(*R* &r, split)

Basic splitting constructor. Splits `r` into two subranges.

R`::R`(*R* &r, *proportional_split* proportion)

Optional. Proportional splitting constructor. Splits `r` into two subranges in accordance with `proportion`.

See also:

- *blocked_range class*
- *blocked_range2d class*
- *blocked_range3d class*
- *parallel_reduce algorithm*
- *parallel_for algorithm*
- *split class*

Splittable

[req.splittable]

A type is splittable if it has a *splitting constructor* that allows an instance to be split into two pieces. The splitting constructor takes as arguments a reference to the original object, and a dummy argument of type `split`, which is defined by the library. The dummy argument distinguishes the splitting constructor from a copy constructor. After the constructor runs, `x` and the newly constructed object should represent the two pieces of the original `x`. The library uses splitting constructors in two contexts:

- *Partitioning* a range into two subranges that can be processed concurrently.
- *Forking* a body (function object) into two bodies that can run concurrently.

Types that meet the *Range requirements* may additionally define an optional *proportional splitting constructor*, distinguished by an argument of type *proportional_split Class*.

A type *X* satisfies *Splittable* if it meets the following requirements:

Splittable Requirements: Pseudo-Signature, Semantics

X`::X`(*X* &x, split)

Split `x` into `x` and newly constructed object.

See also:

- *Range requirements*

ParallelForBody

[req.parallel_for_body]

A type *Body* satisfies *ParallelForBody* if it meets the following requirements:

ParallelForBody Requirements: Pseudo-Signature, Semantics

Body`::Body`(const *Body*&)

Copy constructor.

`Body::~Body()`

Destructor.

void `Body::operator()`(Range &range) const

Applies body to a range. `Range` type must meet the *Range requirements*.

See also:

- *parallel_for algorithm*

ParallelForFunc

[req.parallel_for_func]

A type *F* satisfies *ParallelForFunc* if it meets the following requirements:

ParallelForFunc Requirements: Pseudo-Signature, Semantics

void `F::operator()`(Index index) const

Applies the function to the index. `Index` type must be the same as corresponding template parameter of the *parallel_for algorithm*.

See also:

- *parallel_for algorithm*
- *ParallelForIndex named requirement*

ParallelForIndex

[req.parallel_for_index]

A type *Index* satisfies *ParallelForIndex* if it meets the following requirements:

ParallelForIndex Requirements: Pseudo-Signature, Semantics

Index`::Index`(int)

Constructor from an `int` value.

Index`::Index`(const *Index*&)

Copy constructor.

`Index::~Index()`
: Destructor.

Index &**operator=**(const Index&)
: Assignment.

Index &**operator++**()
: Adjust `*this` to the next value.

bool **operator<**(const Index &i, const Index &j)
: Value of *i* precedes value of *j*.

bool **operator<=**(const Index &i, const Index &j)
: Value of *i* precedes or equal to the value of *j*.

D **operator-**(const Index &i, const Index &j)
: Number of values in range `[i,j)`.

Index **operator+**(const Index &i, const Index &j)
: Sum of *i* and *j* values.

Index **operator+**(const Index &i, D k)
: *k*-th value after *i*.

Index **operator***(const Index &i, const Index &j)
: Multiplication of *i* and *j* values.

Index **operator/**(const Index &i, const Index &j)
: Quotient of *i* and *j* values.

`D` is the type of the expression `j-i`. It can be any integral type that is convertible to `size_t`. Examples that model the Index requirements are integral types and pointers.

NOTE: It is recommended to use integral types as `ParallelForIndex`. See the `[basic.fundamental]` section of the ISO C++ Standard for information about integral types.

See also:

- *parallel_for algorithm*

ParallelReduceBody

[req.parallel_reduce_body]

A type *Body* satisfies *ParallelReduceBody* if it meets the following requirements:

ParallelReduceBody Requirements: Pseudo-Signature, Semantics

Body`::`**Body**(*Body*&, split)
: Splitting constructor. Must be able to run concurrently with `operator()` and method `join`.

`Body::~Body()`
: Destructor.

void `Body::`**operator()**(const Range &range)
: Accumulates result for a subrange. `Range` type must meet the *Range requirements*.

void `Body::join`(Body &rhs)

Joins results. The result in rhs should be merged into the result of `this`.

See also:

- *parallel_reduce algorithm*
- *parallel_determinstic_reduce algorithm*

ParallelReduceFunc

[req.parallel_reduce_body]

A type *Func* satisfies *ParallelReduceFunc* if it meets the following requirements:

ParallelReduceFunc Requirements: Pseudo-Signature, Semantics

Value `Func::operator()`(const Range &range, const Value &x) const

Accumulates result for a subrange, starting with initial value `x`. `Range` type must meet the *Range requirements*. `Value` type must be the same as a corresponding template parameter for the *parallel_reduce algorithm* algorithm.

See also:

- *parallel_reduce algorithm*
- *parallel_determinstic_reduce algorithm*

ParallelReduceReduction

[req.parallel_reduce_reduction]

A type *Reduction* satisfies *ParallelReduceReduction* if it meets the following requirements:

ParallelReduceReduction Requirements: Pseudo-Signature, Semantics

Value `Reduction::operator()`(const Value &x, const Value &y) const

Combines results `x` and `y`. `Value` type must be the same as a corresponding template parameter for the *parallel_reduce algorithm* algorithm.

See also:

- *parallel_reduce algorithm*
- *parallel_determinstic_reduce algorithm*

ParallelForEachBody

[req.parallel_for_each_body]

A type *Body* satisfies *ParallelForBody* if it meets the *Function Objects* requirements described in the [function.objects] section of the ISO C++ standard. It should also meet one of the following requirements:

ParallelForEachBody Requirements: Pseudo-Signature, Semantics

`Body::operator()` (ItemType item) const

Process the received item.

`Body::operator()` (ItemType item, oneapi::tbb::feeder<ItemType> &feeder) const

Process the received item. May invoke the `feeder.add(T)` function to spawn additional items.

Note: `ItemType` may be optionally passed to `Body::operator()` by reference. `const` and `volatile` type qualifiers are also applicable.

Terms

- `iterator` determines the type of the iterator passed into the `parallel_for_each` algorithm, which is `decltype(std::begin(c))` for the overloads that accept the `Container` template argument or `InputIterator`.
- `value_type` - the type `std::iterator_traits<iterator>::value_type`.
- `reference` - the type `std::iterator_traits<iterator>::reference`.

`oneapi::tbb::parallel_for_each` requires the `Body::operator()` call with an object of the `reference` type to be well-formed if the `iterator` meets the *Forward iterator* requirements described in the [forward.iterators] section of the ISO C++ Standard.

oneapi::tbb::parallel_for_each algorithm requires the `Body::operator()` call with an object of type `const value_type&` or `value_type&&` to be well-formed if following requirements are met:

- the iterator meets the *Input iterator* requirements described in the [input.iterators] section of the ISO C++ Standard
- the iterator does not meet the *Forward iterator* requirements described in the [forward.iterators] section of the ISO C++ Standard

Caution: If the `Body` only takes non-const lvalue reference to the `value_type`, the requirements described above are violated, and the program can be ill-formed.

Additional elements submitted into `oneapi::tbb::parallel_for_each` through the `feeder::add` are passed to the `Body` as rvalues. In this case, the corresponding execution of the `Body` is required to be well-formed.

See also:

- *parallel_for_each algorithm*
- *feeder class*

ContainerBasedSequence

[req.container_based_sequence]

A type *C* satisfies *ContainerBasedSequence* if it meets the following requirements:

ContainerBasedSequence Requirements: Pseudo-Signature, Semantics

> **Note:** In this page `c` is an object of type (possibly `const`) `C`.
>
> Templates that use the named requirement can impose stricter requirements on the iterator concept.

`std::begin(c)`

Returns an input iterator to the beginning of the sequence represented by `c`.

`std::end(c)`

Returns an input iterator one past the end of the sequence represented by `c`.

See also:

- *parallel_for_each algorithm*
- *parallel_sort algorithm*

ParallelScanBody

[req.parallel_scan]

A type *Body* satisfies *ParallelScanBody* if it meets the following requirements:

ParallelScanBody Requirements: Pseudo-Signature, Semantics

void `Body::operator()`(const Range &r, pre_scan_tag)

Accumulates summary for range `r`. For example, when computing a running sum of an array, the summary for a range `r` is the sum of the array elements corresponding to `r`.

void `Body::operator()`(const Range &r, final_scan_tag)

Computes scan result and summary for range `r`.

Body`::Body(`*Body* &b, split`)`

Splits `b` so that `this` and `b` can accumulate summaries separately.

void `Body::reverse_join`(Body &b)

Merges the summary accumulated by `b` into the summary accumulated by `this`, where `this` was created earlier from `b` by splitting constructor.

void `Body::assign`(Body &b)

Assigns summary of `b` to `this`.

See also:

- *parallel_scan algorithm*

ParallelScanCombine

[req.parallel_scan_combine]

A type *Combine* satisfies *ParallelScanCombine* if it meets the following requirements:

ParallelScanCombine Requirements: Pseudo-Signature, Semantics

Value `Combine::operator()`(const Value &left, const Value &right) const

Combines summaries `left` and `right` and returns the result `Value` type must be the same as a corresponding template parameter for the `parallel_scan` algorithm.

See also:

- *parallel_scan algorithm*

ParallelScanFunc

[req.parallel_scan_func]

A type *Scan* satisfies *ParallelScanFunc* if it meets the following requirements:

ParallelScanFunc Requirements: Pseudo-Signature, Semantics

Value `Scan::operator()`(const Range &r, const Value &sum, bool is_final) const

Starting with `sum`, computes the summary and, for `is_final == true`, the scan result for range `r`. Returns the computed summary. `Value` type must be the same as a corresponding template parameter for the `parallel_scan` algorithm.

See also:

- *parallel_scan algorithm*

BlockedRangeValue

[req.blocked_range_value]

A type *Value* satisfies *BlockedRangeValue* if it meets the following requirements:

BlockedRangeValue Requirements: Pseudo-Signature, Semantics

Value`::Value`(const *Value*&)

Copy constructor.

`Value::~Value`()

Destructor.

void `operator=`(const Value&)

Assignment.

Note: The return type `void` in the pseudo-signature denotes that `operator=` is not required to return a value. The actual `operator=` can return a value, which will be ignored by `blocked_range` .

---

bool **operator<**(const Value &i, const Value &j)

Value *i* precedes value *j*.

D **operator-**(const Value &i, const Value &j)

Number of values in range `[i,j)`.

Value **operator+**(const Value &i, D k)

k-th value after *i*.

`D` is the type of the expression `j-i`. It can be any integral type that is convertible to `size_t`. Examples that model the Value requirements are integral types, pointers, and STL random-access iterators whose difference can be implicitly converted to a `size_t`.

See also:

- *blocked_range class*
- *blocked_range2d class*
- *blocked_range3d class*
- *parallel_reduce algorithm*
- *parallel_for algorithm*

FilterBody

[req.filter_body]

A type *Body* should meet one of the following requirements depending on the filter type:

MiddleFilterBody Requirements: Pseudo-Signature, Semantics

OutputType `Body::operator()`(InputType item) const

Processes the received item and then returns it.

FirstFilterBody Requirements: Pseudo-Signature, Semantics

OutputType `Body::operator()`(oneapi::tbb::flow_control &fc) const

Returns the next item from an input stream. Calls `fc.stop()` at the end of an input stream.

LastFilterBody Requirements: Pseudo-Signature, Semantics

void `Body::operator()`(InputType item) const

Processes the received item.

SingleFilterBody Requirements: Pseudo-Signature, Semantics

void `Body::operator()`(oneapi::tbb::flow_control &fc) const

Processes an element from an input stream. Calls `fc.stop()` at the end of an input stream.

See also:

- *filter class*

Mutexes

Mutex

[req.mutex]

The mutexes and locks have relatively spartan interfaces that are designed for high performance. The interfaces enforce the *scoped locking pattern*, which is widely used in C++ libraries because:

- Does not require to remember to release the lock
- Releases the lock if an exception is thrown out of the mutual exclusion region protected by the lock

There are two parts of the pattern: a *mutex* object, for which construction of a *lock* object acquires a lock on the mutex and destruction of the *lock* object releases the lock. Here is an example:

```
{
    // Construction of myLock acquires lock on myMutex
    M::scoped_lock myLock( myMutex );
    // ... actions to be performed while holding the lock ...
    // Destruction of myLock releases lock on myMutex
}
```

If the actions throw an exception, the lock is automatically released as the block is exited.

```
class M {
    // Implementation specifics
    // ...

    // Represents acquisition of a mutex
    class scoped_lock {
    public:
        constexpr scoped_lock() noexcept;
        scoped_lock(M& m);
        ~scoped_lock();

        scoped_lock(const scoped_lock&) = delete;
        scoped_lock& operator=(const scoped_lock&) = delete;

        void acquire(M& m);
        bool try_acquire(M& m);
        void release();
    };
};
```

A type *M* satisfies the *Mutex* requirements if it meets the following conditions:

type `M::scoped_lock`

Corresponding scoped lock type.

`M::scoped_lock()`

Constructs `scoped_lock` without acquiring mutex.

`M::scoped_lock(M&)`

Constructs `scoped_lock` and acquire the lock on a provided mutex.

`M::~scoped_lock()`
: Releases a lock (if acquired).

void `M::scoped_lock::acquire(M&)`
: Acquires a lock on a provided mutex.

bool `M::scoped_lock::try_acquire(M&)`
: Attempts to acquire a lock on a provided mutex. Returns true if the lock is acquired, false otherwise.

void `M::scoped_lock::release()`
: Releases an acquired lock.

Also, the `Mutex` type requires a set of traits to be defined:

static constexpr bool `M::is_rw_mutex`
: True if mutex is a reader-writer mutex; false, otherwise.

static constexpr bool `M::is_recursive_mutex`
: True if mutex is a recursive mutex; false, otherwise.

static constexpr bool `M::is_fair_mutex`
: True if mutex is fair; false, otherwise.

A mutex type and an `M::scoped_lock` type are neither copyable nor movable.

The following table summarizes the library classes that model the `Mutex` requirement and provided guarantees.

Table 1: Provided guarantees for Mutexes that model the Mutex requirement

.	Fair	Reentrant
`mutex`	No	No
`spin_mutex`	No	No
`speculative_spin_mutex`	No	No
`queuing_mutex`	Yes	No
`null_mutex`	Yes	Yes

Note: Implementation is allowed to have an opposite guarantees (positive) in case of negative statements from the table above.

See the *oneAPI Threading Building Blocks Developer Guide* for description of the mutex properties and the rationale for null mutexes.

See also:

- *mutex*
- *spin_mutex*
- *speculative_spin_mutex*
- *queuing_mutex*
- *null_mutex*

ReaderWriterMutex

[req.rw_mutex]

The *ReaderWriterMutex* requirement extends the *Mutex Requirement* to include the notion of reader-writer locks. It introduces a boolean parameter `write` that specifies whether a writer lock (`write = true`) or reader lock (`write = false`) is being requested. Multiple reader locks can be held simultaneously on a *ReaderWriterMutex* if it does not have a writer lock on it. A writer lock on a *ReaderWriterMutex* excludes all other threads from holding a lock on the mutex at the same time.

```
class RWM {
    // Implementation specifics
    // ...

    // Represents acquisition of a mutex.
    class scoped_lock {
    public:
        constexpr scoped_lock() noexcept;
        scoped_lock(RWM& m, bool write = true);
        ~scoped_lock();

        scoped_lock(const scoped_lock&) = delete;
        scoped_lock& operator=(const scoped_lock&) = delete;

        void acquire(RWM& m, bool write = true);
        bool try_acquire(RWM& m, bool write = true);
        void release();

        bool upgrade_to_writer();
        bool downgrade_to_reader();
    };
};
```

A type *RWM* satisfies *ReaderWriterMutex* if it meets the following requirements. They form a superset of the *Mutex requirements*.

type `RWM::scoped_lock`

Corresponding scoped-lock type.

`RWM::scoped_lock()`

Constructs `scoped_lock` without acquiring any mutex.

`RWM::scoped_lock(RWM&, bool write = true)`

Constructs `scoped_lock` and acquires a lock on a given mutex. The lock is a writer lock if `write` is true; a reader lock otherwise.

`RWM::~scoped_lock()`

Releases a lock (if acquired).

void `RWM::scoped_lock::acquire(RWM&, bool write = true)`

Acquires a lock on a given mutex. The lock is a writer lock if `write` is true; it is a reader lock, otherwise.

bool `RWM::scoped_lock::try_acquire(RWM&, bool write = true)`

Attempts to acquire a lock on a given mutex. The lock is a writer lock if `write` is true; it is a reader lock, otherwise. Returns `true` if the lock is acquired, `false` otherwise.

RWM::*scoped_lock*::**release**()

Releases a lock. The effect is undefined if no lock is held.

bool RWM::*scoped_lock*::**upgrade_to_writer**()

Changes a reader lock to a writer lock. Returns `false` if lock was released and reacquired. Otherwise, returns `true`, including the case when the lock was already a writer lock.

bool RWM::*scoped_lock*::**downgrade_to_reader**()

Changes a writer lock to a reader lock. Returns `false` if lock was released and reacquired. Otherwise, returns `true`, including the case when the lock was already a reader lock.

Like the *Mutex* requirement, *ReaderWriterMutex* also requires a set of traits to be defined.

static constexpr bool M::**is_rw_mutex**

True if mutex is a reader-writer mutex; false, otherwise.

static constexpr bool M::**is_recursive_mutex**

True if mutex is a recursive mutex; false, otherwise.

static constexpr bool M::**is_fair_mutex**

True if mutex is fair; false, otherwise.

The following table summarizes the library classes that model the *ReaderWriterMutex* requirement and provided guarantees.

Table 2: Provided guarantees for Mutexes that model the ReaderWriterMutex requirement

.	Fair	Reentrant
rw_mutex	No	No
spin_rw_mutex	No	No
speculative_spin_rw_mutex	No	No
queuing_rw_mutex	Yes	No
null_rw_mutex	Yes	Yes

Note: Implementation is allowed to have an opposite guarantees (positive) in case of negative statements from the table above.

Note: For all currently provided reader-writer mutexes,

- `is_recursive_mutex` is `false`
- `scoped_lock::downgrade_to_reader` always returns `true`

However, other implementations of the ReaderWriterMutex requirement are not required to do the same.

See also:

- *rw_mutex*
- *spin_rw_mutex*
- *speculative_spin_rw_mutex*
- *queuing_rw_mutex*
- *null_rw_mutex*

Containers

HashCompare

[req.hash_compare]

HashCompare is an object which is used to calculate hash code for an object and compare two objects for equality.

The type `H` satisfies `HashCompare` if it meets the following requirements:

HashCompare Requirements: Pseudo-Signature, Semantics

H`::H`(const *H*&)
: Copy constructor.

`H::~H()`
: Destructor.

std::size_t `H::hash`(const KeyType &k) const
: Calculates the hash for a provided key.

ReturnType `H::equal`(const KeyType &k1, const KeyType &k2) const
: Requirements:

  - The type `ReturnType` should be implicitly convertible to `bool`.

  Compares `k1` and `k2` for equality.

  If this function returns `true`, `H::hash(k1)` should be equal to `H::hash(k2)`.

ContainerRange

[req.container_range]

`ContainerRange` is a range that represents a concurrent container or a part of the container.

The `ContainerRange` object can be used to traverse the container in parallel algorithms like `parallel_for`.

The type `CR` satisfies the `ContainerRange` requirements if:

- The type `CR` meets the requirements of *Range requirements*.
- The type `CR` provides the following member types and functions:

  type `CR::value_type`
  : The type of the item in the range.

  type `CR::reference`
  : Reference type to the item in the range.

  type `CR::const_reference`
  : Constant reference type to the item in the range.

  type `CR::iterator`
  : Iterator type for range traversal.

  type `CR::size_type`
  : Unsigned integer type for obtaining grain size.

type CR::**difference_type**
: The type of the difference between two iterators.

iterator CR::**begin**()
: Returns an iterator to the beginning of the range.

iterator CR::**end**()
: Returns an iterator to the position that follows the last element in the range.

size_type CR::**grainsize**() const
: Returns the range grain size.

Task scheduler

SuspendFunc

[req.suspend_func]

A type *Func* satisfies *SuspendFunc* if it meets the following requirements:

SuspendFunc Requirements: Pseudo-Signature, Semantics

Func::**Func**(const *Func*&)
: Copy constructor.

void Func::**operator()**(oneapi::tbb::task::suspend_point)
: Body that accepts the current task execution point to resume later.

See also:

- *resumable tasks*

Flow Graph

AsyncNodeBody

[req.async_node_body]

A type *Body* satisfies *AsyncNodeBody* if it meets the following requirements:

AsyncNodeBody Requirements: Pseudo-Signature, Semantics

Body::**Body**(const *Body*&)
: Copy constructor.

Body::**~Body**()
: Destructor.

void Body::**operator()**(const Input &v, GatewayType &gateway)
: **Requirements:**

  - The `Input` type must be the same as the `Input` template type argument of the `async_node` instance in which the `Body` object is passed during construction.

- The `GatewayType` type must be the same as the `gateway_type` member type of the `async_node` instance in which the `Body` object is passed during construction.

The input value `v` is submitted by the flow graph to an external activity. The *gateway interface* allows the external activity to communicate with the enclosing flow graph.

ContinueNodeBody

[req.continue_node_body]

A type *Body* satisfies *ContinueNodeBody* if it meets the following requirements:

ContinueNodeBody Requirements: Pseudo-Signature, Semantics

Body`::Body(`const *Body*`&)`

Copy constructor.

`Body::~Body()`

Destructor.

Output `Body::operator()(`const continue_msg &v`)`

Requirements: The type `Output` must be the same as the template type argument `Output` of the `continue_node` instance in which the `Body` object is passed during construction.

Performs operation and returns a value of type Output.

See also:

- *continue_node class*
- *continue_msg class*

GatewayType

[req.gateway_type]

A type *T* satisfies *GatewayType* if it meets the following requirements:

GatewayType Requirements: Pseudo-Signature, Semantics

bool `T::try_put(`const Output &v`)`

Requirements: The type `Output` must be the same as the template type argument `Output` of the corresponding `async_node` instance.

Broadcasts `v` to all successors of the corresponding `async_node` instance.

void `T::reserve_wait()`

Notifies a flow graph that work has been submitted to an external activity.

void `T::release_wait()`

Notifies a flow graph that work submitted to an external activity has completed.

FunctionNodeBody

[req.function_node_body]

A type *Body* satisfies *FunctionNodeBody* if it meets the following requirements:

FunctionNodeBody Requirements: Pseudo-Signature, Semantics

Body`::Body`(const *Body*&)

Copy constructor.

`Body::~Body()`

Destructor.

Output `Body::operator()`(const Input &v)

Requirements: The `Input` and `Output` types must be the same as the `Input` and `Output` template type arguments of the `function_node` instance in which the `Body` object is passed during construction.

Performs operation on `v` and returns a value of type `Output`.

JoinNodeFunctionObject

[req.join_node_function_object]

A type *Func* satisfies *JoinNodeFunctionObject* if it meets the following requirements:

JoinNodeFunctionObject Requirements: Pseudo-Signature, Semantics

Func`::Func`(const *Func*&)

Copy constructor.

`Func::~Func()`

Destructor.

Key `Func::operator()`(const Input &v)

Requirements: The `Key` and `Input` types must be the same as the `K` and the corresponding element of the `OutputTuple` template arguments of the `join_node` instance to which the `Func` object is passed during construction.

Returns key to be used for hashing input messages.

InputNodeBody

[req.input_node_body]

A type *Body* satisfies *InputNodeBody* if it meets the following requirements:

InputNodeBody Requirements: Pseudo-Signature, Semantics

Body`::Body`(const *Body*&)

Copy constructor.

`Body::~Body()`

Destructor.

Output `Body::operator()`(oneapi::tbb::flow_control &fc)

Requirements: The type `Output` must be the same as the template type argument `Output` of the `input_node` instance in which the `Body` object is passed during construction.

Applies body to generate the next item. Call `fc.stop()` when new element cannot be generated. Because `Output` needs to be returned, `Body` may return any valid value of `Output`, to be immediately discarded.

MultifunctionNodeBody

[req.multifunction_node_body]

A type *Body* satisfies *MultifunctionNodeBody* if it meets the following requirements:

MultifunctionNodeBody Requirements: Pseudo-Signature, Semantics

Body`::Body`(const *Body*&)

Copy constructor.

`Body::~Body()`

Destructor.

void `Body::operator()`(const Input &v, OutputPortsType &p)

Requirements:

- The `Input` type must be the same as the `Input` template type argument of the `multifunction_node` instance in which the `Body` object is passed during construction.
- The `OutputPortsType` type must be the same as the `output_ports_type` member type of the `multifunction_node` instance in which the `Body` object is passed during construction.

Performs operation on `v`. May call `try_put()` on zero or more of the output ports. May call `try_put()` on any output port multiple times.

Sequencer

[req.sequencer]

A type *S* satisfies *Sequencer* if it meets the following requirements:

Sequencer Requirements: Pseudo-Signature, Semantics

S`::S`(const *S*&)

Copy constructor.

`S::~S()`

Destructor.

size_t S::operator()(const T &v)

Requirements: The type T must be the same as the template type argument T of the sequencer_node instance in which the S object is passed during construction.

Returns the sequence number for the provided message v.

See also:

- *sequencer_node class*

8.1.5 Thread Safety

[thread_safety]

Unless otherwise stated, the thread safety rules for the library are as follows:

- Two threads can invoke a method or function concurrently on different objects, but not the same object.
- It is unsafe for two threads to invoke concurrently methods or functions on the same object.

Departures from this convention are noted in the classes descriptions. For example, the concurrent containers are more liberal. By their nature, they do permit some concurrent operations on the same container object.

8.2 oneTBB Interfaces

8.2.1 Configuration

[configuration]

This section describes the most general features of oneAPI Threading Building Blocks (oneTBB) such as namespaces, versioning, and macros.

Namespaces

[configuration.namespaces]

This section describes the oneTBB namespace conventions.

tbb Namespace

The tbb namespace contains public identifiers defined by the library that you can reference in your program.

tbb::flow Namespace

The tbb::flow namespace contains public identifiers defined by the library that you can reference in your program related to the flow graph feature. See *Flow Graph* for more information.

oneapi::tbb Namespace

The `tbb` namespace is a part of the top level `oneapi` namespace. Therefore, all API from the `tbb` namespace (incl. the `tbb::flow` namespace) are available in the `oneapi::tbb` namespace. The `oneapi::tbb` namespace can be considered as an alias for the `tbb` namespace:

```
namespace oneapi { namespace tbb = ::tbb; }
```

Version Information

[configuration.version_information]

oneTBB has macros, an environment variable, and a function that reveal version and runtime information.

```
// Defined in header <oneapi/tbb/version.h>

#define TBB_VERSION_MAJOR   /*implementation-defined*/
#define TBB_VERSION_MINOR   /*implementation-defined*/
#define TBB_VERSION_STRING  /*implementation-defined*/

#define TBB_INTERFACE_VERSION_MAJOR /*implementation-defined*/
#define TBB_INTERFACE_VERSION_MINOR /*implementation-defined*/
#define TBB_INTERFACE_VERSION       /*implementation-defined*/

const char* TBB_runtime_version();
int TBB_runtime_interface_version();
```

Version Macros

oneTBB defines macros related to versioning, as described below.

- `TBB_VERSION_MAJOR` macro defined to integral value that represents major library version.
- `TBB_VERSION_MINOR` macro defined to integral value that represents minor library version.
- `TBB_VERSION_STRING` macro defined to the string representation of the full library version.
- `TBB_INTERFACE_VERSION` macro defined to current interface version. The value is a decimal numeral of the form `xyyz` where `x` is the major interface version number and `y` is the minor interface version number. This macro is increased in each release.
- `TBB_INTERFACE_VERSION_MAJOR` macro defined to `TBB_INTERFACE_VERSION/1000`, which is the major interface version number.
- `TBB_INTERFACE_VERSION_MINOR` macro defined to `TBB_INTERFACE_VERSION%1000/10`, which is the minor interface version number.

TBB_runtime_interface_version Function

Function that returns the interface version of the oneTBB library that was loaded at runtime.

The value returned by `TBB_runtime_interface_version()` may differ from the value of `TBB_INTERFACE_VERSION` obtained at compile time. This can be used to identify whether an application was compiled against a compatible version of the oneTBB headers.

In general, the run-time value `TBB_runtime_interface_version()` must be greater than or equal to the compile-time value of `TBB_INTERFACE_VERSION`. Otherwise, the application may fail to resolve all symbols at run time.

TBB_runtime_version Function

Function that returns the version string of the oneTBB library that was loaded at runtime.

The value returned by `TBB_runtime_version()` may differ from the value of `TBB_VERSION_STRING` obtained at compile time.

TBB_VERSION Environment Variable

Set the environment variable `TBB_VERSION` to 1 to cause the library to print information on `stderr`. Each line is of the form "`TBB: tag value`", where *tag* and *value* provide additional library information below.

> **Caution:** This output is implementation specific and may change at any time.

Enabling Debugging Features

[configuration.debug_features]

The following macros control certain debugging features. In general, it is useful to compile with these features on for development code, and off for production code, because the features may decrease performance. The table below summarizes the macros and their default values. A value of 1 enables the corresponding feature; a value of 0 disables the feature.

Table 3: Debugging Macros

Macro	Default Value	Feature
`TBB_USE_DEBUG`	• Windows* OS: 1 if `_DEBUG` is defined, 0, otherwise. • All other systems: 0.	Default value for all other macros in this table.
`TBB_USE_ASSERT`	`TBB_USE_DEBUG`	Enable internal assertion checking. Can significantly slow down performance.
`TBB_USE_PROFILING_TOOLS`	`TBB_USE_DEBUG`	Enable full support for analysis tools.

TBB_USE_ASSERT Macro

The `TBB_USE_ASSERT` macro controls whether error checking is enabled in the header files. Define `TBB_USE_ASSERT` as 1 to enable error checking.

If an error is detected, the library prints an error message on `stderr` and calls the standard C routine `abort`. To stop a program when internal error checking detects a failure, place a breakpoint on `oneapi::tbb::assertion_failure`.

TBB_USE_PROFILING_TOOLS Macro

The `TBB_USE_PROFILING_TOOLS` macro controls support for Intel® Inspector XE, Intel® VTune™ Amplifier XE and Intel® Advisor.

Define `TBB_USE_PROFILING_TOOLS` as 1 to enable full support for these tools. Leave `TBB_USE_PROFILING_TOOLS` undefined or equal to zero to enable top performance in release builds, at the expense of turning off some support for tools.

Feature Macros

[configuration.feature_macros]

Macros in this section control optional features of the library.

TBB_USE_EXCEPTIONS macro

The `TBB_USE_EXCEPTIONS` macro controls whether the library headers use exception-handling constructs such as `try`, `catch`, and `throw`. The headers do not use these constructs when `TBB_USE_EXCEPTIONS=0`.

For the Microsoft Windows*, Linux*, and macOS* operating systems, the default value is 1 if exception handling constructs are enabled in the compiler, and 0, otherwise.

> **Caution:** The runtime library may still throw an exception when `TBB_USE_EXCEPTIONS=0`.

TBB_USE_GLIBCXX_VERSION macro

The `TBB_USE_GLIBCXX_VERSION` macro can be used to specify the proper version of GNU libstdc++ if the detection fails. Define the value of the macro equal to `Major*10000 + Minor*100 + Patch`, where Major.Minor.Patch is the actual GCC/libstdc++ version (if unknown, it can be obtained with the `'gcc -dumpversion'` command). For example, if you use libstdc++ from GCC 4.9.2, define `TBB_USE_GLIBCXX_VERSION=40902`.

8.2.2 Algorithms

[algorithms]

oneAPI Threading Building Blocks provides a set of generic parallel algorithms.

Parallel Functions

collaborative_call_once

[algorithms.collaborative_call_once]

Function template that executes function exactly once.

```
// Defined in header <oneapi/tbb/collaborative_call_once.h>

namespace oneapi {
    namespace tbb {

        template<typename Func, typename... Args>
        void collaborative_call_once(collaborative_once_flag& flag, Func&& func, Args&&..
↪. args);

    } // namespace tbb
} // namespace oneapi
```

Requirements:

- `Func` type must meet the *Function Objects* requirements described in the [function.objects] section of the ISO C++ standard.

Executes the `Func` object only once, even if it is called concurrently. It allows other threads blocked on the same `collaborative_once_flag` to join oneTBB parallel construction called within the `Func` object.

In case of the exception thrown from the `Func` object, the thread calling the `Func` object receives this exception. One of the threads blocked on the same `collaborative_once_flag` calls the `Func` object again.

collaborative_once_flag Class

collaborative_once_flag

[algorithms.collaborative_call_once.collaborative_once_flag]

Special class that `collaborative_call_once` uses to perform a call only once.

```
// Defined in header <oneapi/tbb/collaborative_call_once.h>

namespace oneapi {
    namespace tbb {

        class collaborative_once_flag {
        public:
            collaborative_once_flag();
            collaborative_once_flag(const collaborative_once_flag&) = delete;
            collaborative_once_flag& operator=(const collaborative_once_flag&) = delete;
        };
    } // namespace tbb
} // namespace oneapi
```

Member functions

`collaborative_once_flag()`

Constructs an `collaborative_once_flag` object. The initial state indicates that no function has been called.

Example

The following example shows a class in which the "Lazy initialization" pattern is implemented on the `cachedProperty` field.

```
#include "oneapi/tbb/collaborative_call_once.h"
#include "oneapi/tbb/parallel_reduce.h"
#include "oneapi/tbb/blocked_range.h"

extern double foo(int i);

class LazyData {
    oneapi::tbb::collaborative_once_flag flag;
    double cachedProperty;
public:
    double getProperty() {
        oneapi::tbb::collaborative_call_once(flag, [&] {
            // serial part
            double result{};

            // parallel part where threads can collaborate
            result = oneapi::tbb::parallel_reduce(oneapi::tbb::blocked_range<int>(0,␣
↪1000), 0.,
                [] (auto r, double val) {
                    for(int i = r.begin(); i != r.end(); ++i) {
                        val += foo(i);
                    }
                    return val;
                },
                std::plus<double>{}
            );

            // continue serial part
            cachedProperty = result;
        });

        return cachedProperty;
    }
};
```

parallel_for

[algorithms.parallel_for]

Function template that performs parallel iteration over a range of values.

```
// Defined in header <oneapi/tbb/parallel_for.h>

namespace oneapi {
    namespace tbb {

        template<typename Index, typename Func>
        void parallel_for(Index first, Index last, const Func& f, /* see-below */␣
↪partitioner, task_group_context& context);
        template<typename Index, typename Func>
        void parallel_for(Index first, Index last, const Func& f, task_group_context&␣
↪context);
        template<typename Index, typename Func>
        void parallel_for(Index first, Index last, const Func& f, /* see-below */␣
↪partitioner);
        template<typename Index, typename Func>
        void parallel_for(Index first, Index last, const Func& f);

        template<typename Index, typename Func>
        void parallel_for(Index first, Index last, Index step, const Func& f, /* see-
↪below */ partitioner, task_group_context& context);
        template<typename Index, typename Func>
        void parallel_for(Index first, Index last, Index step, const Func& f, task_group_
↪context& context);
        template<typename Index, typename Func>
        void parallel_for(Index first, Index last, Index step, const Func& f, /* see-
↪below */ partitioner);
        template<typename Index, typename Func>
        void parallel_for(Index first, Index last, Index step, const Func& f);

        template<typename Range, typename Body>
        void parallel_for(const Range& range, const Body& body, /* see-below */␣
↪partitioner, task_group_context& context);
        template<typename Range, typename Body>
        void parallel_for(const Range& range, const Body& body, task_group_context&␣
↪context);
        template<typename Range, typename Body>
        void parallel_for(const Range& range, const Body& body, /* see-below */␣
↪partitioner);
        template<typename Range, typename Body>
        void parallel_for(const Range& range, const Body& body);

    } // namespace tbb
} // namespace oneapi
```

A `partitioner` type may be one of the following entities:

- `const auto_partitioner&`
- `const simple_partitioner&`

- `const static_partitioner&`
- `affinity_partitioner&`

Requirements:

- The `Range` type must meet the *Range requirements*.
- The `Body` type must meet the *ParallelForBody requirements*.
- The `Index` type must meet the *ParallelForIndex requirements*.
- The `Func` type must meet the *ParallelForFunc requirements*.

The `oneapi::tbb::parallel_for(first, last, step, f)` overload represents parallel execution of the loop:

```
for (auto i = first; i < last; i += step) f(i);
```

The loop must not wrap around. The step value must be positive. If omitted, it is implicitly 1. There is no guarantee that the iterations run in parallel. A deadlock may occur if a lesser iteration waits for a greater iteration. The partitioning strategy is `auto_partitioner` when the parameter is not specified.

The `parallel_for(range,body,partitioner)` overload provides a more general form of parallel iteration. It represents parallel execution of `body` over each value in `range`. The optional `partitioner` parameter specifies a partitioning strategy.

`parallel_for` recursively splits the range into subranges to the point such that `is_divisible()` is false for each subrange, and makes copies of the body for each of these subranges. For each such body/subrange pair, it invokes `Body::operator()`.

Some of the copies of the range and body may be destroyed after `parallel_for` returns. This late destruction is not an issue in typical usage, but is something to be aware of when looking at execution traces or writing range or body objects with complex side effects.

`parallel_for` may execute iterations in non-deterministic order. Do not rely on any particular execution order for correctness. However, for efficiency, do expect `parallel_for` to tend towards operating on consecutive runs of values.

In case of serial execution, `parallel_for` performs iterations from left to right in the following sense.

All overloads can accept a *task_group_context* object so that the algorithm's tasks are executed in this context. By default, the algorithm is executed in a bound context of its own.

Complexity

If the range and body take *O(1)* space, and the range splits into nearly equal pieces, the space complexity is *O(P log(N))*, where *N* is the size of the range and *P* is the number of threads.

See also:

- *Partitioners*

parallel_reduce

[algorithms.parallel_reduce]

Function template that computes reduction over a range.

```
// Defined in header <oneapi/tbb/parallel_reduce.h>

namespace oneapi {
    namespace tbb {
```

(continues on next page)

(continued from previous page)

```
        template<typename Range, typename Value, typename Func, typename Reduction>
        Value parallel_reduce(const Range& range, const Value& identity, const Func&␣
↪func, const Reduction& reduction, /* see-below */ partitioner, task_group_context&␣
↪context);
        template<typename Range, typename Value, typename Func, typename Reduction>
        Value parallel_reduce(const Range& range, const Value& identity, const Func&␣
↪func, const Reduction& reduction, /* see-below */ partitioner);
        template<typename Range, typename Value, typename Func, typename Reduction>
        Value parallel_reduce(const Range& range, const Value& identity, const Func&␣
↪func, const Reduction& reduction, task_group_context& context);
        template<typename Range, typename Value, typename Func, typename Reduction>
        Value parallel_reduce(const Range& range, const Value& identity, const Func&␣
↪func, const Reduction& reduction);

        template<typename Range, typename Body>
        void parallel_reduce(const Range& range, Body& body, /* see-below */ partitioner,
↪ task_group_context& context);
        template<typename Range, typename Body>
        void parallel_reduce(const Range& range, Body& body, /* see-below */␣
↪partitioner);
        template<typename Range, typename Body>
        void parallel_reduce(const Range& range, Body& body, task_group_context&␣
↪context);
        template<typename Range, typename Body>
        void parallel_reduce(const Range& range, Body& body);

    } // namespace tbb
} // namespace oneapi
```

A `partitioner` type may be one of the following entities:

- `const auto_partitioner&`
- `const simple_partitioner&`
- `const static_partitioner&`
- `affinity_partitioner&`

Requirements:

- The `Range` type must meet the *Range requirements*.
- The `Body` type must meet the *ParallelReduceBody requirements*.
- The `Value` type must meet the *CopyConstructible* requirements from the [copyconstructible] section and *CopyAssignable* requirements from the [copyassignable] section of the ISO C++ Standard.
- The `Func` type must meet the *ParallelReduceFunc requirements*. Since C++17, `Func` may also be a pointer to a const member function in `Range` that takes `const Value&` argument and returns `Value`.
- The `Reduction` types must meet *ParallelReduceReduction requirements*. Since C++17, `Reduction` may also be a pointer to a const member function in `Value` that takes `const Value&` argument and returns `Value`.

The function template `parallel_reduce` has two forms: The functional form is designed to be easy to use in conjunction with lambda expressions. The imperative form is designed to minimize copying of data.

The functional form `parallel_reduce(range, identity, func, reduction)` performs a parallel reduction by applying *func* to subranges in *range* and reducing the results with the binary operator *reduction*. It returns the result of

the reduction. The *identity* parameter specifies the left identity element for *func*'s `operator()`. Parameters *func* and *reduction* can be lambda expressions.

The imperative form `parallel_reduce(range,body)` performs parallel reduction of *body* over each value in *range*.

A `parallel_reduce` recursively splits the range into subranges to the point such that `is_divisible()` is false for each subrange. A `parallel_reduce` uses the splitting constructor to make one or more copies of the body for each thread. It may copy a body while the body's `operator()` or method `join` runs concurrently. You are responsible for ensuring the safety of such concurrency. In typical usage, the safety requires no extra effort.

`parallel_reduce` may invoke the splitting constructor for the body. For each such split of the body, it invokes the `join` method to merge the results from the bodies. Define `join` to update this to represent the accumulated result for this and rhs. The reduction operation should be associative, but does not have to be commutative. For a noncommutative operation *op*, `left.join(right)` should update *left* to be the result of *left op right*.

A body is split only if the range is split, but the converse is not necessarily to be so. The user must neither rely on a particular choice of body splitting nor on the subranges processed by a given body object being consecutive. `parallel_reduce` makes the choice of body splitting nondeterministically.

When executed serially `parallel_reduce` run sequentially from left to right in the same sense as for `parallel_for`. Sequential execution never invokes the splitting constructor or method join.

All overloads can accept a *task_group_context* object so that the algorithm's tasks are executed in this context. By default, the algorithm is executed in a bound context of its own.

Complexity

If the range and body take *O(1)* space, and the range splits into nearly equal pieces, the space complexity is $O(P \times log(N))$, where N is the size of the range and P is the number of threads.

Example (Imperative Form)

The following code sums the values in an array.

```
#include "oneapi/tbb/parallel_reduce.h"
#include "oneapi/tbb/blocked_range.h"

using namespace oneapi::tbb;

struct Sum {
    float value;
    Sum() : value(0) {}
    Sum( Sum& s, split ) {value = 0;}
    void operator()( const blocked_range<float*>& r ) {
        float temp = value;
        for( float* a=r.begin(); a!=r.end(); ++a ) {
            temp += *a;
        }
        value = temp;
    }
    void join( Sum& rhs ) {value += rhs.value;}
};

float ParallelSum( float array[], size_t n ) {
    Sum total;
    parallel_reduce( blocked_range<float*>( array, array+n ), total );
```

(continues on next page)

(continued from previous page)

```
    return total.value;
}
```

The example generalizes to reduction for any associative operation *op* as follows:

- Replace occurrences of 0 with the identity element for *op*
- Replace occurrences of += with *op*= or its logical equivalent.
- Change the name `Sum` to something more appropriate for *op*.

The operation may be noncommutative. For example, *op* could be matrix multiplication.

Example with Lambda Expressions

The following is similar to the previous example, but written using lambda expressions and the functional form of `parallel_reduce`.

```
#include "oneapi/tbb/parallel_reduce.h"
#include "oneapi/tbb/blocked_range.h"

using namespace oneapi::tbb;

float ParallelSum( float array[], size_t n ) {
    return parallel_reduce(
        blocked_range<float*>( array, array+n ),
        0.f,
        [](const blocked_range<float*>& r, float init)->float {
            for( float* a=r.begin(); a!=r.end(); ++a )
                init += *a;
            return init;
        },
        []( float x, float y )->float {
            return x+y;
        }
    );
}
```

See also:

- *Partitioners*

parallel_deterministic_reduce

[algorithms.parallel_deterministic_reduce]

Function template that computes reduction over a range, with deterministic split/join behavior.

```
// Defined in header <oneapi/tbb/parallel_reduce.h>

namespace oneapi {
    namespace tbb {
```

(continues on next page)

(continued from previous page)

```
        template<typename Range, typename Value, typename Func, typename Reduction>
        Value parallel_deterministic_reduce( const Range& range, const Value& identity,␣
↪const Func& func, const Reduction& reduction, /* see-below */ partitioner, task_group_
↪context& context);
        template<typename Range, typename Value, typename Func, typename Reduction>
        Value parallel_deterministic_reduce( const Range& range, const Value& identity,␣
↪const Func& func, const Reduction& reduction, /* see-below */ partitioner);
        template<typename Range, typename Value, typename Func, typename Reduction>
        Value parallel_deterministic_reduce( const Range& range, const Value& identity,␣
↪const Func& func, const Reduction& reduction, task_group_context& context);
        template<typename Range, typename Value, typename Func, typename Reduction>
        Value parallel_deterministic_reduce( const Range& range, const Value& identity,␣
↪const Func& func, const Reduction& reduction);

        template<typename Range, typename Body>
        void parallel_deterministic_reduce( const Range& range, Body& body, /* see-below␣
↪*/ partitioner, task_group_context& context);
        template<typename Range, typename Body>
        void parallel_deterministic_reduce( const Range& range, Body& body, /* see-below␣
↪*/ partitioner);
        template<typename Range, typename Body>
        void parallel_deterministic_reduce( const Range& range, Body& body, task_group_
↪context& context);
        template<typename Range, typename Body>
        void parallel_deterministic_reduce( const Range& range, Body& body);

    } // namespace tbb
} // namespace oneapi
```

A `partitioner` type may be one of the following entities:

- `const simple_partitioner&`
- `const static_partitioner&`

The function template `parallel_deterministic_reduce` is very similar to the `parallel_reduce` template. It also has the functional and imperative forms and has *similar requirements*.

Unlike `parallel_reduce`, `parallel_deterministic_reduce` has deterministic behavior with regard to splits of both `Body` and `Range` and joins of the bodies. For the functional form, `Func` is applied to a deterministic set of `Ranges`, and `Reduction` merges partial results in a deterministic order. To achieve that, `parallel_deterministic_reduce` uses a `simple_partitioner` or a `static_partitioner` only because other partitioners react to random work stealing behavior.

Caution: Since `simple_partitioner` does not automatically coarsen ranges, make sure to specify an appropriate grain size. See *Partitioners section* for more information.

`parallel_deterministic_reduce` always invokes the `Body` splitting constructor for each range split.

As a result, `parallel_deterministic_reduce` recursively splits a range until it is no longer divisible, and creates a new body (by calling the `Body` splitting constructor) for each new subrange. Like `parallel_reduce`, for each body split the method `join` is invoked in order to merge the results from the bodies.

Therefore, for given arguments, `parallel_deterministic_reduce` executes the same set of split and join operations

no matter how many threads participate in execution and how tasks are mapped to the threads. If the user-provided functions are also deterministic (that is, different runs with the same input result in the same output), multiple calls to `parallel_deterministic_reduce` produce the same result. Note however that the result might differ from that obtained with an equivalent sequential (linear) algorithm.

Complexity

If the range and body take *O(1)* space, and the range splits into nearly equal pieces, the space complexity is *O(P log(N))*, where *N* is the size of the range and *P* is the number of threads.

See also:

- *parallel_reduce*
- *Partitioners*

parallel_scan

[algorithms.parallel_scan]

Function template that computes a parallel prefix.

```
// Defined in header <oneapi/tbb/parallel_scan.h>

template<typename Range, typename Body>
void parallel_scan( const Range& range, Body& body );
template<typename Range, typename Body>
void parallel_scan( const Range& range, Body& body, /* see-below */ partitioner );

template<typename Range, typename Value, typename Scan, typename Combine>
Value parallel_scan( const Range& range, const Value& identity, const Scan& scan, const␣
↪Combine& combine );
template<typename Range, typename Value, typename Scan, typename Combine>
Value parallel_scan( const Range& range, const Value& identity, const Scan& scan, const␣
↪Combine& combine, /* see-below */ partitioner );
```

A `partitioner` type may be one of the following entities:

- `const auto_partitioner&`
- `const simple_partitioner&`

Requirements:

- The `Range` type must meet the *Range requirement*.
- The `Body` type must meet the *ParallelScanBody requirements*.
- The `Value` type must meet the *CopyConstructible* requirements from the [copyconstructible] section and *CopyAssignable* requirements from the [copyassignable] section of the ISO C++ Standard.
- The `Scan` type must meet the *ParallelScanFunc requirements*. Since C++17, `Scan` may also be a pointer to a const member function in `Range` that takes `const Value&` and `bool` arguments and returns `Value`.
- The `Combine` type must meet the *ParallelScanCombine requirements*. Since C++17, `Combine` may also be a pointer to a const member function in `Value` that takes `const Value&` argument and returns `Value`.

The function template `parallel_scan` computes a parallel prefix, also known as a parallel scan. This computation is an advanced concept in parallel computing that is sometimes useful in scenarios that appear to have inherently serial dependences.

A mathematical definition of the parallel prefix is as follows. Let × be an associative operation with left-identity element $id_{\times}$. The parallel prefix of × over a sequence $z_0, z_1, \ldots$ *z*$_{n-1}$ is a sequence $y_0, y_1, y_2, \ldots$ *y*$_{n-1}$ where:

- $y_0 = id_{\times} \times z_0$
- $y_i = y_{i-1} \times z_i$

For example, if × is addition, the parallel prefix corresponds to a running sum. A serial implementation of a parallel prefix is:

```
T temp = id;
for( int i=1; i<=n; ++i ) {
   temp = temp + z[i];
   y[i] = temp;
}
```

Parallel prefix performs this in parallel by reassociating the application of × (+ in example) and using two passes. It may invoke × up to twice as many times as the serial prefix algorithm. Even though it does more work, given the right grain size the parallel algorithm can outperform the serial one because it distributes the work across multiple hardware threads.

The function template `parallel_scan` has two forms. The imperative form `parallel_scan(range, body)` implements parallel prefix generically.

A summary (refer to *ParallelScanBody requirements*) contains enough information such that for two consecutive subranges *r* and *s*:

- If *r* has no preceding subrange, the scan result for *s* can be computed from knowing *s* and the summary for *r*.
- A summary of *r* concatenated with *s* can be computed from the summaries of *r* and *s*.

The functional form `parallel_scan(range, identity, scan, combine)` is designed to use with functors and lambda expressions, hiding some complexities of the imperative form. It uses the same *scan* functor in both passes, differentiating them via a boolean parameter, combines summaries with *combine* functor, and returns the summary computed over the whole *range*. The *identity* argument is the left identity element for `Scan::operator()`.

pre_scan and final_scan Classes

pre_scan_tag and final_scan_tag

[algorithms.parallel_scan.scan_tags]

Types that distinguish the phases of `parallel_scan`.

Types `pre_scan_tag` and `final_scan_tag` are dummy types used in conjunction with `parallel_scan`. See the example in the *parallel_scan* section for demonstration of how they are used in the signature of `operator()`.

```
// Defined in header <oneapi/tbb/parallel_scan.h>

namespace oneapi {
    namespace tbb {

        struct pre_scan_tag {
            static bool is_final_scan();
            operator bool();
        };
```

(continues on next page)

(continued from previous page)

```
        struct final_scan_tag {
            static bool is_final_scan();
            operator bool();
        };

    } // namespace tbb
} // namespace oneapi
```

Member functions

bool **is_final_scan**()

`true` for a `final_scan_tag`, `false`, otherwise.

operator bool()

`true` for a `final_scan_tag`, `false`, otherwise.

The `parallel_scan` template makes an effort to avoid prescanning where possible. When executed serially, `parallel_scan` processes the subranges without any pre-scans by processing the subranges from left to right using final scans. That is why final scans must compute a summary as well as the final scan result. The summary might be needed to process the next subrange if no other thread has pre-scanned it yet.

Example (Imperative Form)

The following code demonstrates how `Body` could be implemented for `parallel_scan` to compute the same result as in the earlier sequential example.

```
class Body {
    T sum;
    T* const y;
    const T* const z;
public:
    Body( T y_[], const T z_[] ) : sum(id), z(z_), y(y_) {}
    T get_sum() const { return sum; }

    template<typename Tag>
    void operator()( const oneapi::tbb::blocked_range<int>& r, Tag ) {
        T temp = sum;
        for( int i=r.begin(); i<r.end(); ++i ) {
            temp = temp + z[i];
            if( Tag::is_final_scan() )
                y[i] = temp;
        }
        sum = temp;
    }
    Body( Body& b, oneapi::tbb::split ) : z(b.z), y(b.y), sum(id) {}
    void reverse_join( Body& a ) { sum = a.sum + sum; }
    void assign( Body& b ) { sum = b.sum; }
};

T DoParallelScan( T y[], const T z[], int n ) {
```

(continues on next page)

(continued from previous page)

```
    Body body(y,z);
    oneapi::tbb::parallel_scan( oneapi::tbb::blocked_range<int>(0,n), body );
    return body.get_sum();
}
```

The definition of `operator()` demonstrates typical patterns when using `parallel_scan`.

- A single template defines both versions. Doing so is not required, but usually saves coding effort, because two versions are usually similar. The library defines the static method `is_final_scan` to enable differentiation between the versions.
- The prescan variant computes the `×` reduction, but does not update `y`. The prescan is used by `parallel_scan` to generate look-ahead partial reductions.
- The final scan variant computes the `×` reduction and updates `y`.

The `reverse_join` operation is similar to the `join` operation used by `parallel_reduce`, except that the arguments are reversed. That is, `this` is the *right* argument of `×`. The template function `parallel_scan` decides if and when to generate parallel work. Thus, it is crucial that `×` is associative and that the methods of `Body` faithfully represent it. Operations such as floating-point addition, which are somewhat associative, can be used with the understanding that the results may be rounded differently depending on the association used by `parallel_scan`. The reassociation may differ between runs even on the same machine. However, when executed serially, `parallel_scan` associates identically to the serial form shown at the beginning of this section.

If you change the example to use a `simple_partitioner`, be sure to provide a grain size. The code below shows how to do this for the grain size of 1000:

```
parallel_scan( blocked_range<int>(0,n,1000), total, simple_partitioner() );
```

Example with Lambda Expressions

The following is analogous to the previous example, but written using lambda expressions and the functional form of `parallel_scan`:

```
T DoParallelScan( T y[], const T z[], int n ) {
    return oneapi::tbb::parallel_scan(
        oneapi::tbb::blocked_range<int>(0,n),
        id,
        [](const oneapi::tbb::blocked_range<int>& r, T sum, bool is_final_scan)->T {
            T temp = sum;
            for( int i=r.begin(); i<r.end(); ++i ) {
                temp = temp + z[i];
                if( is_final_scan )
                    y[i] = temp;
            }
            return temp;
        },
        []( T left, T right ) {
            return left + right;
        }
    );
}
```

See also:

- *blocked_range class*
- *parallel_reduce algorithm*

parallel_for_each

[algorithms.parallel_for_each]

Function template that processes work items in parallel.

```
// Defined in header <oneapi/tbb/parallel_for_each.h>

namespace oneapi {
    namespace tbb {

        template<typename InputIterator, typename Body>
        void parallel_for_each( InputIterator first, InputIterator last, Body body );
        template<typename InputIterator, typename Body>
        void parallel_for_each( InputIterator first, InputIterator last, Body body, task_
↪group_context& context );

        template<typename Container, typename Body>
        void parallel_for_each( Container& c, Body body );
        template<typename Container, typename Body>
        void parallel_for_each( Container& c, Body body, task_group_context& context );

        template<typename Container, typename Body>
        void parallel_for_each( const Container& c, Body body );
        template<typename Container, typename Body>
        void parallel_for_each( const Container& c, Body body, task_group_context&␣
↪context );

    } // namespace tbb
} // namespace oneapi
```

Requirements:

- The `Body` type must meet the *ParallelForEachBody requirements*. Since C++17, `Body` may also be a pointer to a member function in `Index`.
- The `InputIterator` type must meet the *Input Iterator* requirements from the [input.iterators] section of the ISO C++ Standard.
- If `InputIterator` type does not meet the *Forward Iterator* requirements from the [forward.iterators] section of the ISO C++ Standard, the `std::iterator_traits<InputIterator>::value_type` type must be constructible from `std::iterator_traits<InputIterator>::reference`.
- The `Container` type must meet the *ContainerBasedSequence requirements*.
- The type returned by `Container::begin()` must meet the same requirements as the `InputIterator` type above.

The `parallel_for_each` template has two forms.

The sequence form `parallel_for_each(first, last, body)` applies a function object `body` over a sequence `[first,last)`. Items may be processed in parallel.

The container form `parallel_for_each(c, body)` is equivalent to `parallel_for_each(std::begin(c), std::end(c), body)`.

All overloads can accept a *task_group_context* object so that the algorithm's tasks are executed in this context. By default, the algorithm is executed in a bound context of its own.

feeder Class

Additional work items can be added by `body` if it has a second argument of type `feeder`. The function terminates when `body(x)` returns for all items `x` that were in the input sequence or added by method `feeder::add`.

feeder

[algorithms.parallel_for_each.feeder]

Inlet into which additional work items for a `parallel_for_each` can be fed.

```
// Defined in header <oneapi/tbb/parallel_for_each.h>

namespace oneapi {
    namespace tbb {

        template<typename Item>
        class feeder {
        public:
            void add( const Item& item );
            void add( Item&& item );
        };

    } // namespace tbb
} //namespace oneapi
```

Member functions

void **add**(const Item &item)

Adds item to a collection of work items to be processed.

Requirements: The `Item` type must meet the *CopyConstructible* requirements from the [copyconstructible] section of the ISO C++ Standard.

void **add**(Item &&item)

Same as the above but uses the move constructor of `Item`, if available.

Requirements: The `Item` type must meet the *MoveConstructible* requirements from the [moveconstructible] section of the ISO C++ Standard.

Caution: Must be called from a `Body::operator()` created by the `parallel_for_each` function. Otherwise, the termination semantics of method `operator()` are undefined.

Example

The following code sketches a body with the two-argument form of `operator()`.

```
struct MyBody {
    void operator()(item_t item, parallel_do_feeder<item_t>& feeder ) {
        for each new piece of work implied by item do {
            item_t new_item = initializer;
            feeder.add(new_item);
        }
    }
};
```

parallel_invoke

[algortihms.parallel_invoke]

Function template that evaluates several functions in parallel.

```
// Defined in header <oneapi/tbb/parallel_invoke.h>

namespace oneapi {
    namespace tbb {

        template<typename... Functions>
        void parallel_invoke(Functions&&... fs);

    } // namespace tbb
} // namespace oneapi
```

Requirements:

- All members of `Functions` parameter pack must meet *Function Objects* requirements described in the [function.objects] section of the ISO C++ standard.
- Last member of `Functions` parameter pack may be a `task_group_context&` type.

Evaluates each member passed to `parallel_invoke` possibly in parallel. Return values are ignored.

The algorithm can accept a *task_group_context* object so that the algorithm's tasks are executed in this context. By default, the algorithm is executed in a bound context of its own.

Example

The following example evaluates `f()`, `g()`, `h()`, and `bar(1)` in parallel.

```
#include "oneapi/tbb/parallel_invoke.h"

extern void f();
extern void bar(int);

class MyFunctor {
    int arg;
```

(continues on next page)

(continued from previous page)

```
public:
    MyFunctor(int a) : arg(a) {}
    void operator()() const { bar(arg); }
};

void RunFunctionsInParallel() {
    MyFunctor g(2);
    MyFunctor h(3);

    oneapi::tbb::parallel_invoke(f, g, h, []{bar(1);});
}
```

parallel_pipeline

[algorithms.parallel_pipeline]

Strongly-typed interface for pipelined execution.

```
// Defined in header <oneapi/tbb/parallel_pipeline.h>

namespace oneapi {
    namespace tbb {

        void parallel_pipeline( size_t max_number_of_live_tokens, const filter<void,void>
↪& filter_chain );
        void parallel_pipeline( size_t max_number_of_live_tokens, const filter<void,void>
↪& filter_chain, task_group_context& context );

    } // namespace tbb
} // namespace oneapi
```

A `parallel_pipeline` algorithm represents pipelined application of a series of filters to a stream of items. Each filter operates in a particular mode: parallel, serial in-order, or serial out-of-order.

To build and run a pipeline from functors g_0, g_1, g_2, ..., g_n, write:

```
parallel_pipeline( max_number_of_live_tokens,
                   make_filter<void,I1>(mode0,g0) &
                   make_filter<I1,I2>(mode1,g1) &
                   make_filter<I2,I3>(mode2,g2) &
                   ...
                   make_filter<In,void>(moden,gn) );
```

In general, the g_i functor should define its `operator()` to map objects of type I_i to objects of type I_{i+1}. Functor g_0 is a special case, because it notifies the pipeline when the end of an input stream is reached. Functor g_0 must be defined such that for a *flow_control* object *fc*, the expression g_0 (*fc*) either returns the next value in the input stream, or invokes *fc.stop()* if the end of the input stream is reached and returns a dummy value.

Each *filter* should be specified by two template arguments. These arguments define filters input and output types. The first and last filters are special cases. Input type of the first filter must be *void*, output type of the last filter must be *void* too.

Before passing to *parallel_pipeline*, concatenate all filters to one(`filter<void, void>`) with *filter::operator&()*. The operator requires that the second template argument of its left operand matches the first template argument of its

second operand.

The number of items processed in parallel depends on the structure of the pipeline and number of available threads. *max_number_of_live_tokens* sets the threshold for concurrently processed items.

If the *context* argument is specified, pipeline's tasks are executed in this context. By default, the algorithm is executed in a bound context of its own.

Example

The following example uses `parallel_pipeline` to compute the root-mean-square of a sequence defined by [*first* , *last*).

```
float RootMeanSquare( float* first, float* last ) {
    float sum=0;
    parallel_pipeline( /*max_number_of_live_token=*/16,
        make_filter<void,float*>(
            filter_mode::serial_in_order,
            [&](flow_control& fc)-> float*{
                if( first<last ) {
                    return first++;
                } else {
                    fc.stop();
                    return nullptr;
                }
            }
        ) &
        make_filter<float*,float>(
            filter_mode::parallel,
            [](float* p){return (*p)*(*p);}
        ) &
        make_filter<float,void>(
            filter_mode::serial_in_order,
            [&](float x) {sum+=x;}
        )
    );
    return sqrt(sum);
}
```

filter Class Template

filter

[algorithms.parallel_pipeline.filter]

A `filter` class template represents a strongly-typed filter in a `parallel_pipeline` algorithm, with its template parameters specifying the filter input and output types. A `filter` can be constructed from a functor or by composing two `filter` objects with `operator&()`. The same `filter` object can be reused in multiple `&` expressions.

The `filter` class should only be used in conjunction with `parallel_pipeline` functions.

```
// Defined in header <oneapi/tbb/parallel_pipeline.h>
```

(continues on next page)

(continued from previous page)

```
namespace oneapi {
    namespace tbb {

        template<typename InputType, typename OutputType>
        class filter {
        public:
            filter() = default;
            filter( const filter& rhs ) = default;
            filter( filter&& rhs ) = default;
            void operator=(const filter& rhs) = default;
            void operator=( filter&& rhs ) = default;

            template<typename Body>
            filter( filter_mode mode, const Body& body );

            filter& operator&=( const filter<OutputType,OutputType>& right );

            void clear();
        }

        template<typename T, typename U, typename Body>
        filter<T,U> make_filter( filter::mode mode, const Body& f );
        template<typename T, typename V, typename U>
        filter<T,U> operator&( const filter<T,V>& left, const filter<V,U>& right );

    } // namespace tbb
} // namespace oneapi
```

Requirements:

- If *InputType* is `void`, a `Body` type must meet the *FirstFilterBody requirements*.
- If *OutputType* is `void`, a `Body` type must meet the *LastFilterBody requirements*. Since C++17, `Body` may also be a pointer to a member function in `InputType`.
- If *InputType* and *OutputType* are not `void`, a `Body` type must meet the *MiddleFilterBody requirements*. Since C++17, `Body` may also be a pointer to a member function in `InputType` that returns `OutputType` or a pointer to a data member in `InputType` of type `OutputType`.
- If *InputType* and *OutputType* are `void`, a `Body` type must meet the *SingleFilterBody requirements*.

filter_mode Enumeration

filter_mode

[algorithms.parallel_pipeline.filter_mode]

A `filter_mode` enumeration represents an execution mode of a `filter` in a `parallel_pipeline` algorithm.

Its enumerated values and their meanings are as follows:

- A `parallel` filter can process multiple items in parallel and without a particular order.
- A `serial_out_of_order` filter processes items one at a time and without a particular order.

- A `serial_in_order` filter processes items one at a time. The order in which items are processed is implicitly set by the first *serial_in_order* filter and respected by all other such filters in the pipeline.

```
// Defined in header <oneapi/tbb/parallel_pipeline.h>

namespace oneapi {
    namespace tbb {

        enum class filter_mode {
            parallel = /*implementation-defined*/,
            serial_in_order = /*implementation-defined*/,
            serial_out_of_order = /*implementation-defined*/
        };

    } // namespace tbb
} // namespace oneapi
```

Member functions

filter()

Constructs an undefined filter.

> **Caution:** The effect of using an undefined filter by `operator&()` or `parallel_pipeline` is undefined.

template<typename **Body**>
filter(filter_mode mode, const *Body* &body)

Constructs a `filter` that uses a copy of a provided `body` to map an input value of type *InputType* to an output value of type *OutputType*, and that operates in the specified `mode`.

void **clear**()

Sets `*this` to an undefined filter.

Non-member functions

template<typename **T**, typename **U**, typename **Func**>
filter<*T*, *U*> **make_filter**(*filter*::mode mode, const *Func* &f)

Returns `filter<T, U>(mode, f)`.

template<typename **T**, typename **V**, typename **U**>
filter<*T*, *U*> **operator&**(const *filter*<*T*, *V*> &left, const *filter*<*V*, *U*> &right)

Returns a `filter` representing the composition of filters *left* and *right*. The composition behaves as if the output value of *left* becomes the input value of *right*.

Deduction Guides

```
template<typename Body>
filter(filter_mode, Body) -> filter<filter_input<Body>, filter_output<Body>>;
```

Where:

- `filter_input<Body>` is an alias to the `Body::operator()` input parameter type. If `Body::operator()` input parameter type is `flow_control` then `filter_input<Body>` is `void`.
- `filter_output<Body>` is an alias to the `Body::operator()` return type.

flow_control Class

flow_control

[algorithms.parallel_pipeline.flow_control]

Enables the first filter in a composite filter to indicate when the end of input stream is reached.

Template function `parallel_pipeline` passes a `flow_control` object to the functor of the first filter. When the functor reaches the end of its input stream, it should invoke `fc.stop()` and return a dummy value that will not be passed to the next filter.

```
// Defined in header <oneapi/tbb/parallel_pipeline.h>

namespace oneapi {
    namespace tbb {

        class flow_control {
        public:
            void stop();
        };

    } // namespace tbb
} namespace oneapi
```

Member functions

void **stop**()

Indicates that first filter of the pipeline reaches the end of its output.

See also:

- *FilterBody requiremnts*
- *filter class*

See also:

- *task_group_context*

parallel_sort

[algorithms.parallel_sort]

Function template that sorts a sequence.

```
// Defined in header <oneapi/tbb/parallel_sort.h>

namespace oneapi {
    namespace tbb {

        template<typename RandomAccessIterator>
        void parallel_sort( RandomAccessIterator begin, RandomAccessIterator end );
        template<typename RandomAccessIterator, typename Compare>
        void parallel_sort( RandomAccessIterator begin, RandomAccessIterator end, const␣
↪Compare& comp );

        template<typename Container>
        void parallel_sort( Container&& c );
        template<typename Container>
        void parallel_sort( Container&& c, const Compare& comp );

    } //  namespace tbb
} // namespace oneapi
```

Requirements:

- The `RandomAccessIterator` type must meet the *Random Access Iterators* requirements from [random.access.iterators] and *ValueSwappable* requirements from the [swappable.requirements] ISO C++ Standard section.
- The `Compare` type must meet the *Compare* type requirements from the [alg.sorting] ISO C++ Standard section.
- The `Container` type must meet the *ContainerBasedSequence requirements* which iterators must meet the *Random Access Iterators* requirements from [random.access.iterators] and *Swappable* requirements from the [swappable.requirements] ISO C++ Standard section.
- The type of dereferenced `RandomAccessIterator` or dereferenced `Container` iterator must meet the MoveAssignable requirements from [moveassignable] section of ISO C++ Standard and the MoveConstructible requirements from [moveconstructible] section of ISO C++ Standard.

Sorts a sequence or a container. The sort is neither stable nor deterministic: relative ordering of elements with equal keys is not preserved and not guaranteed to repeat if the same sequence is sorted again.

A call `parallel_sort( begin, end, comp )` sorts the sequence *[begin, end)* using the argument `comp` to determine relative orderings. If `comp( x, y )` returns `true`, *x* appears before *y* in the sorted sequence.

A call `parallel_sort( begin, end )` is equivalent to `parallel_sort( begin, end, comp )`, where *comp* uses *operator<* to determine relative orderings.

A call `parallel_sort( c, comp )` is equivalent to `parallel_sort( std::begin(c), std::end(c), comp )`.

A call `parallel_sort( c )` is equivalent to `parallel_sort( c, comp )`, where *comp* uses *operator<* to determine relative orderings.

Complexity

`parallel_sort` is a comparison sort with an average time complexity of $O(N \times log(N))$, where N is the number of elements in the sequence. `parallel_sort` may be executed concurrently to improve execution time.

Blocked Ranges

Types that meet the *Range requirements*.

blocked_range

[algorithms.blocked_range]

Class template for a recursively divisible half-open interval.

A `blocked_range` represents a half-open range [*i*,*j*) that can be recursively split.

A `blocked_range` meets the *Range requirements*.

A `blocked_range` specifies a *grain size* of type `size_t`.

A `blocked_range` is splittable into two subranges if the size of the range exceeds its grain size. The ideal grain size depends on the context of the `blocked_range`, which is typically passed as the range argument to the loop templates `parallel_for`, `parallel_reduce`, or `parallel_scan`.

```
// Defined in header <oneapi/tbb/blocked_range.h>

namespace oneapi {
    namespace tbb {

        template<typename Value>
        class blocked_range {
        public:
            // types
            using size_type = size_t;
            using const_iterator = Value;

            // constructors
            blocked_range( Value begin, Value end, size_type grainsize=1 );
            blocked_range( blocked_range& r, split );
            blocked_range( blocked_range& r, proportional_split& proportion );

            // capacity
            size_type size() const;
            bool empty() const;

            // access
            size_type grainsize() const;
            bool is_divisible() const;

            // iterators
            const_iterator begin() const;
            const_iterator end() const;
        };

    } // namespace tbb
} // namespace oneapi
```

Requirements:

- The `Value` type must meet the *BlockedRangeValue requirements*.

Member functions

type **size_type**

The type for measuring the size of a `blocked_range`. The type is always a `size_t`.

type **const_iterator**

The type of a value in the range. Despite its name, the `const_iterator` type is not necessarily an STL iterator; it merely needs to meet the *BlockedRangeValue requirements*. However, it is convenient to call it `const_iterator` so that if it is a const_iterator, the `blocked_range` behaves like a read-only STL container.

blocked_range(Value begin, Value end, *size_type* grainsize = 1)

Requirements: The parameter `grainsize` must be positive. The debug version of the library raises an assertion failure if this requirement is not met.

Effects: Constructs a `blocked_range` representing the half-open interval `[begin, end)` with the given `grainsize`.

Example: The statement "`blocked_range<int> r(5, 14, 2);`" constructs a range of `int` that contains the values 5 through 13 inclusive, with the grain size of 2. Afterwards, `r.begin()==5` and `r.end()==14`.

blocked_range(*blocked_range* &range, split)

Basic splitting constructor.

Requirements: `is_divisible()` is true.

Effects: Partitions `range` into two subranges. The newly constructed `blocked_range` is approximately the second half of the original `range`, and `range` is updated to be the remainder. Each subrange has the same `grainsize` as the original range.

Example: Let `r` be a `blocked_range` that represents a half-open interval `[i, j)` with a grain size `g`. Running the statement `blocked_range<int> s(r, split);` subsequently causes r to represent `[i, i+(j-i)/2)` and `s` to represent `[i+(j-i)/2, j)`, both with grain size `g`.

blocked_range(*blocked_range* &range, *proportional_split* proportion)

Proportional splitting constructor.

Requirements: `is_divisible()` is true.

Effects: Partitions `range` into two subranges such that the ratio of their sizes is close to the ratio of `proportion.left()` to `proportion.right()`. The newly constructed `blocked_range` is the subrange at the right, and `range` is updated to be the subrange at the left.

Example: Let `r` be a `blocked_range` that represents a half-open interval `[i, j)` with a grain size `g`. Running the statement `blocked_range<int> s(r, proportional_split(2, 3));` subsequently causes `r` to represent `[i, i+2*(j-i)/(2+3))` and `s` to represent `[i+2*(j-i)/(2+3), j)`, both with grain size `g`.

size_type **size**() const

Requirements: `end()<begin()` is false.

Effects: Determines size of range.

Returns: `end()-begin()`.

bool **empty**() const

Effects: Determines if range is empty.

Returns: `!(begin()<end())`

size_type **grainsize**() const

Returns: Grain size of range.

bool **is_divisible**() const

Requirements: end()<begin() is false.

Effects: Determines if the range can be split into subranges.

Returns: True if size()>grainsize(); false, otherwise.

const_iterator **begin**() const

Returns: Inclusive lower bound of the range.

const_iterator **end**() const

Returns: Exclusive upper bound of the range.

See also:

- *parallel_reduce*
- *parallel_for*
- *parallel_scan*

blocked_range2d

[algorithms.blocked_range2d]

Class template that represents a recursively divisible two-dimensional half-open interval.

A blocked_range2d represents a half-open two-dimensional range [i0,j0)×[i1,j1). Each axis of the range has its own splitting threshold. A blocked_range2d is divisible if either axis is divisible.

A blocked_range2d meets the *Range requirements*.

```
// Defined in header <oneapi/tbb/blocked_range2d.h>

namespace oneapi {
    namespace tbb {

        template<typename RowValue, typename ColValue=RowValue>
        class blocked_range2d {
        public:
            // Types
            using row_range_type = blocked_range<RowValue>;
            using col_range_type = blocked_range<ColValue>;

            // Constructors
            blocked_range2d(
                RowValue row_begin, RowValue row_end,
                typename row_range_type::size_type row_grainsize,
                ColValue col_begin, ColValue col_end,
                typename col_range_type::size_type col_grainsize);
            blocked_range2d( RowValue row_begin, RowValue row_end,
                             ColValue col_begin, ColValue col_end );
            // Splitting constructors
            blocked_range2d( blocked_range2d& r, split );
            blocked_range2d( blocked_range2d& r, proportional_split proportion );

            // Capacity
```

(continues on next page)

(continued from previous page)

```
            bool empty() const;

            // Access
            bool is_divisible() const;
            const row_range_type& rows() const;
            const col_range_type& cols() const;
        };

    } // namespace tbb
} // namespace oneapi
```

Requirements:

- The *RowValue* and *ColValue* must meet the *blocked_range requirements*

Member types

```
using row_range_type = blocked_range<RowValue>;
```

The type of the row values.

```
using col_range_type = blocked_range<ColValue>;
```

The type of the column values.

Member functions

```
blocked_range2d(
    RowValue row_begin, RowValue row_end,
    typename row_range_type::size_type row_grainsize,
    ColValue col_begin, ColValue col_end,
    typename col_range_type::size_type col_grainsize);
```

Effects: Constructs a `blocked_range2d` representing a two-dimensional space of values. The space is the half-open Cartesian product `[row_begin, row_end) x [col_begin, col_end)`, with the given grain sizes for the rows and columns.

Example: The statement `blocked_range2d<char,int> r('a', 'z'+1, 3, 0, 10, 2 );` constructs a two-dimensional space that contains all value pairs of the form `(i, j)`, where `i` ranges from `'a'` to `'z'` with a grain size of 3, and `j` ranges from 0 to 9 with a grain size of 2.

```
blocked_range2d(RowValue row_begin, RowValue row_end,
                ColValue col_begin, ColValue col_end);
```

Same as `blocked_range2d(row_begin,row_end,1,col_begin,col_end,1)`.

```
blocked_range2d(blocked_range2d& range, split);
```

Basic splitting constructor.

Requirements: `is_divisible()` is true.

Effects: Partitions range into two subranges. The newly constructed `blocked_range2d` is approximately the second half of the original `range`, and `range` is updated to be the remainder. Each subrange has the same grain size as the original `range`. Splitting is done either by rows or columns. The choice of which axis to split is intended to cause, after repeated splitting, the subranges to approach the aspect ratio of the respective row and column grain sizes.

```
blocked_range2d(blocked_range2d& range, proportional_split proportion);
```

Proportional splitting constructor.

Requirements: `is_divisible()` is true.

Effects: Partitions `range` into two subranges in the given `proportion` across one of its axes. The choice of which axis to split is made in the same way as for the basic splitting constructor; then, proportional splitting is done for the chosen axis. The second axis and the grain sizes for each subrange remain the same as in the original range.

```
bool empty() const;
```

Effects: Determines if range is empty.

Returns: `rows().empty()||cols().empty()`

```
bool is_divisible() const;
```

Effects: Determines if range can be split into subranges.

Returns: `rows().is_divisible()||cols().is_divisible()`

```
const row_range_type& rows() const;
```

Returns: Range containing the rows of the value space.

```
const col_range_type& cols() const;
```

Returns: Range containing the columns of the value space.

See also:

- *blocked_range*

blocked_range3d

[algorithms.blocked_range3d]

Class template that represents a recursively divisible three-dimensional half-open interval.

A `blocked_range3d` is the three-dimensional extension of `blocked_range2d`.

```
namespace oneapi {
    namespace tbb {
        template<typename PageValue, typename RowValue=PageValue, typename␣
↪ColValue=RowValue>
        class blocked_range3d {
        public:
            // Types
            using page_range_type = blocked_range<PageValue>;
            using row_range_type = blocked_range<RowValue>;
            using col_range_type = blocked_range<ColValue>;
```

(continues on next page)

(continued from previous page)

```
            // Constructors
            blocked_range3d(
                PageValue page_begin, PageValue page_end,
                typename page_range_type::size_type page_grainsize,
                RowValue row_begin, RowValue row_end,
                typename row_range_type::size_type row_grainsize,
                ColValue col_begin, ColValue col_end,
                typename col_range_type::size_type col_grainsize );
            blocked_range3d( PageValue page_begin, PageValue page_end
                             RowValue row_begin, RowValue row_end,
                             ColValue col_begin, ColValue col_end );
            blocked_range3d( blocked_range3d& r, split );
            blocked_range3d( blocked_range3d& r, proportional_split& proportion );

            // Capacity
            bool empty() const;

            // Access
            bool is_divisible() const;
            const page_range_type& pages() const;
            const row_range_type& rows() const;
            const col_range_type& cols() const;
        };

    } // namespace tbb
} // namespace oneapi
```

Requirements:

- The *PageValue*, *RowValue* and *ColValue* must meet the *blocked_range requirements*

Member types

```
using page_range_type = blocked_range<PageValue>;
```

The type of the page values.

```
using row_range_type = blocked_range<RowValue>;
```

The type of the row values.

```
using col_range_type = blocked_range<ColValue>;
```

The type of the column values.

Member functions

```
blocked_range3d(PageValue page_begin, PageValue page_end,
    typename page_range_type::size_type page_grainsize,
    RowValue row_begin, RowValue row_end,
    typename row_range_type::size_type row_grainsize,
    ColValue col_begin, ColValue col_end,
    typename col_range_type::size_type col_grainsize);
```

Effects: Constructs a `blocked_range3d` representing a three-dimensional space of values. The space is the half-open Cartesian product `[page_begin, page_end) x [row_begin, row_end) x [col_begin, col_end)`, with the given grain sizes for the pages, rows and columns.

Example: The statement `blocked_range3d<int,char,int> r(0, 6, 2, 'a', 'z'+1, 3, 0, 10, 2 );` constructs a three-dimensional space that contains all value pairs of the form `(i, j, k)`, where `i` ranges from 0 to 6 with a grain size of 2, `j` ranges from `'a'` to `'z'` with a grain size of 3, and `k` ranges from 0 to 9 with a grain size of 2.

```
blocked_range3d(PageValue page_begin, PageValue page_end,
        RowValue row_begin, RowValue row_end,
        ColValue col_begin, ColValue col_end);
```

Same as `blocked_range3d(page_begin,page_end,1,row_begin,row_end,1,col_begin,col_end,1)`.

```
blocked_range3d( blocked_range3d& range, split );
```

Basic splitting constructor.

Requirements: `is_divisible()` is true.

Effects: Partitions `range` into two subranges. The newly constructed `blocked_range3d` is approximately the second half of the original `range`, and `range` is updated to be the remainder. Each subrange has the same grain size as the original `range`. Splitting is done either by pages, rows, or columns. The choice of which axis to split is intended to cause, after repeated splitting, the subranges to approach the aspect ratio of the respective page, row, and column grain sizes.

```
blocked_range3d( blocked_range3d& range, proportional_split proportion );
```

Proportional splitting constructor.

Requirements: `is_divisible()` is true.

Effects: Partitions `range` into two subranges in the given `proportion` across one of its axes. The choice of which axis to split is made in the same way as for the basic splitting constructor; then, proportional splitting is done for the chosen axis. The second axis and the grain sizes for each subrange remain the same as in the original range.

```
bool empty() const;
```

Effects: Determines if range is empty.

Returns: `pages.empty()||rows().empty()||cols().empty()`

```
bool is_divisible() const;
```

Effects: Determines if the range can be split into subranges.

Returns: `pages().is_divisible()||rows().is_divisible()||cols().is_divisible()`

```
const page_range_type& pages() const;
```

Returns: Range containing the pages of the value space.

```
const row_range_type& rows() const;
```

Returns: Range containing the rows of the value space.

```
const col_range_type& cols() const;
```

Returns: Range containing the columns of the value space.

See also:

- *blocked_range*
- *blocked_range2d*

Partitioners

A partitioner specifies how a loop template should partition its work among threads.

auto_partitioner

[algorithms.auto_partitioner]

Specifies that a parallel loop should optimize its range subdivision based on work-stealing events.

A loop template with an `auto_partitioner` attempts to minimize range splitting while providing ample opportunities for work stealing.

The range subdivision is initially limited to S subranges, where S is proportional to the number of threads specified by the *global_contol* or *task_arena*. Each of these subranges is not divided further unless it is stolen by an idle thread. If stolen, it is further subdivided to create additional subranges. Thus a loop template with an `auto_partitioner` creates additional subranges only when it is necessary to balance a load.

An `auto_partitioner` performs sufficient splitting to balance load, not necessarily splitting as finely as `Range::is_divisible` permits. When used with classes such as `blocked_range`, the selection of an appropriate grain size is less important, and often acceptable performance can be achieved with the default grain size of 1.

The `auto_partitioner` class satisfies the *CopyConstructibe* requirement from the ISO C++ [utility.arg.requirements] section.

Tip: When using `auto_partitioner` and a `blocked_range` for a parallel loop, the body may receive a subrange larger than the grain size of the `blocked_range`. Therefore, do not assume that the grain size is an upper bound of the subrange size. Use `simple_partitioner` if an upper bound is required.

```
// Defined in header <oneapi/tbb/partitioner.h>

namespace oneapi {
    namespace tbb {

        class auto_partitioner {
        public:
```

(continues on next page)

(continued from previous page)

```
            auto_partitioner() = default;
            ~auto_partitioner() = default;
        };

    } // namespace tbb
} // namespace oneapi
```

affinity_partitioner

[algorithms.affinity_partitioner]

Hints that loop iterations should be assigned to threads in a way that optimizes for cache affinity.

An `affinity_partitioner` hints that execution of a loop template should use the same task affinity pattern for splitting the work as used by previous execution of the loop (or another loop) with the same `affinity_partitioner` object.

`affinity_partitioner` uses proportional splitting when it is enabled for a *Range* type.

Unlike the other partitioners, it is important that the same `affinity_partitioner` object be passed to the loop templates to be optimized for affinity.

The `affinity_partitioner` class satisfies the *CopyConstructibe* requirement from the ISO C++ [utility.arg.requirements] section.

```
// Defined in header <oneapi/tbb/partitioner.h>

namespace oneapi {
    namespace tbb {

        class affinity_partitioner {
        public:
            affinity_partitioner() = default;
            ~affinity_partitioner() = default;
        };

    } // namespace tbb
} // namespace oneapi
```

See also:

- *Range named requirement*

static_partitioner

[algorithms.static_partitioner]

Specifies that a parallel algorithm should distribute the work uniformly across threads and should not do additional load balancing.

An algorithm with a `static_partitioner` distributes the range across threads in subranges of approximately equal size. The number of subranges is equal to the number of threads that can possibly participate in task execution, as specified by *global_contol* or *task_arena* classes. These subranges are not further split.

Caution: The regularity of subrange sizes is not guaranteed if the range type does not support proportional splitting, or if the grain size is set larger than the size of the range divided by the number of threads participating in task execution.

In addition, `static_partitioner` uses a deterministic task affinity pattern to hint the task scheduler how the subranges should be assigned to threads.

The `static_partitioner` class satisfies the *CopyConstructibe* requirement from the ISO C++ [utility.arg.requirements] section.

Tip: Use `static_partitioner` to:

- Parallelize small well-balanced workloads where enabling additional load balancing opportunities brings more overhead than performance benefits.
- Port OpenMP* parallel loops with `schedule(static)` if deterministic work partitioning across threads is important.

```
// Defined in header <oneapi/tbb/partitioner.h>

namespace oneapi {
    namespace tbb {

        class static_partitioner {
        public:
           static_partitioner() = default;
           ~static_partitioner() = default;
        };

    } // namespace tbb
} // namespace oneapi
```

See also:

- *Range named requirement*

simple_partitioner

[algorithms.simple_partitioner]

Specifies that a parallel loop should recursively split its range until it cannot be further subdivided.

A `simple_partitioner` specifies that a loop template should recursively divide its range until for each subrange *r*, the condition `!r.is_divisible()` holds. This is the default behavior of the loop templates that take a range argument.

The `simple_partitioner` class satisfies the *CopyConstructibe* requirement from the ISO C++ [utility.arg.requirements] section.

Tip: When using `simple_partitioner` and a `blocked_range` for a parallel loop, make sure to specify an appropriate grain size for the `blocked_range`. The default grain size is 1, which may make the subranges much too small for efficient execution.

```
// Defined in header <oneapi/tbb/partitioner.h>

namespace oneapi {
    namespace tbb {

        class simple_partitioner {
        public:
            simple_partitioner() = default;
            ~simple_partitioner() = default;
        };

    } // namespace tbb
} // namespace oneapi
```

See also:

- *Range named requirement*

Split Tags

proportional split

[algorithms.proportional_split]

Type of an argument for a proportional splitting constructor of *Range*.

An argument of type `proportional_split` may be used by classes that satisfy *Range requirements* to distinguish a proportional splitting constructor from a basic splitting constructor and from a copy constructor, and to suggest a ratio in which a particular instance of the class should be split.

```
// Defined in header <oneapi/tbb/blocked_range.h>
// Defined in header <oneapi/tbb/blocked_range2d.h>
// Defined in header <oneapi/tbb/blocked_range3d.h>
// Defined in header <oneapi/tbb/partitioner.h>
// Defined in header <oneapi/tbb/parallel_for.h>
// Defined in header <oneapi/tbb/parallel_reduce.h>
// Defined in header <oneapi/tbb/parallel_scan.h>

namespace oneapi {
    namespace tbb {
        class proportional_split {
        public:
            proportional_split(std::size_t _left = 1, std::size_t _right = 1);

            std::size_t left() const;
            std::size_t right() const;

            explicit operator split() const;
        };
    } // namespace tbb
} // namespace oneapi
```

Member functions

proportional_split(std::size_t _left = 1, std::size_t _right = 1)

Constructs a proportion with the ratio specified by coefficients *_left* and *_right*.

std::size_t **left**() const

Returns the size of the left part of the proportion.

std::size_t **right**() const

Returns the size of the right part of the proportion.

explicit **operator split**() const

Makes `proportional_split` convertible to the `split` type to use with ranges that do not support proportional splitting.

See also:

- *split*
- *Range requirements*

split

[algorithms.split]

Type of an argument for a splitting constructor of *Range*. An argument of type `split` is used to distinguish a splitting constructor from a copy constructor.

```
// Defined in header <oneapi/tbb/blocked_range.h>
// Defined in header <oneapi/tbb/blocked_range2d.h>
// Defined in header <oneapi/tbb/blocked_range3d.h>
// Defined in header <oneapi/tbb/partitioner.h>
// Defined in header <oneapi/tbb/parallel_for.h>
// Defined in header <oneapi/tbb/parallel_reduce.h>
// Defined in header <oneapi/tbb/parallel_scan.h>

class split;
```

See also:

- *Range requirements*

8.2.3 Flow Graph

[flow_graph]

In addition to loop parallelism, the oneAPI Threading Building Blocks (oneTBB) library also supports graph parallelism. With this feature, highly scalable and completely sequential graphs can be created.

There are three types of components used to implement a graph:

- A `graph` class instance
- Nodes
- Ports and edges

Graph Class

The `graph` class instance owns all the tasks created on behalf of the flow graph. Users can wait on the `graph` if they need to wait for the completion of all of the tasks related to the flow graph execution. Users can also register external interactions with the `graph` and run tasks under the ownership of the flow graph.

graph

[flow_graph.graph]

Class that serves as a handle to a flow graph of nodes and edges.

```
// Defined in header <oneapi/tbb/flow_graph.h>

namespace oneapi {
namespace tbb {
namespace flow {

    class graph {
    public:
        graph();
        graph(task_group_context& context);
        ~graph();

        void wait_for_all();

        void reset(reset_flags f = rf_reset_protocol);
        void cancel();
        bool is_cancelled();
        bool exception_thrown();
    };

} // namespace flow
} // namespace tbb
} // namespace oneapi
```

reset_flags enumeration

reset_flags Enumeration

[flow_graph.reset_flags]

A `reset_flags` enumeration represents flags that can be passed to the `graph::reset()` function.

```
// Defined in header <oneapi/tbb/flow_graph.h>

namespace oneapi {
namespace tbb {
namespace flow {

    enum reset_flags {
        rf_reset_protocol = /*implementation-defined*/,
```

(continues on next page)

(continued from previous page)

```
        rf_reset_bodies = /*implementation-defined*/,
        rf_clear_edges = /*implementation-defined*/
    };

} // namespace flow
} // namespace tbb
} // namespace oneapi
```

Its enumerated values and their meanings are as follows:

- `rf_reset_protocol` - All buffers are emptied, internal state of nodes reinitialized. All calls to `reset()` perform these actions.
- `rf_reset_bodies` - When nodes with bodies are created, the body specified in the constructor is copied and preserved. When `rf_reset_bodies` is specified, the current body of the node is deleted and replaced with a copy of the body saved during construction.

  Caution: If the body contains state which has an external component (such as a file descriptor), the node may not behave the same on re-execution of the graph after body replacement. In this case, the node should be re-created.

- `rf_clear_edges` - All edges are removed from the graph.

Member functions

graph(*task_group_context* &context)

Constructs a graph with no nodes. If `context` is specified, the graph tasks are executed in this context. By default, the graph is executed in a bound context of its own.

~graph()

Calls `wait_for_all()` on the graph, then destroys the graph.

void **wait_for_all**()

Blocks execution until all tasks associated with the graph have completed or cancelled.

void **reset**(reset_flags f = rf_reset_protocol)

Resets the graph according to the specified flags. Flags to `reset()` can be combined with bitwise-`or`.

Note: `reset()` is a thread-unsafe operation, don't call it concurrently.

void **cancel**()

Cancels all tasks in the graph.

bool **is_cancelled**()

Returns: `true` if the graph was cancelled during the last call to `wait_for_all()`; `false`, otherwise.

bool **exception_thrown**()

Returns: `true` if during the last call to `wait_for_all()` an exception was thrown; `false`, otherwise.

Nodes

Abstract Interfaces

To be used as a graph node type, a class needs to inherit certain abstract types and implement the corresponding interfaces. `graph_node` is the base class for any other node type; its interfaces always have to be implemented. If a node sends messages to other nodes, it has to implement the `sender` interface, while with the `receiver` interface the node may accept messages. For nodes that have multiple inputs and/or outputs, each input port is a `receiver` and each output port is a `sender`.

graph_node

[flow_graph.graph_node]

A base class for all graph nodes.

```
namespace oneapi {
namespace tbb {
namespace flow {

    class graph_node {
    public:
        explicit graph_node( graph &g );
        virtual ~graph_node();
    };

} // namespace flow
} // namespace tbb
} // namespace oneapi
```

The `graph_node` class is a base class for all flow graph nodes. The virtual destructor allows flow graph nodes to be destroyed through pointers to `graph_node`. For example, a `vector< graph_node * >` can be used to hold the addresses of flow graph nodes that will need to be destroyed later.

sender

[flow_graph.sender]

A base class for all nodes that may send messages.

```
namespace oneapi {
namespace tbb {
namespace flow {

    template< typename T >
    class sender { /*unspecified*/ };

} // namespace flow
} // namespace tbb
} // namespace oneapi
```

The `T` type is a message type.

receiver

[flow_graph.receiver]

A base class for all nodes that may receive messages.

```
namespace oneapi {
namespace tbb {
namespace flow {

    template< typename T >
    class receiver { /*unspecified*/ };

} // namespace flow
} // namespace tbb
} // namespace oneapi
```

The `T` type is a message type.

Properties

Every node in a flow graph has its own properties.

Forwarding and Buffering

[flow_graph.forwarding_and_buffering]

Forwarding

In a `flow::graph`, nodes that forward messages to successors have one of two possible forwarding policies, which are a property of the node:

- **broadcast-push** - the message will be pushed to as many successors as will accept the message. If no successor accepts the message, the fate of the message depends on the output buffering policy of the node.
- **single-push** - if the message is accepted by a successor, no further push of that message will occur. If a successor rejects the message, the next successor in the set is tried. This continues until a successor accepts the message, or all successors have been attempted. If no successor accepts the message, it will be retained for a possible future resend. Message that is successfully transferred to a successor is removed from the node.

Buffering

There are two policies for handling a message that cannot be pushed to any successor:

- **buffering** - if no successor accepts a message, it is stored so subsequent node processing can use it. Nodes that buffer outputs have "yes" in the "try_get()?" column below.
- **discarding** - if no successor accepts a message, it is discarded and has no further effect on graph execution. Nodes that discard outputs have "no" in the "try_get()?" column below.

The following table lists the policies of each node:

Table 4: Buffering and Forwarding properties summary

Node	try_get()?	Forwarding
Functional Nodes		
input_node	yes	broadcast-push
function_node<rejecting>	no	broadcast-push
function_node<queueing>	no	broadcast-push
continue_node	no	broadcast-push
multifunction_node<rejecting>	no	broadcast-push
multifunction_node<queueing>	no	broadcast-push
Buffering Nodes		
buffer_node	yes	single-push
priority_queue_node	yes	single-push
queue_node	yes	single-push
sequencer_node	yes	single-push
overwrite_node	yes	broadcast-push
write_once_node	yes	broadcast-push
Split/Join Nodes		
join_node<queueing>	yes	broadcast-push
join_node<reserving>	yes	broadcast-push
join_node<tag_matching>	yes	broadcast-push
split_node	no	broadcast-push
indexer_node	no	broadcast-push
Other Nodes		
broadcast_node	no	broadcast-push
limiter_node	no	broadcast-push

Functional Nodes

Functional nodes do computations in response to input messages (if any), and send the result or a signal to their successors.

continue_node

[flow_graph.continue_node]

A node that executes a specified body object when triggered.

```
// Defined in header <oneapi/tbb/flow_graph.h>

namespace oneapi {
namespace tbb {
namespace flow {

    template< typename Output, typename Policy = /*implementation-defined*/ >
    class continue_node : public graph_node, public receiver<continue_msg>, public sender
↪<Output> {
    public:
        template<typename Body>
        continue_node( graph &g, Body body, node_priority_t priority = no_priority );
        template<typename Body>
```

(continues on next page)

(continued from previous page)

```
        continue_node( graph &g, Body body, Policy /*unspecified*/ = Policy(),
                       node_priority_t priority = no_priority );

        template<typename Body>
        continue_node( graph &g, int number_of_predecessors, Body body,
                       node_priority_t priority = no_priority );
        template<typename Body>
        continue_node( graph &g, int number_of_predecessors, Body body,
                       Policy /*unspecified*/ = Policy(), node_priority_t priority = no_
↪priority );

        continue_node( const continue_node &src );
        ~continue_node();

        bool try_put( const input_type &v );
        bool try_get( output_type &v );
    };

} // namespace flow
} // namespace tbb
} // namespace oneapi
```

Requirements:

- The type `Output` must meet the *CopyConstructible* requirements from [copyconstructible] ISO C++ Standard section.
- The type `Policy` can be specified as *lightweight policy* or defaulted.
- The type `Body` must meet the *ContinueNodeBody requirements*.

A `continue_node` is a `graph_node`, `receiver<continue_msg>`, and `sender<Output>`.

This node is used for nodes that wait for their predecessors to complete before executing, but no explicit data is passed across the incoming edges.

A `continue_node` maintains an internal threshold that defines the number of predecessors. This value can be provided at construction. Call of the *make_edge function* with `continue_node` as a receiver increases its threshold. Call of the *remove_edge function* with `continue_node` as a receiver decreases it.

Each time the number of `try_put()` calls reaches the defined threshold, node's `body` is called and the node starts counting the number of `try_put()` calls from the beginning.

`continue_node` has a *discarding* and *broadcast-push properties*.

The body object passed to a `continue_node` is copied. Updates to member variables do not affect the original object used to construct the node. If the state held within a body object must be inspected from outside of the node, the *copy_body function* can be used to obtain an updated copy.

Member functions

```
template<typename Body>
continue_node( graph &g, Body body, node_priority_t priority = no_priority );
```

Constructs a `continue_node` that invokes `body`. The internal threshold is set to 0.

This function specifies *node priority*.

```
template<typename Body>
continue_node( graph &g, Body body, Policy /*unspecified*/ = Policy(),
               node_priority_t priority = no_priority );
```

Constructs a `continue_node` that invokes `body`. The internal threshold is set to 0.

This function specifies *lightweight policy* and *node priority*.

```
template<typename Body>
continue_node( graph &g, int number_of_predecessors, Body body,
               node_priority_t priority = no_priority );
```

Constructs a `continue_node` that invokes `body`. The internal threshold is set to `number_of_predecessors`.

This function specifies *node priority*.

```
template<typename Body>
continue_node( graph &g, int number_of_predecessors, Body body,
               Policy /*unspecified*/ = Policy(), node_priority_t priority = no_priority␣
↪);
```

Constructs a `continue_node` that invokes `body`. The internal threshold is set to `number_of_predecessors`.

This function specifies *lightweight policy* and *node priority*.

```
template<typename Body>
continue_node( graph &g, int number_of_predecessors, Body body );
```

Constructs a `continue_node` that invokes `body`. The internal threshold is set to `number_of_predecessors`.

```
continue_node( const continue_node &src )
```

Constructs a `continue_node` that has the same initial state that `src` had after its construction. It does not copy the current count of `try_puts` received, or the current known number of predecessors. The `continue_node` that is constructed has a reference to the same `graph` object as `src`, has a copy of the initial `body` used by `src`, and only has a non-zero threshold if `src` is constructed with a non-zero threshold.

The new body object is copy-constructed from a copy of the original body provided to `src` at its construction.

```
bool try_put( const Input &v )
```

Increments the count of `try_put()` calls received. If the incremented count is equal to the number of known predecessors, performs the `body` function object execution. It does not wait for the execution of the body to complete.

Returns: `true`

```
bool try_get( Output &v )
```

Returns: `false`

Deduction Guides

```
template <typename Body, typename Policy>
continue_node(graph&, Body, Policy, node_priority_t = no_priority)
    -> continue_node<continue_output_t<std::invoke_result_t<Body, continue_msg>>, Policy>
;

template <typename Body, typename Policy>
continue_node(graph&, int, Body, Policy, node_priority_t = no_priority)
    -> continue_node<continue_output_t<std::invoke_result_t<Body, continue_msg>>, Policy>
;

template <typename Body>
continue_node(graph&, Body, node_priority_t = no_priority)
    -> continue_node<continue_output_t<std::invoke_result_t<Body, continue_msg>>, /
*default-policy*/>;

template <typename Body>
continue_node(graph&, int, Body, node_priority_t = no_priority)
    -> continue_node<continue_output_t<std::invoke_result_t<Body, continue_msg>>, /
*default-policy*/>;
```

Where:

- `continue_output_t<Output>` is an alias to *Output* template argument type. If *Output* specified as `void`, `continue_output_t<Output>` is an alias to `continue_msg` type.

Example

A set of `continue_nodes` forms a *Dependency Flow Graph*.

function_node

[flow_graph.function_node]

A node that executes a user-provided body on incoming messages.

```
// Defined in header <oneapi/tbb/flow_graph.h>

namespace oneapi {
namespace tbb {
namespace flow {

    template < typename Input, typename Output = continue_msg, typename Policy = /
↪*implementation-defined*/ >
    class function_node : public graph_node, public receiver<Input>, public sender
↪<Output> {
    public:
        template<typename Body>
        function_node( graph &g, size_t concurrency, Body body, Policy /*unspecified*/ =␣
↪Policy(),
                       node_priority_t priority = no_priority );
        template<typename Body>
        function_node( graph &g, size_t concurrency, Body body,
                       node_priority_t priority = no_priority );
        ~fuction_node();

        function_node( const function_node &src );

        bool try_put( const Input &v );
        bool try_get( Output &v );
    };

} // namespace flow
} // namespace tbb
} // namespace oneapi
```

Requirements:

- The `Input` type must meet the *DefaultConstructible* requirements from [defaultconstructible] and the *CopyConstructible* requirements from [copyconstructible] ISO C++ Standard sections.
- The `Output` type must meet the *CopyConstructible* requirements from [copyconstructible] ISO C++ Standard section.
- The type `Policy` may be specified as *lightweight, queueing and rejecting policies* or defaulted.
- The type `Body` must meet the *FunctionNodeBody requirements*. Since C++17, `Body` may also be a pointer to a const member function in `Input` that returns `Output` or a pointer to a data member in `Input` of type `Output`.

`function_node` has a user-settable concurrency limit. It can be set to one of *predefined values*. The user can also provide a value of type `std::size_t` to limit concurrency to a value between 1 and *tbb::flow::unlimited*.

Messages that cannot be immediately processed due to concurrency limits are handled according to the *Policy* template argument.

`function_node` is a `graph_node`, `receiver<Input>`, and `sender<Output>`.

`function_node` has a *discarding* and *broadcast-push properties*.

The body object passed to a `function_node` is copied. Updates to member variables do not affect the original object used to construct the node. If the state held within a body object must be inspected from outside of the node, the *copy_body* function can be used to obtain an updated copy.

Member functions

```
template<typename Body>
function_node( graph &g, size_t concurrency, Body body,
               node_priority_t priority = no_priority );
```

Constructs a `function_node` that invokes a copy of `body`. Most of `concurrency` calls to `body` can be made concurrently.

Use this function to specify *node priority*.

```
template<typename Body>
function_node( graph &g, size_t concurrency, Body body, Policy /*unspecified*/ =␣
↪Policy(),
               node_priority_t priority = no_priority );
```

Constructs a `function_node` that invokes a copy of `body`. Most of `concurrency` calls to `body` can be made concurrently.

Use this function to specify *policy* and *node priority*.

```
function_node( const function_node &src )
```

Constructs a `function_node` that has the same initial state that `src` had when it was constructed. The `function_node` that is constructed has a reference to the same `graph` object as `src`, has a copy of the initial body used by `src`, and has the same concurrency threshold as `src`. The predecessors and successors of `src` are not copied.

The new body object is copy-constructed from a copy of the original body provided to `src` at its construction. Changes made to member variables in `src`'s body after the construction of `src` do not affect the body of the new `function_node`.

```
bool try_put( const Input &v )
```

If the concurrency limit allows, executes the user-provided body on the incoming message `v`. Otherwise, depending on the policy of the node, either queues the incoming message `v` or rejects it.

Returns: `true` if the input was accepted; and `false`, otherwise.

```
bool try_get( Output &v )
```

Returns: `false`

Deduction Guides

```
template <typename Body, typename Policy>
function_node(graph&, size_t, Body, Policy, node_priority_t = no_priority)
    ->function_node<std::decay_t<input_t<Body>>, output_t<Body>, Policy>;

template <typename Body>
function_node(graph&, size_t, Body, node_priority_t = no_priority)
    ->function_node<std::decay_t<input_t<Body>>, output_t<Body>, /*default-policy*/>;
```

Where:

- `input_t` is an alias to `Body` input argument type.
- `output_t` is an alias to `Body` return type.

Example

Data Flow Graph example illustrates how `function_node` performs computation on input data and passes the result to successors.

input_node

[flow_graph.input_node]

A node that generates messages by invoking the user-provided functor and broadcasts the result to all of its successors.

```
// Defined in header <oneapi/tbb/flow_graph.h>

namespace oneapi {
namespace tbb {
namespace flow {

    template < typename Output >
    class input_node : public graph_node, public sender<Output> {
    public:
        template< typename Body >
        input_node( graph &g, Body body );
        input_node( const input_node &src );
        ~input_node();

        void activate();
        bool try_get( Output &v );
    };

} // namespace flow
} // namespace tbb
} // namespace oneapi
```

Requirements:

- The `Output` type must meet the *DefaultConstructible* requirements from [defaultconstructible], *CopyConstructible* requirements from [copyconstructible] and *CopyAssignable* requirements from [copyassignable] ISO C++ Standard sections.

- The type `Body` must meet the *InputNodeBody requirements*.

This node can have no predecessors. It executes a user-provided `body` function object to generate messages that are broadcast to all successors. It is a serial node and never calls its `body` concurrently. This node can buffer a single item. If no successor accepts an item that it has generated, the message is buffered and provided to successors before a new item is generated.

`input_node` is a `graph_node` and `sender<Output>`.

`input_node` has a *buffering* and *broadcast-push* *properties*.

An `input_node` continues to invoke `body` and broadcast messages until the `body` toggles `fc.stop()` or it has no valid successors. A message may be generated and then rejected by all successors. In this case, the message is buffered and will be the next message sent once a successor is added to the node or `try_get` is called. Calls to `try_get` return a message from the buffer, or invoke `body` to attempt to generate a new message. A call to `body` is made only when the buffer is empty.

The body object passed to an `input_node` is copied. Updates to member variables do not affect the original object used to construct the node. If the state held within a body object must be inspected from outside of the node, the *copy_body function* can be used to obtain an updated copy.

Member functions

template<typename **Body**>
input_node(*graph* &g, *Body* body)

Constructs an `input_node` that invokes `body`. By default, the node is created in an inactive state, which means that messages are not generated until a call to `activate` is made.

input_node(const *input_node* &src)

Constructs an `input_node` that has the same initial state that `src` had when it was constructed. The `input_node` that is constructed has a reference to the same `graph` object as `src`, has a copy of the initial body used by `src`, and has the same initial active state as `src`. The successors of `src` are not copied.

The new body object is copy-constructed from a copy of the original body provided to `src` at its construction. Changes made to member variables in `src` body after the construction of `src` do not affect the body of the new `input_node`.

void **activate**()

Sets the `input_node` to the active state, which enables messages generation.

bool **try_get**(Output &v)

Copies the message from the buffer to `v` if available, or, if the node is in active state, invokes `body` to attempt to generate a new message that will be copied into `v`.

Returns: `true` if a message is copied to `v`; `false`, otherwise.

Deduction Guides

```
template <typename Body>
input_node(graph&, Body) -> input_node<std::decay_t<input_t<Body>>>;
```

Where:

- `input_t` is an alias to `Body` input argument type.

multifunction_node

[flow_graph.multifunction_node]

A node that used for nodes that receive messages at a single input port and may generate one or more messages that are broadcast to successors.

```
// Defined in header <oneapi/tbb/flow_graph.h>

namespace oneapi {
namespace tbb {
namespace flow {

    template < typename Input, typename Output, typename Policy = /*implementation-
↪defined*/ >
    class multifunction_node : public graph_node, public receiver<Input> {
    public:
        template<typename Body>
        multifunction_node( graph &g, size_t concurrency, Body body, Policy /
↪*unspecified*/ = Policy(),
                            node_priority_t priority = no_priority );
        template<typename Body>
        multifunction_node( graph &g, size_t concurrency, Body body,
                            node_priority_t priority = no_priority );

        multifunction_node( const multifunction_node& other );
        ~multifunction_node();

        bool try_put( const Input &v );

        using output_ports_type = /*implementation-defined*/;
        output_ports_type& output_ports();
    };

} // namespace flow
} // namespace tbb
} // namespace oneapi
```

Requirements:

- The `Input` type must meet the *DefaultConstructible* requirements from [defaultconstructible] and the *CopyConstructible* requirements from [copyconstructible] ISO C++ Standard sections.
- The type `Policy` can be specified as *lightweight, queueing and rejecting policies* or defaulted.
- The type `Body` must meet the *MultifunctionNodeBody requirements*. Since C++17, `Body` may also be a pointer to a const member function in `Input` that takes `output_ports_type&` argument.

`multifunction_node` has a user-settable concurrency limit. It can be set to one of *predefined values*. The user can also provide a value of type `std::size_t` to limit concurrency to a value between 1 and *tbb::flow::unlimited*.

When the concurrency limit allows, it executes the user-provided body on incoming messages. The body can create one or more output messages and broadcast them to successors.

`multifunction_node` is a `graph_node`, `receiver<InputType>` and has a tuple of `sender<Output>` outputs.

`multifunction_node` has a *discarding* and *broadcast-push properties*.

The body object passed to a `multifunction_node` is copied. Updates to member variables do not affect the original object used to construct the node. If the state held within a body object must be inspected from outside of the node, the *copy_body function* can be used to obtain an updated copy.

Member types

`output_ports_type` is an alias to a `std::tuple` of output ports.

Member functions

```
template<typename Body>
multifunction_node( graph &g, size_t concurrency, Body body,
                    node_priority_t priority = no_priority );
```

Constructs a `multifunction_node` that invokes a copy of `body`. Most `concurrency` calls to `body` can be made concurrently.

Use this function to specify *node priority*.

```
template<typename Body>
multifunction_node( graph &g, size_t concurrency, Body body, Policy /*unspecified*/ =␣
↪Policy(),
                    node_priority_t priority = no_priority );
```

Constructs a `multifunction_node` that invokes a copy of `body`. Most `concurrency` calls to `body` can be made concurrently.

Use this function to specify a *policy* and *node priority*.

```
multifunction_node( const multifunction_node &src )
```

Constructs a `multifunction_node` that has the same initial state that `other` had when it was constructed. The `multifunction_node` that is constructed has a reference to the same `graph` object as `other`, has a copy of the initial `body` used by `other`, and has the same concurrency threshold as `other`. The predecessors and successors of `other` are not copied.

The new body object is copy-constructed from a copy of the original body provided to `other` at its construction. Changes made to member variables in `other` body after the construction of `other` do not affect the body of the new `multifunction_node`.

```
bool try_put( const input_type &v )
```

If the concurrency limit allows, executes the user-provided body on the incoming message `v`. Otherwise, depending on the policy of the node, either queues the incoming message `v` or rejects it.

Returns: `true` if the input was accepted; `false`, otherwise.

```
output_ports_type& output_ports();
```

Returns: a `std::tuple` of output ports.

async_node

[flow_graph.async_node]

A node that enables communication between a flow graph and an external activity managed by the user or another runtime.

```
// Defined in header <oneapi/tbb/flow_graph.h>

namespace oneapi {
namespace tbb {
namespace flow {

    template < typename Input, typename Output, typename Policy = /*implemetation-
↪defined*/ >
    class async_node : public graph_node, public receiver<Input>, public sender<Output> {
    public:
        template<typename Body>
        async_node( graph &g, size_t concurrency, Body body, Policy /*unspecified*/ =␣
↪Policy(),
                    node_priority_t priority = no_priority );
        template<typename Body>
        async_node( graph &g, size_t concurrency, Body body, node_priority_t priority =␣
↪no_priority );

        async_node( const async_node& src );
        ~async_node();

        using gateway_type = /*implementation-defined*/;
        gateway_type& gateway();

        bool try_put( const input_type& v );
        bool try_get( output_type& v );
    };

} // namespace flow
} // namespace tbb
} // namespace oneapi
```

Requirements:

- The `Input` type must meet the *DefaultConstructible* requirements from [defaultconstructible] and the *CopyConstructible* requirements from [copyconstructible] ISO C++ Standard sections.
- The type `Policy` can be specified as *lightweight, queueing and rejecting policies* or defaulted.
- The type `Body` must meet the *AsyncNodeBody requirements*. Since C++17, `Body` may also be a pointer to a const member function in `Input` that takes `gateway_type&` argument.

`async_node` executes a user-provided body on incoming messages. The body typically submits the messages to an external activity for processing outside of the graph. It is responsibility of `body` to be able to pass the message to an

external activity. This node also provides the `gateway_type` interface that allows an external activity to communicate with the flow graph.

`async_node` is a `graph_node`, `receiver<Input>`, and a `sender<Output>`.

`async_node` has a *discarding* and *broadcast-push* *properties*.

`async_node` has a user-settable concurrency limit, which can be set to one of *predefined values*. The user can also provide a value of type `std::size_t` to limit concurrency to a value between 1 and *tbb::flow::unlimited*.

The body object passed to a `async_node` is copied. Updates to member variables do not affect the original object used to construct the node. If the state held within a body object must be inspected from outside of the node, the *copy_body* function can be used to obtain an updated copy.

Member types

`gateway_type` meets the *GatewayType requirements*.

Member functions

```
template<typename Body>
async_node( graph &g, size_t concurrency, Body body,
               node_priority_t priority = no_priority );
```

Constructs an `async_node` that invokes a copy of `body`. The `concurrency` value limits the number of simultaneous `body` invocations for the node.

This function specifies *node priority*.

```
template<typename Body>
async_node( graph &g, size_t concurrency, Body body, Policy /*unspecified*/ = Policy(),
               node_priority_t priority = no_priority );
```

Constructs a `async_node` that invokes a copy of `body`. Most `concurrency` calls to `body` can be made concurrently.

This function specifies a *policy* and *node priority*.

```
async_node( const async_node &src )
```

Constructs an `async_node` that has the same initial state that `src` had when it was constructed. The `async_node` that is constructed has a reference to the same `graph` object as `src`, has a copy of the initial body used by `src`, and has the same concurrency threshold as `src`. The predecessors and successors of `src` are not copied.

The new body object is copy-constructed from a copy of the original body provided to `src` at its construction. Changes made to member variables in `src`'s body after the construction of `src` do not affect the body of the new `async_node.`

```
gateway_type& gateway()
```

Returns reference to the `gateway_type` interface.

```
bool try_put( const input_type& v )
```

If the concurrency limit allows, executes the user-provided body on the incoming message `v`. Otherwise, depending on the policy of the node, either queues the incoming message `v` or rejects it.

Returns: `true` if the input was accepted; and `false`, otherwise.

```
bool try_get( output_type& v )
```

Returns: `false`

Auxiliary

Function Nodes Policies

[flow_graph.function_node_policies]

`function_node`, `multifunction_node`, `async_node` and `continue_node` can be specified by the `Policy` parameter, which is represented as a set of tag classes. This parameter affects behavior of node execution.

```
// Defined in header <oneapi/tbb/flow_graph.h>

namespace oneapi {
namespace tbb {
namespace flow {

    class queueing { /*unspecified*/ };
    class rejecting { /*unspecified*/ };
    class lightweight { /*unspecified*/ };
    class queueing_lightweight { /*unspecified*/ };
    class rejecting_lightweight { /*unspecified*/ };

} // namespace flow
} // namespace tbb
} // namespace oneapi
```

Each policy class satisfies the *CopyConstructible* requirements from [copyconstructible] ISO C++ Standard sections.

Queueing

This policy defines behavior for input messages acceptance. The `queueing` policy means that input messages that cannot be processed right away are kept to be processed when possible.

Rejecting

This policy defines behavior for input messages acceptance. The `rejecting` policy means that input messages that cannot be processed right away are not accepted by the node and it is responsibility of a predecessor to handle this.

Lightweight

This policy allows to specify that the node body takes little time to process, as a non-binding hint for an implementation to reduce overheads associated with the node execution. Any optimization applied by an implementation must have no observable side effects on the node and graph execution.

When combined with another policy, the `lightweight` policy results in extending the behavior of that other policy with the optimization hint. This rule automatically applies to functional nodes that have a default value for the `Policy` template parameter. For example, if the default value of `Policy` is `queueing`, specifying `lightweight` as the `Policy` value is equivalent to specifying `queueing_lightweight`.

The function call `operator()` of a node body must be `noexcept` for lightweight policies to have effect.

Example

The example below shows the application of the `lightweight` policy to a graph with a pipeline topology. It is reasonable to apply the `lightweight` policy to the second and third nodes because the bodies of these nodes are small. This allows the second and third nodes to execute without task scheduling overhead. The `lightweight` policy is not specified for the first node in order to permit concurrent invocations of the graph.

```
#include "oneapi/tbb/flow_graph.h"

int main() {
    using namespace oneapi::tbb::flow;

    graph g;

    function_node< int, int > add( g, unlimited, [](const int &v) {
        return v+1;
    } );
    function_node< int, int, lightweight > multiply( g, unlimited, [](const int &v)␣
↪noexcept {
        return v*2;
    } );
    function_node< int, int, lightweight > cube( g, unlimited, [](const int &v) noexcept
↪{
        return v*v*v;
    } );

    make_edge(add, multiply);
    make_edge(multiply, cube);

    for(int i = 1; i <= 10; ++i)
        add.try_put(i);
    g.wait_for_all();
```

(continues on next page)

(continued from previous page)

```
    return 0;
}
```

Nodes Priorities

[flow_graph.node_priorities]

Flow graph provides interface for setting relative priorities at construction of flow graph functional nodes, guiding threads that execute the graph to prefer nodes with higher priority.

```
namespace oneapi {
namespace tbb {
namespace flow {

    typedef unsigned int node_priority_t;

    const node_priority_t no_priority = node_priority_t(0);

} // namespace flow
} // namespace tbb
} // namespace oneapi
```

`function_node`, `multifunction_node`, `async_node` and `continue_node` has a constructor with parameter of `node_priority_t` type, which sets the node priority in the graph: the larger the specified value for the parameter, the higher the priority. The special constant value `no_priority`, which is also the default value of the parameter, switches priority off for a particular node.

For a particular graph, tasks to execute the nodes whose priority is specified have precedence over tasks for the nodes with lower or no priority value set. When looking for a task to execute, a thread chooses the one with the highest priority from those in the graph that are available for execution.

Example

The following basic example demonstrates prioritization of one path in the graph over the other, which may help to improve overall performance of the graph.

Consider executing the graph from the picture above using two threads. Assume that the nodes `f1` and `f3` take equal time to execute, while the node `f2` takes longer. That makes the nodes `bs`, `f2`, and `fe` constitute the critical path in this graph. Due to the non-deterministic behavior in selection of the tasks, oneTBB might execute nodes `f1` and `f3` in parallel first, which would make the whole graph execution time last longer than the case when one of the threads chooses the node `f2` just after the broadcast node. By setting a higher priority on node `f2`, threads are guided to take the critical path task earlier, thus reducing overall execution time.

```
#include <iostream>
#include <cmath>

#include "oneapi/tbb/tick_count.h"
#include "oneapi/tbb/global_control.h"

#include "oneapi/tbb/flow_graph.h"
```

(continues on next page)

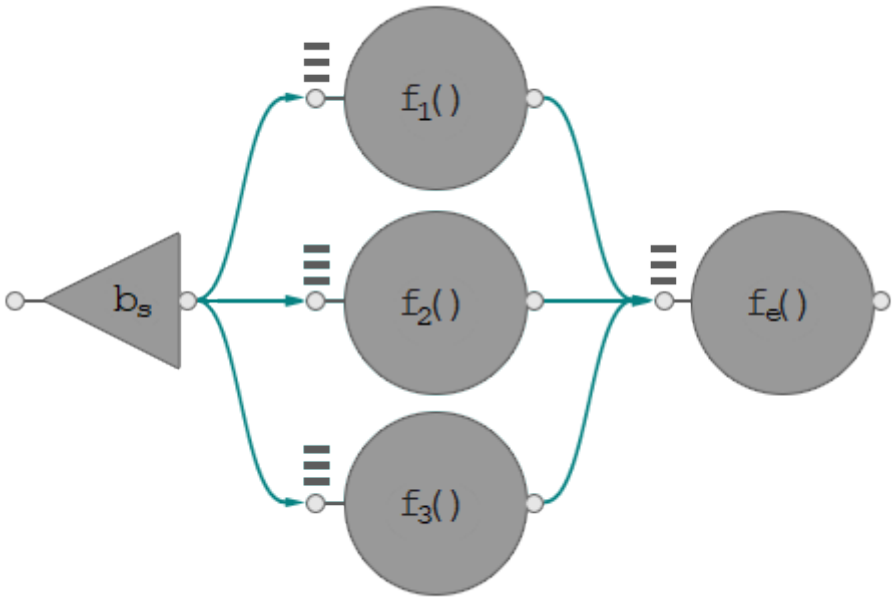


Fig. 1: Dependency flow graph with a critical path.

(continued from previous page)

```
void spin_for( double delta_seconds ) {
    oneapi::tbb::tick_count start = oneapi::tbb::tick_count::now();
    while( (oneapi::tbb::tick_count::now() - start).seconds() < delta_seconds ) ;
}

static const double unit_of_time = 0.1;

struct Body {
    unsigned factor;
    Body( unsigned times ) : factor( times ) {}
    void operator()( const oneapi::tbb::flow::continue_msg& ) {
        // body execution takes 'factor' units of time
        spin_for( factor * unit_of_time );
    }
};

int main() {
    using namespace oneapi::tbb::flow;

    const int max_threads = 2;
     oneapi::tbb::global_control control(oneapi::tbb::global_control::max_allowed_
↪parallelism, max_threads);

    graph g;

    broadcast_node<continue_msg> bs(g);
```

(continues on next page)

(continued from previous page)

```
    continue_node<continue_msg> f1(g, Body(5));

    // f2 is a heavy one and takes the most execution time as compared to the other␣
↪nodes in the
    // graph. Therefore, let the graph start this node as soon as possible by␣
↪prioritizing it over
    // the other nodes.
    continue_node<continue_msg> f2(g, Body(10), node_priority_t(1));

    continue_node<continue_msg> f3(g, Body(5));

    continue_node<continue_msg> fe(g, Body(7));

    make_edge( bs, f1 );
    make_edge( bs, f2 );
    make_edge( bs, f3 );

    make_edge( f1, fe );
    make_edge( f2, fe );
    make_edge( f3, fe );

    oneapi::tbb::tick_count start = oneapi::tbb::tick_count::now();

    bs.try_put( continue_msg() );
    g.wait_for_all();

    double elapsed = std::floor((oneapi::tbb::tick_count::now() - start).seconds() /␣
↪unit_of_time);

    std::cout << "Elapsed approximately " << elapsed << " units of time" << std::endl;

    return 0;
}
```

Predefined Concurrency Limits

[flow_graph.concurrency_limits]

Predefined constants that can be used as `function_node`, `multifunction_node`, and `async_node` constructors arguments to define concurrency limit.

```
// Defined in header <oneapi/tbb/flow_graph.h>

namespace oneapi {
namespace tbb {
namespace flow {

    std::size_t unlimited = /*implementation-defined*/;
    std::size_t serial = /*implementation-defined*/;

} // namespace flow
```

(continues on next page)

(continued from previous page)

```
} // namespace tbb
} // namespace oneapi
```

`unlimited` concurrency allows an unlimited number of invocations of the body to execute concurrently.

`serial` concurrency allows only a single call of body to execute concurrently.

copy_body

[flow_graph.copy_body]

`copy_body` is a function template that returns a copy of the body function object from the following nodes:

- *continue_node*
- *function_node*
- *multifunction_node*
- *input_node*
- *async_node*

```
namespace oneapi {
namespace tbb {
namespace flow {

    // Defined in header <oneapi/tbb/flow_graph.h>

    template< typename Body, typename Node >
    Body copy_body( Node &n );

} // namespace flow
} // namespace tbb
} // namespace oneapi
```

Buffering Nodes

Buffering nodes are designed to accumulate input messages and pass them to successors in a predefined order, depending on the node type.

overwrite_node

[flow_graph.overwrite_node]

A node that is a buffer of a single item that can be overwritten.

```
// Defined in header <oneapi/tbb/flow_graph.h>

namespace oneapi {
namespace tbb {
namespace flow {
```

(continues on next page)

(continued from previous page)

```
    template<typename T>
    class overwrite_node : public graph_node, public receiver<T>, public sender<T> {
    public:
        explicit overwrite_node( graph &g );
        overwrite_node( const overwrite_node &src );
        ~overwrite_node();

        bool try_put( const T &v );
        bool try_get( T &v );

        bool is_valid( );
        void clear( );
    };

} // namespace flow
} // namespace tbb
} // namespace oneapi
```

Requirements:

- The type `T` must meet the *DefaultConstructible* requirements from [defaultconstructible] and *CopyAssignable* requirements from [copyassignable] ISO C++ Standard sections.

This type of node buffers a single item of type `T`. The value is initially invalid. Gets from the node are non-destructive.

`overwrite_node` is a `graph_node`, `receiver<T>` and `sender<T>`.

`overwrite_node` has a *buffering* and *broadcast-push* *properties*.

`overwrite_node` allows overwriting its single item buffer.

Member functions

explicit **`overwrite_node`**(*graph* &g)

Constructs an object of type `overwrite_node` that belongs to the graph `g` with an invalid internal buffer item.

`overwrite_node`(const *overwrite_node* &src)

Constructs an object of type `overwrite_node` that belongs to the graph `g` with an invalid internal buffer item. The buffered value and list of successors and predecessors are not copied from `src`.

`~overwrite_node`()

Destroys the overwrite_node.

bool **`try_put`**(const T &v)

Stores `v` in the internal single item buffer and calls `try_put(v)` on all successors.

Returns: `true`

bool **`try_get`**(T &v)

If the internal buffer is valid, assigns the value to `v`.

Returns:`true` if `v` is assigned to; `false`, otherwise.

bool **`is_valid`**()

Returns: `true` if the buffer holds a valid value; `false`, otherwise.

void **clear**()

Invalidates the value held in the buffer.

Examples

The example demonstrates `overwrite_node` as a single-value storage that might be updated. Data can be accessed with direct `try_get()` call.

```
#include "oneapi/tbb/flow_graph.h"

int main() {
    const int data_limit = 20;
    int count = 0;

    oneapi::tbb::flow::graph g;

    oneapi::tbb::flow::function_node< int, int > data_set_preparation(g,
        oneapi::tbb::flow::unlimited, []( int data ) {
            printf("Prepare large data set and keep it inside node storage\n");
            return data;
        });

    oneapi::tbb::flow::overwrite_node< int > overwrite_storage(g);

    oneapi::tbb::flow::input_node< int > data_generator(g,
        [&]( oneapi::tbb::flow_control& fc ) -> int {
            if ( count < data_limit ) {
                return ++count;
            }
            fc.stop();
            return {};
        });

    oneapi::tbb::flow::function_node< int > process(g, oneapi::tbb::flow::unlimited,
        [&]( const int& data) {
            int data_from_storage = 0;
            overwrite_storage.try_get(data_from_storage);
            printf("Data from a storage: %d\n", data_from_storage);
            printf("Data to process: %d\n", data);
        });

    oneapi::tbb::flow::make_edge(data_set_preparation, overwrite_storage);
    oneapi::tbb::flow::make_edge(data_generator, process);

    data_set_preparation.try_put(1);
    data_generator.activate();

    g.wait_for_all();

    return 0;
}
```

`overwrite_node` supports reserving `join_node` as its successor. See the example in *the example section of*

write_once_node.

write_once_node

[flow_graph.write_once_node]

A node that is a buffer of a single item that cannot be overwritten.

```
// Defined in header <oneapi/tbb/flow_graph.h>

namespace oneapi {
namespace tbb {
namespace flow {

    template< typename T >
    class write_once_node : public graph_node, public receiver<T>, public sender<T> {
    public:
        explicit write_once_node( graph &g );
        write_once_node( const write_once_node &src );
        ~write_once_node();

        bool try_put( const T &v );
        bool try_get( T &v );

        bool is_valid( );
        void clear( );
    };

} // namespace flow
} // namespace tbb
} // namespace oneapi
```

Requirements:

- The `T` type must meet the *DefaultConstructible* requirements from [defaultconstructible] and *CopyAssignable* requirements from [copyassignable] ISO C++ Standard sections.

This type of node buffers a single item of type `T`. The value is initially invalid. Gets from the node are non-destructive.

`write_once_node` is a `graph_node`, `receiver<T>` and `sender<T>`.

`write_once_node` has a *buffering* and *broadcast-push* *properties*.

`write_once_node` does not allow overwriting its single item buffer.

Member functions

explicit **`write_once_node`**(*graph* &g)

Constructs an object of type `write_once_node` that belongs to the graph `g`, with an invalid internal buffer item.

`write_once_node`(const *write_once_node* &src)

Constructs an object of type `write_once_node` with an invalid internal buffer item. The buffered value and list of successors is not copied from `src`.

~write_once_node()

Destroys the `write_once_node`.

bool **try_put**(const T &v)

Stores `v` in the internal single item buffer if it does not contain a valid value already. If a new value is set, the node broadcast it to all successors.

Returns: `true` for the first time after construction or a call to `clear()`; `false`, otherwise.

bool **try_get**(T &v)

If the internal buffer is valid, assigns the value to `v`.

Returns: `true` if `v` is assigned to; `false`, otherwise.

bool **is_valid**()

Returns: `true` if the buffer holds a valid value; `false`, otherwise.

void **clear**()

Invalidates the value held in the buffer.

Example

Usage scenario is similar to *overwrite_node* but an internal buffer can be updated only after `clear()` call. The following example shows the possibility to connect the node to a reserving `join_node`, avoiding direct calls to the `try_get()` method from the body of the successor node.

```
#include "oneapi/tbb/flow_graph.h"

typedef int data_type;

int main() {
    using namespace oneapi::tbb::flow;

    graph g;

    function_node<data_type, data_type> static_result_computer_n(
        g, serial,
        [&](const data_type& msg) {
            // compute the result using incoming message and pass it further, e.g.:
            data_type result = data_type((msg << 2 + 3) / 4);
            return result;
        });
    write_once_node<data_type> write_once_n(g); // for buffering once computed value

    buffer_node<data_type> buffer_n(g);
    join_node<tuple<data_type, data_type>, reserving> join_n(g);

    function_node<tuple<data_type, data_type>> consumer_n(
        g, unlimited,
        [&](const tuple<data_type, data_type>& arg) {
            // use the precomputed static result along with dynamic data
            data_type precomputed_result = get<0>(arg);
            data_type dynamic_data = get<1>(arg);
        });
```

(continues on next page)

(continued from previous page)

```
    make_edge(static_result_computer_n, write_once_n);
    make_edge(write_once_n, input_port<0>(join_n));
    make_edge(buffer_n, input_port<1>(join_n));
    make_edge(join_n, consumer_n);

    // do one-time calculation that will be reused many times further in the graph
    static_result_computer_n.try_put(1);

    for (int i = 0; i < 100; i++) {
        buffer_n.try_put(1);
    }

    g.wait_for_all();

    return 0;
}
```

buffer_node

[flow_graph.buffer_node]

A node that is an unbounded buffer of messages. Messages are forwarded in an arbitrary order.

```
// Defined in header <oneapi/tbb/flow_graph.h>

namespace oneapi {
namespace tbb {
namespace flow {

    template< typename T>
    class buffer_node : public graph_node, public receiver<T>, public sender<T> {
    public:
        explicit buffer_node( graph &g );
        buffer_node( const buffer_node &src );
        ~buffer_node();

        bool try_put( const T &v );
        bool try_get( T &v );
    };

} // namespace flow
} // namespace tbb
} // namespace oneapi
```

Requirements:

- The type `T` must meet the *CopyConstructible* requirements from [copyconstructible] and *CopyAssignable* requirements from [copyassignable] ISO C++ Standard sections.

`buffer_node` is a `graph_node`, `receiver<T>`, and `sender<T>`.

`buffer_node` has a *buffering* and *single-push* *properties*.

`buffer_node` forwards messages in an arbitrary order to a single successor in its successor set.

Member functions

explicit **`buffer_node`**(*graph* &g)

Constructs an empty `buffer_node` that belongs to the graph `g`.

explicit **`buffer_node`**(const *buffer_node* &src)

Constructs an empty `buffer_node` that belongs to the same graph `g` as `src`. Any intermediate state of `src`, including its links to predecessors and successors, is not copied.

bool **`try_put`**(const T &v)

Adds `v` to the set of items managed by the node, and tries forwarding it to a successor.

Returns: `true`

bool **`try_get`**(T &v)

Returns: `true` if an item can be removed from the node and assigned to `v`. Returns `false` if there is no non-reserved item currently in the node.

queue_node

[flow_graph.queue_node]

A node that forwards messages in a first-in first-out (FIFO) order.

```
// Defined in header <oneapi/tbb/flow_graph.h>

namespace oneapi {
namespace tbb {
namespace flow {

    template <typename T >
    class queue_node : public graph_node, public receiver<T>, public sender<T> {
    public:
        explicit queue_node( graph &g );
        queue_node( const queue_node &src );
        ~queue_node();

        bool try_put( const T &v );
        bool try_get( T &v );
    };

} // namespace flow
} // namespace tbb
} // namespace oneapi
```

Requirements:

- The type `T` must meet the *CopyConstructible* requirements from [copyconstructible] and *CopyAssignable* requirements from [copyassignable] ISO C++ Standard sections.

`queue_node` forwards messages in a FIFO order to a single successor in its successor set.

`queue_node` is a `graph_node`, `receiver` and `sender`.

queue_node has a *buffering* and *single-push* *properties*.

Member functions

explicit **queue_node**(*graph* &g)

Constructs an empty queue_node that belongs to the graph g.

queue_node(const *queue_node* &src)

Constructs an empty queue_node that belongs to the same graph g as src. Any intermediate state of src, including its links to predecessors and successors, is not copied.

bool **try_put**(const T &v)

Adds v to the set of items managed by the node, and tries forwarding the least recently added item to a successor.

Returns: true.

bool **try_get**(T &v)

Returns: true if an item can be taken from the node and assigned to v. Returns false if there is no item currently in the queue_node or if the node is reserved.

Example

Usage scenario is similar to *buffer_node* except that messages are passed in first-in first-out (FIFO) order.

priority_queue_node

[flow_graph.priority_queue_node]

A class template that forwards messages in a priority order.

```
// Defined in header <oneapi/tbb/flow_graph.h>

namespace oneapi {
namespace tbb {
namespace flow {

    template< typename T, typename Compare = std::less<T>>
    class priority_queue_node : public graph_node, public receiver<T>, public sender<T> {
    public:
        explicit priority_queue_node( graph &g );
        priority_queue_node( const priority_queue_node &src );
        ~priority_queue_node();

        bool try_put( const T &v );
        bool try_get( T &v );
    };

} // namespace flow
} // namespace tbb
} // namespace oneapi
```

Requirements:

- The type `T` must meet the *CopyConstructible* requirements from [copyconstructible] and *CopyAssignable* requirements from [copyassignable] ISO C++ Standard sections.
- The type `Compare` must meet the *Compare* type requirements from [alg.sorting] ISO C++ Standard section. If `Compare` instance throws an exception, then behavior is undefined.

The next message to be forwarded has the largest priority as determined by the `Compare` template argument.

`priority_queue_node` is a `graph_node`, `receiver<T>`, and `sender<T>`.

`priority_queue_node` has a *buffering* and *single-push* *properties*.

Member functions

explicit **`priority_queue_node`**(*graph* &g)

Constructs an empty `priority_queue_node` that belongs to the graph `g`.

`priority_queue_node`(const *priority_queue_node* &src)

Constructs an empty `priority_queue_node` that belongs to the same graph `g` as `src`. Any intermediate state of `src`, including its links to predecessors and successors, is not copied.

bool **`try_put`**(const T &v)

Adds `v` to the `priority_queue_node` and tries forwarding to a successor the item with the largest priority among all of the items that were added to the node and have not been yet forwarded to successors.

Returns: `true`

bool **`try_get`**(T &v)

Returns: `true` if a message is available in the node and the node is not currently reserved. Otherwise, returns `false`. If the node returns `true`, the message with the largest priority is copied to `v`.

Example

Usage scenario is similar to *sequencer_node* except that the `priority_queue_node` provides local order, passing the message with highest priority of all stored at the moment, while `sequencer_node` enforces global order and does not allow a "smaller priority" message to pass through before all preceding messages.

sequencer_node

[flow_graph.sequencer_node]

A node that forwards messages in a sequence order.

```
// Defined in header <oneapi/tbb/flow_graph.h>

namespace oneapi {
namespace tbb {
namespace flow {

    template< typename T >
    class sequencer_node : public graph_node, public receiver<T>, public sender<T> {
    public:
        template< typename Sequencer >
        sequencer_node( graph &g, const Sequencer &s );
```

(continues on next page)

(continued from previous page)

```
        sequencer_node( const sequencer_node &src );

        bool try_put( const T &v );
        bool try_get( output_type &v );
    };

} // namespace flow
} // namespace tbb
} // namespace oneapi
```

Requirements:

- The type `T` must meet the *CopyConstructible* requirements from [copyconstructible] and *CopyAssignable* requirements from [copyassignable] ISO C++ Standard sections.
- The type `Sequencer` must meet the *Sequencer requirements*. Since C++17, `Sequencer` may also be a pointer to a const member function in `T` that returns `size_t` or a pointer to a data member in `T` of type `size_t`. If `Sequencer` instance throws an exception, behavior is undefined.

`sequencer_node` forwards messages in a sequence order to a single successor in its successor set.

`sequencer_node` is a `graph_node`, `receiver<T>` and `sender<T>`.

Each item that passes through a `sequencer_node` is ordered by its sequencer order number. These sequence order numbers range from 0 to the largest integer representable by the `std::size_t` type. Sequencer order number of an item is determined by passing the item to a user-provided `Sequencer` function object.

Note: The `sequencer_node` rejects duplicate sequencer numbers.

Member functions

template<typename **Sequencer**>
sequencer_node(*graph* &g, const *Sequencer* &s)

Constructs an empty `sequencer_node` that belongs to the graph `g` and uses `s` to compute sequence numbers for items.

sequencer_node(const *sequencer_node* &src)

Constructs an empty `sequencer_node` that belongs to the same graph `g` as `src` and uses a copy of the `Sequencer s` used to construct `src`. The list of predecessors, the list of successors, and the messages inside are not copied.

> **Caution:** The new sequencer object is copy-constructed from a copy of the original sequencer object provided to `src` at its construction. Changes made to member variables in the `src` object do not affect the sequencer of the new `sequencer_node`.

bool **try_put**(const T &v)

Adds `v` to the `sequencer_node` and tries forwarding the next item in the sequence to a successor.

Returns: `true`

bool **try_get**(T &v)

Returns: `true` if the next item in the sequence is available in the `sequencer_node`. If so, it is removed from

the node and assigned to `v`. Returns `false` if the next item in sequencer order is not available or if the node is reserved.

Deduction Guides

```
template <typename Body>
sequencer_node(graph&, Body) -> input_node<std::decay_t<input_t<Body>>>;
```

Where:

- `input_t` is an alias to `Body` input argument type.

Example

The example demonstrates ordering capabilities of the `sequencer_node`. While being processed in parallel, the data is passed to the successor node in the exact same order it was read.

```
#include "oneapi/tbb/flow_graph.h"

struct Message {
    int id;
    int data;
};

int main() {
    oneapi::tbb::flow::graph g;

    // Due to parallelism the node can push messages to its successors in any order
    oneapi::tbb::flow::function_node< Message, Message > process(g,␣
↪oneapi::tbb::flow::unlimited, [] (Message msg) -> Message {
        msg.data++;
        return msg;
    });

    oneapi::tbb::flow::sequencer_node< Message > ordering(g, [](const Message& msg) ->␣
↪int {
        return msg.id;
    });

    oneapi::tbb::flow::function_node< Message > writer(g, oneapi::tbb::flow::serial, []␣
↪(const Message& msg) {
        printf("Message recieved with id: %d\n", msg.id);
    });

    oneapi::tbb::flow::make_edge(process, ordering);
    oneapi::tbb::flow::make_edge(ordering, writer);

    for (int i = 0; i < 100; ++i) {
        Message msg = { i, 0 };
        process.try_put(msg);
    }
```

(continues on next page)

(continued from previous page)

```
    g.wait_for_all();
}
```

Service Nodes

These nodes are designed for advanced control of the message flow, such as combining messages from different paths in a graph or limiting the number of simultaneously processed messages, as well as for creating reusable custom nodes.

limiter_node

[flow_graph.limiter_node]

A node that counts and limits the number of messages that pass through it.

```
// Defined in header <oneapi/tbb/flow_graph.h>

namespace oneapi {
namespace tbb {
namespace flow {

    template< typename T, typename DecrementType=continue_msg >
    class limiter_node : public graph_node, public receiver<T>, public sender<T> {
    public:
        limiter_node( graph &g, size_t threshold );
        limiter_node( const limiter_node &src );

        receiver<DecrementType>& decrementer();

        bool try_put( const T &v );
        bool try_get( T &v );
    };

} // namespace flow
} // namespace tbb
} // namespace oneapi
```

Requirements:

- `T` type must meet the *DefaultConstructible* requirements from [defaultconstructible] ISO C++ Standard section.
- The `DecrementType` type must be an integral type or `continue_msg`.

`limiter_node` is a `graph_node`, `receiver<T>`, and `sender<T>`

`limiter_node` has a *discarding* and *broadcast-push properties*.

This node does not accept new messages once the user-specified `threshold` is reached. The internal counter of broadcasts is adjusted through use of the *decrementer*, a `receiver` object embedded into the node that can be obtained by calling the `decrementer` method. The counter values are truncated to be inside the [0, `threshold`] interval.

The template parameter `DecrementType` specifies the type of the message that can be sent to the decrementer. This template parameter is defined to `continue_msg` by default. If an integral type is specified, positive values sent to the decrementer determine the value by which the internal counter of broadcasts will be decreased, while negative values determine the value by which the internal counter of broadcasts will be increased.

If `continue_msg` is used as an argument for the `DecrementType` template parameter, the `decrementer`'s port of the `limiter_node` also acquires the behavior of the `continue_node`. This behavior requires the number of messages sent to it to be equal to the number of connected predecessors before decrementing the internal counter of broadcasts by one.

When `try_put` call on the decrementer results in the new value of the counter of broadcasts to be less than the `threshold`, the `limiter_node` tries to get a message from one of its known predecessors and forward that message to all its successors. If it cannot obtain a message from a predecessor, it decrements the counter of broadcasts.

Member functions

limiter_node(*graph* &g, size_t threshold)

Constructs a `limiter_node` that allows up to `threshold` items to pass through before rejecting `try_put`'s.

limiter_node(const *limiter_node* &src)

Constructs a `limiter_node` that has the same initial state that `src` had at its construction. The new `limiter_node` belongs to the same graph `g` as `src`, has the same `threshold`. The list of predecessors, the list of successors, and the current count of broadcasts are not copied from `src`.

receiver<DecrementType> &**decrementer**()

Obtains a reference to the embedded `receiver` object that is used for the internal counter adjustments.

bool **try_put**(const T &v)

If the broadcast count is below the threshold, `v` is broadcast to all successors.

Returns: `true` if `v` is broadcast; `false` if `v` is not broadcast because the threshold has been reached.

bool **try_get**(T &v)

Returns: `false`.

broadcast_node

[flow_graph.broadcast_node]

A node that broadcasts incoming messages to all of its successors.

```
// Defined in header <oneapi/tbb/flow_graph.h>

namespace oneapi {
namespace tbb {
namespace flow {

    template< typename T >
    class broadcast_node :
    public graph_node, public receiver<T>, public sender<T> {
    public:
        explicit broadcast_node( graph &g );
        broadcast_node( const broadcast_node &src );

        bool try_put( const T &v );
        bool try_get( T &v );
    };

} // namespace flow
```

(continues on next page)

(continued from previous page)

```
} // namespace tbb
} //namespace oneapi
```

`broadcast_node` is a `graph_node`, `receiver<T>`, and `sender<T>`.

`broadcast_node` has a *discarding* and *broadcast-push* *properties*.

All messages are forwarded immediately to all successors.

Member functions

explicit **`broadcast_node`**(*graph* &g)

Constructs an object of type `broadcast_node` that belongs to the graph `g`.

`broadcast_node`(const *broadcast_node* &src)

Constructs an object of type `broadcast_node` that belongs to the same graph `g` as `src`. The list of predecessors and the list of successors are not copied.

bool **`try_put`**(const input_type &v)

Broadcasts `v` to all successors.

Returns: always returns `true`, even if it was unable to successfully forward the message to any of its successors.

bool **`try_get`**(output_type &v)

Returns: `false`.

join_node

[flow_graph.join_node]

A node that creates a tuple from a set of messages received at its input ports and broadcasts the tuple to all of its successors.

```
// Defined in header <oneapi/tbb/flow_graph.h>

namespace oneapi {
namespace tbb {
namespace flow {
    using tag_value = /*implementation-specific*/;

    template<typename OutputTuple, class JoinPolicy = /*implementation-defined*/>
    class join_node : public graph_node, public sender< OutputTuple > {
    public:
        using input_ports_type = /*implementation-defined*/;

        explicit join_node( graph &g );
        join_node( const join_node &src );

        input_ports_type &input_ports( );

        bool try_get( OutputTuple &v );
    };
```

(continues on next page)

(continued from previous page)

```
    template<typename OutputTuple, typename K, class KHash=tbb_hash_compare<K> >
    class join_node< OutputTuple, key_matching<K,KHash> > : public graph_node, public␣
↪sender< OutputTuple > {
    public:
        using input_ports_type = /*implementation-defined*/;

        explicit join_node( graph &g );
        join_node( const join_node &src );

        template<typename B0, typename B1>
        join_node( graph &g, B0 b0, B1 b1 );
        template<typename B0, typename B1, typename B2>
        join_node( graph &g, B0 b0, B1 b1, B2 b2 );
        template<typename B0, typename B1, typename B2, typename B3>
        join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3 );
        template<typename B0, typename B1, typename B2, typename B3, typename B4>
        join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4 );
        template<typename B0, typename B1, typename B2, typename B3, typename B5>
        join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5 );
        template<typename B0, typename B1, typename B2, typename B3, typename B5,␣
↪typename B6>
        join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6 );
        template<typename B0, typename B1, typename B2, typename B3, typename B5,␣
↪typename B6, typename B6>
        join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6, B7 b7 );
        template<typename B0, typename B1, typename B2, typename B3, typename B5,␣
↪typename B6, typename B7>
        join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6, B7 b7 );
        template<typename B0, typename B1, typename B2, typename B3, typename B5,␣
↪typename B6, typename B7, typename B8>
        join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6, B7 b7, B8␣
↪b8 );
        template<typename B0, typename B1, typename B2, typename B3, typename B5,␣
↪typename B6, typename B7, typename B8, typename B9>
        join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6, B7 b7, B8␣
↪b8, B9 b9 );

        input_ports_type &input_ports( );

        bool try_get( OutputTuple &v );
    };

} // namespace flow
} // namespace tbb
} // namespace oneapi
```

Requirements:

- The type `OutputTuple` must be an instantiation of `std::tuple`. Each type that the tuple stores must meet the *DefaultConstructible* requirements from [defaultconstructible], *CopyConstructible* requirements from [copyconstructible] and *CopyAssignable* requirements from [copyassignable] ISO C++ Standard sections.
- The `JoinPolicy` type must be specified as one of *buffering policies* for `join_node`.

- The KHash type must meet the *HashCompare requirements*.
- The Bi types must meet the *JoinNodeFunctionObject requirements*. Since C++17, each of Bi types may also be a pointer to a const member function in Input that returns Key or a pointer to a data member of type Key in Input.

A join_node is a graph_node and a sender<OutputTuple>. It contains a tuple of input ports, each of which is a receiver<Type> for each *Type* in OutputTuple. It supports multiple input receivers with distinct types and broadcasts a tuple of received messages to all of its successors. All input ports of a join_node must use the same buffering policy.

The behavior of a join_node is based on its buffering policy.

join_node Policies

[flow_graph.join_node_policies]

join_node supports three buffering policies at its input ports: reserving, queueing, and key_matching.

```
// Defined in header <oneapi/tbb/flow_graph.h>

namespace oneapi {
namespace tbb {
namespace flow {

    struct reserving;
    struct queueing;
    template<typename K, class KHash=tbb_hash_compare<K> > struct key_matching;
    using tag_matching = key_matching<tag_value>;

} // namespace flow
} // namespace tbb
} // namespace oneapi
```

- queueing - As each input port is put to, the incoming message is added to an unbounded first-in first-out queue in the port. When there is at least one message at each input port, the join_node broadcasts a tuple containing the head of each queue to all successors. If at least one successor accepts the tuple, the head of each input port's queue is removed; otherwise, the messages remain in their respective input port queues.
- reserving - As each input port is put to, the join_node marks that an input may be available at that port and returns false. When all ports have been marked as possibly available, the join_node tries to reserve a message at each port from their known predecessors. If it is unable to reserve a message at a port, it unmarks that port, and releases all previously acquired reservations. If it is able to reserve a message at all ports, it broadcasts a tuple containing these messages to all successors. If at least one successor accepts the tuple, the reservations are consumed; otherwise, they are released.
- key_matching<typename K, class KHash=tbb_hash_compare<K>> - As each input port is put to, a user-provided function object is applied to the message to obtain its key. The message is then added to a hash table of the input port. When there is a message at each input port for a given key, the join_node removes all matching messages from the input ports, constructs a tuple containing the matching messages and attempts to broadcast it to all successors. If no successor accepts the tuple, it is saved and will be forwarded on a subsequent try_get.
- tag_matching - A specialization of key_matching that accepts keys of type tag_value.

The function template *input_port* simplifies the syntax for getting a reference to a specific input port.

join_node has a *buffering* and *broadcast-push* *properties*.

Member types

`input_ports_type` is an alias to a tuple of input ports.

Member functions

```
explicit join_node( graph &g );
```

Constructs an empty `join_node` that belongs to the graph `g`.

```
template<typename B0, typename B1>
join_node( graph &g, B0 b0, B1 b1 );
template<typename B0, typename B1, typename B2>
join_node( graph &g, B0 b0, B1 b1, B2 b2 );
template<typename B0, typename B1, , typename B2, typename B3>
join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3 );
template<typename B0, typename B1, , typename B2, typename B3, typename B4>
join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4 );
template<typename B0, typename B1, , typename B2, typename B3, typename B5>
join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5 );
template<typename B0, typename B1, , typename B2, typename B3, typename B5, typename B6>
join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6 );
template<typename B0, typename B1, , typename B2, typename B3, typename B5, typename B6,␣
↪typename B6>
join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6, B7 b7 );
template<typename B0, typename B1, , typename B2, typename B3, typename B5, typename B6,␣
↪typename B7>
join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6, B7 b7 );
template<typename B0, typename B1, , typename B2, typename B3, typename B5, typename B6,␣
↪typename B7, typename B8>
join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6, B7 b7, B8 b8 );
template<typename B0, typename B1, , typename B2, typename B3, typename B5, typename B6,␣
↪typename B7, typename B8, typename B9>
join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6, B7 b7, B8 b8, B9␣
↪b9 );
```

A constructor only available in the `key_matching` specialization of `join_node`.

Creates a `join_node` that uses the function objects `b0`, `b1`, … , `bN` to determine the tags for the input ports `0` through `N`.

> **Caution:** Function objects passed to the join_node constructor must not throw. They are called in parallel, and should be pure, take minimal time, and be non-blocking.

```
join_node( const join_node &src )
```

Creates a `join_node` that has the same initial state that `src` had at its construction. The list of predecessors, messages in the input ports, and successors are not copied.

```
input_ports_type &input_ports( )
```

Returns: a `std::tuple` of receivers. Each element inherits values from `receiver<T>`, where `T` is the type of message expected at that input. Each tuple element can be used like any other `receiver<T>`. The behavior of the ports is based on the selected `join_node` policy.

```
bool try_get( output_type &v )
```

Attempts to generate a tuple based on the buffering policy of the `join_node`.

If it can successfully generate a tuple, it copies it to `v` and returns `true`. Otherwise, it returns `false`.

Non-Member Types

```
using tag_value = /*implementation-specific*/;
```

`tag_value` is an unsigned integral type for defining the `tag_matching` policy.

Deduction Guides

```
template <typename Body, typename... Bodies>
join_node(graph&, Body, Bodies...)
    ->join_node<std::tuple<std::decay_t<input_t<Body>>, std::decay_t<input_t<Bodies>>...>
↪, key_matching<output_t<Body>>>;
```

Where:

- `input_t` is an alias to the input argument type of the passed function object.
- `output_t` is an alias to the return type of the passed function object.

split_node

[flow_graph.split_node]

A `split_node` sends each element of the incoming `std::tuple` to the output port that matches the element index in the incoming tuple.

```
// Defined in header <oneapi/tbb/flow_graph.h>

namespace oneapi {
namespace tbb {
namespace flow {

    template < typename TupleType >
    class split_node : public graph_node, public receiver<TupleType> {
    public:
        explicit split_node( graph &g );
        split_node( const split_node &other );
        ~split_node();
```

(continues on next page)

(continued from previous page)

```
        bool try_put( const TupleType &v );

        using output_ports_type = /*implementation-defined*/ ;
        output_ports_type& output_ports();
    };

} // namespace flow
} // namespace tbb
} // namespace oneapi
```

Requirements:

- The type `TupleType` must be an instantiation of `std::tuple`.

`split_node` is a `receiver<TupleType>` and has a tuple of `sender` output ports. Each of output ports is specified by corresponding tuple element type. This node receives a tuple at its single input port and generates a message from each element of the tuple, passing each to the corresponding output port.

`split_node` has a *discarding* and *broadcast-push* *properties*.

`split_node` has unlimited concurrency, and behaves as a `broadcast_node` with multiple output ports.

Member functions

explicit **`split_node`**(*graph* &g)

Constructs a `split_node` registered with graph `g`.

`split_node`(const *split_node* &other)

Constructs a `split_node` that has the same initial state that `other` had when it was constructed. The `split_node` that is constructed has a reference to the same `graph` object as `other`. The predecessors and successors of `other` are not copied.

`~split_node`()

Destructor

bool **`try_put`**(const TupleType &v)

Broadcasts each element of the incoming tuple to the nodes connected to the `split_node` output ports. The element at index `i` of `v` will be broadcast through the $\texttt{i}^{\text{th}}$ output port.

Returns: `true`

output_ports_type &**`output_ports`**()

Returns: a `std::tuple` of output ports.

indexer_node

[flow_graph.indexer_node]

`indexer_node` broadcasts messages received at input ports to all of its successors. The messages are broadcast individually as they are received at each port. The output is a *tagged message* that contains a tag and a value; the tag identifies the input port on which the message was received.

```
// Defined in header <oneapi/tbb/flow_graph.h>

namespace oneapi {
namespace tbb {
namespace flow {

    template<typename T0, typename... TN>
    class indexer_node : public graph_node, public sender</*implementation_defined*/> {
    public:
        indexer_node(graph &g);
        indexer_node(const indexer_node &src);

        using input_ports_type = /*implementation_defined*/;
        input_ports_type &input_ports();

        using output_type = tagged_msg<size_t, T0, TN...>;
        bool try_get( output_type &v );
    };

} // namespace flow
} // namespace tbb
} // namespace oneapi
```

Requirements:

- The `T0` type and all types in `TN` template parameter pack must meet the *CopyConstructible* requirements from [copyconstructible] ISO C++ Standard section.

An `indexer_node` is a `graph_node` and `sender<tagged_msg<size_t, T0, TN...>>`. It contains a tuple of input ports, each of which is a `receiver` specified by corresponding input template parameter pack element. It supports multiple input receivers with distinct types and broadcasts each received message to all of its successors. Unlike a `join_node`, each message is broadcast individually to all successors of the `indexer_node` as it arrives at an input port. Before broadcasting, a message is tagged with the index of the port on which the message arrived.

`indexer_node` has a *discarding* and *broadcast-push* *properties*.

The function template *input_port* simplifies the syntax for getting a reference to a specific input port.

Member types

- `input_ports_type` is an alias to a `std::tuple` of input ports.
- `output_type` is an alias to the message of type `tagged_msg`, which is sent to successors.

Member functions

`indexer_node`(*graph* &g)

Constructs an `indexer_node` that belongs to the graph `g`.

`indexer_node`(const *indexer_node* &src)

Constructs an `indexer_node`. The list of predecessors, messages in the input ports, and successors are not copied.

input_ports_type &**`input_ports`**()

Returns: A `std::tuple` of receivers. Each element inherits from `receiver<T>` where `T` is the type of message expected at that input. Each tuple element can be used like any other `receiver<T>`.

bool **`try_get`**(output_type &v)

An `indexer_node` contains no buffering and therefore does not support gets.

Returns: `false`.

See also:

- *input_port function template*
- *tagged_msg template class*

composite_node

[flow_graph.composite_node]

A node that encapsulates a collection of other nodes as a first class graph node.

```
// Defined in header <oneapi/tbb/flow_graph.h>

namespace oneapi {
namespace tbb {
namespace flow {

    template< typename InputTuple, typename OutputTuple > class composite_node;

    // composite_node with both input ports and output ports
    template< typename... InputTypes, typename... OutputTypes>
    class composite_node <std::tuple<InputTypes...>, std::tuple<OutputTypes...> > :␣
↪public graph_node {
    public:
        typedef std::tuple< receiver<InputTypes>&... > input_ports_type;
        typedef std::tuple< sender<OutputTypes>&... > output_ports_type;

        composite_node( graph &g );
        virtual ~composite_node();
```

(continues on next page)

(continued from previous page)

```
        void set_external_ports(input_ports_type&& input_ports_tuple, output_ports_type&&
↪ output_ports_tuple);
        input_ports_type& input_ports();
        output_ports_type& output_ports();
    };

    // composite_node with only input ports
    template< typename... InputTypes>
    class composite_node <std::tuple<InputTypes...>, std::tuple<> > : public graph_node{
    public:
        typedef std::tuple< receiver<InputTypes>&... > input_ports_type;

        composite_node( graph &g );
        virtual ~composite_node();

        void set_external_ports(input_ports_type&& input_ports_tuple);
        input_ports_type& input_ports();
    };

    // composite_nodes with only output_ports
    template<typename... OutputTypes>
    class composite_node <std::tuple<>, std::tuple<OutputTypes...> > : public graph_node{
    public:
        typedef std::tuple< sender<OutputTypes>&... > output_ports_type;

        composite_node( graph &g );
        virtual ~composite_node();

        void set_external_ports(output_ports_type&& output_ports_tuple);
        output_ports_type& output_ports();
    };

} // namespace flow
} // namespace tbb
} // namespace oneapi
```

- The `InputTuple` and `OutputTuple` must be instantiations of `std::tuple`.

`composite_node` is a `graph_node`, `receiver<T>`, and `sender<T>`.

The `composite_node` can package any number of other nodes. It maintains input and output port references to nodes in the package that border the `composite_node`. This allows the references to be used to make edges to other nodes outside of the `composite_node`. The `InputTuple` is a tuple of input types. The `composite_node` has an input port for each type in `InputTuple`. Likewise, the `OutputTuple` is a tuple of output types. The `composite_node` has an output port for each type in `OutputTuple`.

The composite_node is a multi-port node with three specializations.

- **A multi-port node with multi-input ports and multi-output ports:** This specialization has a tuple of input ports, each of which is a `receiver` of a type in `InputTuple`. Each input port is a reference to a port of a node that the `composite_node` encapsulates. Similarly, this specialization also has a tuple of output ports, each of which is a `sender` of a type in `OutputTuple`. Each output port is a reference to a port of a node that the `composite_node` encapsulates.
- **A multi-port node with only input ports and no output ports:** This specialization only has a tuple of input

ports.

- **A multi-port node with only output ports and no input_ports:** This specialization only has a tuple of output ports.

The function template *input_port* can be used to get a reference to a specific input port and the function template *output_port* can be used to get a reference to a specific output port.

Construction of a `composite_node` is done in two stages:

- Defining the `composite_node` with specification of `InputTuple` and `OutputTuple`.
- Making aliases from the encapsulated nodes that border the `composite_node` to the input and output ports of the `composite_node`. This step is mandatory as without it the `composite_node` input and output ports are not bound to any actual nodes. Making the aliases is achieved by calling the method `set_external_ports`.

The composite_node does not meet the *CopyConstructible* requirements from [copyconstructible] ISO C++ Standard section.

Member functions

`composite_node`(*graph* &g)

Constructs a `composite_node` that belongs to the graph `g`.

void **`set_external_ports`**(input_ports_type &&input_ports_tuple, output_ports_type &&output_ports_tuple)

Creates input and output ports of the `composite_node` as aliases to the ports referenced by `input_ports_tuple` and `output_ports_tuple`, respectively. That is, a port referenced at position `N` in `input_ports_tuple` is mapped as the `N`th input port of the `composite_node`, similarly for output ports.

input_ports_type &**`input_ports`**()

Returns: A `std::tuple` of `receivers`. Each element is a reference to the actual node or input port that was aliased to that position in `set_external_ports()`.

> **Caution:** Calling `input_ports()` without a prior call to `set_external_ports()` results in undefined behavior.

output_ports_type &**`output_ports`**()

Returns: A `std::tuple` of `senders`. Each element is a reference to the actual node or output port that was aliased to that position in `set_external_ports()`.

> **Caution:** Calling `output_ports()` without a prior call to `set_external_ports()` results in undefined behavior.

See also:

- *input_port function template*
- *output_port function template*

Ports and Edges

Flow Graph provides an API to manage connections between the nodes. For nodes that have more than one input or output ports (for example, `join_node`), making a connection requires to specify a certain port by using special helper functions.

input_port

[flow_graph.input_port]

A template function that returns a reference to a specific input port of a given `join_node`, `indexer_node` or `composite_node`.

```
// Defined in header <oneapi/tbb/flow_graph.h>

namespace oneapi {
namespace tbb {
namespace flow {

    template<size_t N, typename NodeType>
    /*implementation-defined*/& input_port(NodeType &n);

} // namespace flow
} // namespace tbb
} // namespace oneapi
```

See also:

- *join_node template class*
- *indexer_node template class*
- *composite_node template class*

output_port

[flow_graph.output_port]

A template function that returns a reference to a specific output port of a given `split_node`, `indexer_node`, or `composite_node`.

```
// Defined in header <oneapi/tbb/flow_graph.h>

namespace oneapi {
namespace tbb {
namespace flow {

    template<size_t N, typename NodeType>
    /*implementation-defined*/& output_port(NodeType &n);

} // namespace flow
} // namespace tbb
} // namespace oneapi
```

See also:

- *split_node Template Class*
- *multifunction_node Template Class*
- *composite_node Template Class*

make_edge

[flow_graph.make_edge]

A function template for building edges between nodes.

```
// Defined in header <oneapi/tbb/flow_graph.h>

namespace oneapi {
namespace tbb {
namespace flow {

    template<typename Message>
    inline void make_edge( sender<Message> &p, receiver<Message> &s );

    template< typename MultiOutputNode, typename MultiInputNode >
    inline void make_edge( MultiOutputNode& output, MultiInputNode& input );

    template<typename MultiOutputNode, typename Message>
    inline void make_edge( MultiOutputNode& output, receiver<Message> input );

    template<typename Message, typename MultiInputNode>
    inline void make_edge( sender<Message> output, MultiInputNode& input );

} // namespace flow
} // namespace tbb
} // namespace oneapi
```

Requirements:

- The *MultiOutputNode* type must have a valid `MultiOutputNode::output_ports_type` qualified-id that denotes a type.
- The *MultiInputNode* type must have a valid `MultiInputNode::input_ports_type` qualified-id that denotes a type.

The common form of `make_edge(sender, receiver)` creates an edge between provided `sender` and `receiver` instances.

Overloads that accept a *MultiOutputNode* type instance make an edge from port `0` of a multi-output predecessor.

Overloads that accept a *MultiInputNode* type instance make an edge to port `0` of a multi-input successor.

remove_edge

[flow_graph.remove_edge]

A function template for building edges between nodes.

```
// Defined in header <oneapi/tbb/flow_graph.h>

namespace oneapi {
namespace tbb {
namespace flow {

    template<typename Message>
    inline void remove_edge( sender<Message> &p, receiver<Message> &s );

    template< typename MultiOutputNode, typename MultiInputNode >
    inline void remove_edge( MultiOutputNode& output, MultiInputNode& input );

    template<typename MultiOutputNode, typename Message>
    inline void remove_edge( MultiOutputNode& output, receiver<Message> input );

    template<typename Message, typename MultiInputNode>
    inline void remove_edge( sender<Message> output, MultiInputNode& input );

} // namespace flow
} // namespace tbb
} // namespace oneapi
```

Requirements:

- The *MultiOutputNode* type must have a valid `MultiOutputNode::output_ports_type` qualified-id that denotes a type.
- The *MultiInputNode* type must have a valid `MultiInputNode::input_ports_type` qualified-id that denotes a type.

The common form of `remove_edge(sender, receiver)` creates an edge between provided `sender` and `receiver` instances.

Overloads that accept a *MultiOutputNode* type instance remove an edge from port `0` of a multi-output predecessor.

Overloads that accept a *MultiInputNode* type instance remove an edge to port `0` of a multi-input successor.

Special Messages Types

Flow Graph supports a set of specific message types.

continue_msg

[flow_graph.continue_msg]

An empty class that represents a continue message. An object of this class is used to indicate that the sender has completed.

```
// Defined in header <oneapi/tbb/flow_graph.h>

namespace oneapi {
namespace tbb {
namespace flow {

    class continue_msg {};

} // namespace flow
} // namespace tbb
} // namespace oneapi
```

tagged_msg

[flow_graph.tagged_msg]

A class template composed of a tag and a message. The message is a value that can be one of several defined types.

```
// Defined in header <oneapi/tbb/flow_graph.h>

namespace oneapi {
namespace tbb {
namespace flow {

    template<typename TagType, typename... TN>
    class tagged_msg {
    public:
        template<typename T, typename R>
        tagged_msg(T const &index, R const &val);

        TagType tag() const;

        template<typename V>
        const V& cast_to() const;

        template<typename V>
        bool is_a() const;

    };

} // namespace flow
```

(continues on next page)

(continued from previous page)

```
} // namespace tbb
} // namespace oneapi
```

Requirements:

- All types in `TN` template parameter pack must meet the *CopyConstructible* requirements from [copyconstructible] ISO C++ Standard section.
- The type *TagType* must be an integral unsigned type.

The `tagged_msg` class template is intended for messages whose type is determined at runtime. A message of one of the types `TN` is tagged with a tag of type `TagType`. The tag then can serve to identify the message. In the flow graph, `tagged_msg` is used as the output of *indexer_node*.

Member functions

template<typename **T**, typename **R**>
tagged_msg(*T* const &index, *R* const &value)

Requirements:

- The type *R* must be the same as one of the `TN` types.
- The type *T* must be acceptable as a `TagType` constructor parameter.

Constructs a `tagged_msg` with tag `index` and value `val`.

TagType **tag**() const

Returns the current tag.

template<typename **V**>
const *V* &**cast_to**() const

Requirements:

- The type `V` must be the same as one of the `TN` types.

Returns the value stored in `tagged_msg`. If the value is not of type `V`, the `std::runtime_error` exception is thrown.

template<typename **V**>
bool **is_a**() const

Requirements:

- The type `V` must be the same as one of the `TN` types.

Returns true if `V` is the type of the value held by the `tagged_msg`. Returns false, otherwise.

Non-member functions

```
template<typename V, typename T>
const V& cast_to(T const &t) {
    return t.cast_to<V>();
}

template<typename V, typename T>
bool is_a(T const &t);
```

Requirements:

- The type `T` must be an instantiated `tagged_msg` class template.
- The type `V` must be the same as one of the corresponding template arguments for `tagged_msg`.

The free-standing template functions `cast_to` and `is_a` applied to a `tagged_msg` object are equivalent to the calls of the corresponding methods of that object.

See also:

- *indexer_node class template*

Examples

Dependency Flow Graph Example

In the following example, five computations A-E are set up with the partial ordering shown below in "A simple dependency graph.". For each edge in the flow graph, the node at the tail of the edge must complete its execution before the node at the head may begin.

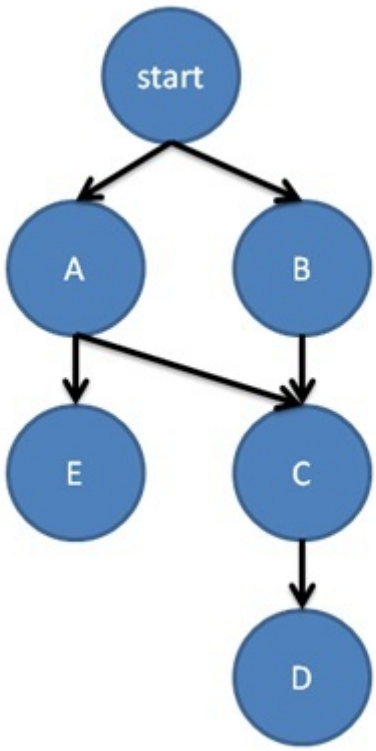


Fig. 2: A simple dependency graph.

```
#include <cstdio>
#include "oneapi/tbb/flow_graph.h"

using namespace oneapi::tbb::flow;
```

(continues on next page)

(continued from previous page)

```
struct body {
    std::string my_name;
    body(const char *name) : my_name(name) {}
    void operator()(continue_msg) const {
        printf("%s\n", my_name.c_str());
    }
};

int main() {
    graph g;

    broadcast_node< continue_msg > start(g);
    continue_node<continue_msg> a(g, body("A"));
    continue_node<continue_msg> b(g, body("B"));
    continue_node<continue_msg> c(g, body("C"));
    continue_node<continue_msg> d(g, body("D"));
    continue_node<continue_msg> e(g, body("E"));

    make_edge(start, a);
    make_edge(start, b);
    make_edge(a, c);
    make_edge(b, c);
    make_edge(c, d);
    make_edge(a, e);

    for (int i = 0; i < 3; ++i) {
        start.try_put(continue_msg());
        g.wait_for_all();
    }

    return 0;
}
```

In this example, nodes A-E print out their names. All of these nodes are therefore able to use `struct body` to construct their body objects.

In function `main`, the flow graph is set up once and then run three times. All of the nodes in this example pass around `continue_msg` objects. This type is used to communicate that a node has completed execution.

The first line in function `main` instantiates a `graph` object `g`. On the next line, a `broadcast_node` named `start` is created. Anything passed to this node will be broadcast to all of its successors. The node `start` is used in the `for` loop at the bottom of `main` to launch the execution of the rest of the flow graph.

In the example, five `continue_node` objects are created, named a - e. Each node is constructed with a reference to `graph g` and the function object to invoke when it runs. The successor / predecessor relationships are set up by the `make_edge` calls that follow the declaration of the nodes.

After the nodes and edges are set up, the `try_put` in each iteration of the `for` loop results in a broadcast of a `continue_msg` to both `a` and `b`. Both `a` and `b` are waiting for a single `continue_msg`, since they both have only a single predecessor, `start`.

When they receive the message from `start`, they execute their body objects. When complete, each of them forwards a message to a successor, and so on. The graph uses tasks to execute the node bodies as well as to forward messages between the nodes, allowing computation to execute concurrently when possible.

See also:

- *continue_msg class*
- *continue_node class*

Message Flow Graph Example

This example calculates the sum `x*x + x*x*x` for all `x = 1 to 10`. The layout of this example is shown in the figure below.

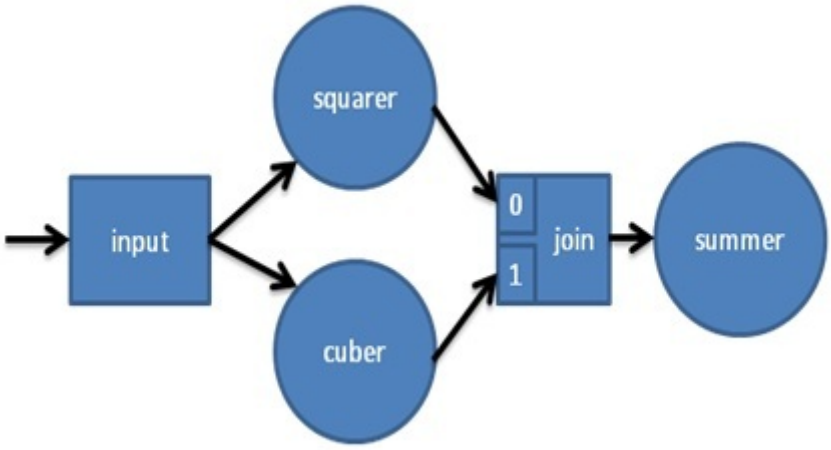


Fig. 3: A simple message flow graph.

Each value enters through the `broadcast_node<int> input`. This node broadcasts the value to both `squarer` and `cuber`, which calculate `x*x` and `x*x*x`, respectively. The output of each of these nodes is put to one of `join`'s ports. A tuple containing both values is created by `join_node<std::tuple<int,int>> join` and forwarded to `summer`, which adds both values to the running total. Both `squarer` and `cuber` allow unlimited concurrency, that is they each may process multiple values simultaneously. The final `summer`, which updates a shared total, is only allowed to process a single incoming tuple at a time, eliminating the need for a lock around the shared value.

```
#include <cstdio>
#include "oneapi/tbb/flow_graph.h"

using namespace oneapi::tbb::flow;

struct square {
  int operator()(int v) { return v*v; }
};

struct cube {
  int operator()(int v) { return v*v*v; }
};

class sum {
  int &my_sum;
public:
  sum( int &s ) : my_sum(s) {}
  int operator()( std::tuple<int, int> v ) {
```

(continues on next page)

(continued from previous page)

```
    my_sum += get<0>(v) + get<1>(v);
    return my_sum;
  }
};

int main() {
  int result = 0;

  graph g;
  broadcast_node<int> input(g);
  function_node<int,int> squarer( g, unlimited, square() );
  function_node<int,int> cuber( g, unlimited, cube() );
  join_node<std::tuple<int,int>, queueing> join( g );
  function_node<std::tuple<int,int>,int>
      summer( g, serial, sum(result) );

  make_edge( input, squarer );
  make_edge( input, cuber );
  make_edge( squarer, get<0>( join.input_ports() ) );
  make_edge( cuber, get<1>( join.input_ports() ) );
  make_edge( join, summer );

  for (int i = 1; i <= 10; ++i)
      input.try_put(i);
  g.wait_for_all();

  printf("Final result is %d\n", result);
  return 0;
}
```

In the example code above, the classes `square`, `cube`, and `sum` define the three user-defined operations. Each class is used to create a `function_node`.

In function `main`, the flow graph is set up and then the values 1-10 are put into the node `input`. All the nodes in this example pass around values of type `int`. The nodes used in this example are all class templates and therefore can be used with any type that supports copy construction, including pointers and objects.

8.2.4 Task Scheduler

[scheduler]

oneAPI Threading Building Blocks (oneTBB) provides a task scheduler, which is the engine that drives the algorithm templates and task groups. The exact tasking API depends on the implementation.

The tasks are quanta of computation. The scheduler implements worker thread pool and maps tasks onto these threads. The mapping is non-preemptive. Once a thread starts running a task, the task is bound to that thread until completion. During that time, the thread services other tasks only when it waits for completion of nested parallel constructs, as described below. While waiting, either user or worker thread may run any available task, including unrelated tasks created by this or other threads.

The task scheduler is intended for parallelizing computationally intensive work. Because task objects are not scheduled preemptively, they should generally avoid making calls that might block a thread for long periods during which the thread cannot service other tasks.

Caution: There is no guarantee that *potentially* parallel tasks *actually* execute in parallel, because the scheduler adjusts actual parallelism to fit available worker threads. For example, given a single worker thread, the scheduler creates no actual parallelism. For example, it is generally unsafe to use tasks in a producer consumer relationship, because there is no guarantee that the consumer runs at all while the producer is running.

Scheduling controls

task_group_context

[scheduler.task_group_context]

`task_group_context` represents a set of properties used by task scheduler for execution of the associated tasks. Each task is associated with only one `task_group_context` object.

The `task_group_context` objects form a forest of trees. Each tree's root is a `task_group_context` constructed as `isolated`.

`task_group_context` is cancelled explicitly by the user request, or implicitly when an exception is thrown out of an associated task. Canceling `task_group_context` causes the entire subtree rooted at it to be cancelled.

The `task_group_context` carries floating point settings inherited from the parent `task_group_context` object or captured with a dedicated interface.

```
// Defined in header <oneapi/tbb/task_group.h>

namespace oneapi {
namespace tbb {

    class task_group_context {
    public:
        enum kind_t {
            isolated = /* implementation-defined */,
            bound = /* implementation-defined */
        };
        enum traits_type {
            fp_settings = /* implementation-defined */,
            default_traits = 0
        };

        task_group_context( kind_t relation_with_parent = bound,
                            uintptr_t traits = default_traits );
        ~task_group_context();

        void reset();
        bool cancel_group_execution();
        bool is_group_execution_cancelled() const;
        void capture_fp_settings();
        uintptr_t traits() const;
    };

} // namespace tbb;
} // namespace oneapi
```

Member types and constants

enum `kind_t::isolated`

When passed to the specific constructor, the created `task_group_context` object has no parent.

enum `kind_t::bound`

When passed to the specific constructor, the created `task_group_context` object becomes a child of the innermost running task's group when the first task associated to the `task_group_context` is passed to the task scheduler. If there is no innermost running task on the current thread, the `task_group_context` becomes `isolated`.

enum `traits_type::fp_settings`

When passed to the specific constructor, the flag forces the context to capture floating-point settings from the current thread.

Member functions

`task_group_context`(kind_t relation_to_parent = bound, uintptr_t traits = default_traits)

Constructs an empty `task_group_context`.

`~task_group_context`()

Destroys an empty task_group_context. The behavior is undefined if there are still extant tasks associated with this `task_group_context`.

bool `cancel_group_execution`()

Requests that tasks associated with this `task_group_context` are not executed.

Returns `false` if this `task_group_context` is already cancelled; `true`, otherwise. If concurrently called by multiple threads, exactly one call returns `true` and the rest return `false`.

bool `is_group_execution_cancelled`() const

Returns `true` if this `task_group_context` has received the cancellation request.

void `reset`()

Reinitializes this `task_group_context` to the uncancelled state.

> **Caution:** This method is only safe to call once all tasks associated with the group's subordinate groups have completed. This method must not be invoked concurrently by multiple threads.

void `capture_fp_settings`()

Captures floating-point settings from the current thread.

> **Caution:** This method is only safe to call once all tasks associated with the group's subordinate groups have completed. This method must not be invoked concurrently by multiple threads.

uintptr_t `traits`() const

Returns traits of this `task_group_context`.

global_control

[scheduler.global_control]

Use this class to control certain settings or behavior of the oneTBB dynamic library.

An object of class `global_control`, or a "control variable", affects one of several behavioral aspects, or parameters, of TBB. The `global_control` class is primarily intended for use at the application level, to control the whole application behavior.

The current set of parameters that you can modify is defined by the `global_control::parameter` enumeration. The parameter and the value it should take are specified as arguments to the constructor of a control variable. The impact of the control variable ends when its lifetime is complete.

Control variables can be created in different threads, and may have nested or overlapping scopes. However, at any point in time each controlled parameter has a single active value that applies to the whole process. This value is selected from all currently existing control variables by applying a parameter-specific selection rule.

```
// Defined in header <oneapi/tbb/global_control.h>

namespace oneapi {
namespace tbb {
    class global_control {
    public:
        enum parameter {
            max_allowed_parallelism,
            thread_stack_size,
            terminate_on_exception
        };

        global_control(parameter p, size_t value);
        ~global_control();

        static size_t active_value(parameter param);
    };
} // namespace tbb
} // namespace oneapi
```

Member types and constants

enum `parameter::`**`max_allowed_parallelism`**

Selection rule: minimum

Limits total number of worker threads that can be active in the task scheduler to `parameter_value - 1`.

Note: With `max_allowed_parallelism` set to 1, `global_control` enforces serial execution of all tasks by the application thread(s), that is, the task scheduler does not allow worker threads to run. There is one exception: if some work is submitted for execution via `task_arena::enqueue`, a single worker thread will still run ignoring the `max_allowed_parallelism` restriction.

---

enum `parameter::`**`thread_stack_size`**

Selection rule: maximum

Set stack size for working threads created by the library.

enum `parameter::terminate_on_exception`

Selection rule: logical disjunction

Setting the parameter to 1 causes termination in any condition that would throw or rethrow an exception. If set to 0 (default), the parameter does not affect the implementation behavior.

Member functions

`global_control`(parameter param, size_t value)

Constructs a `global_control` object with a specified control parameter and it's value.

`~global_control`()

Destructs a control variable object and ends it's impact.

static size_t **`active_value`**(parameter param)

Returns the currently active value of the setting defined by `param`.

See also:

- *task_arena*

Resumable tasks

[scheduler.resumable_tasks]

Functions to suspend task execution at a specific point and signal to resume it later.

```
// Defined in header <oneapi/tbb/task.h>

using oneapi::tbb::task::suspend_point = /* implementation-defined */;
template < typename Func > void oneapi::tbb::task::suspend( Func );
void oneapi::tbb::task::resume( oneapi::tbb::task::suspend_point );
```

Requirements:

- The `Func` type must meet the *SuspendFunc requirements*.

The `oneapi::tbb::task::suspend` function called within a running task suspends execution of the task and switches the thread to participate in other oneTBB parallel work. This function accepts a user callable object with the current execution context `oneapi::tbb::task::suspend_point` as an argument. The user-specified callable object is executed by the calling thread.

The `oneapi::tbb::task::suspend_point` context tag must be passed to the `oneapi::tbb::task::resume` function to trigger a program execution at the suspended point. The `oneapi::tbb::task::resume` function can be called at any point of an application, even on a separate thread. In this regard, this function acts as a signal for the task scheduler.

Note: There are no guarantees that the same thread that called `oneapi::tbb::task::suspend` continues execution after the suspended point. However, these guarantees are provided for the outermost blocking oneTBB calls (such as `oneapi::tbb::parallel_for` and `oneapi::tbb::flow::graph::wait_for_all`) and `oneapi::tbb::task_arena::execute` calls.

Example

```
// Parallel computation region
oneapi::tbb::parallel_for(0, N, [&](int) {
    // Suspend the current task execution and capture the context
    oneapi::tbb::task::suspend([&] (oneapi::tbb::task::suspend_point tag) {
        // Dedicated user-managed activity that processes async requests.
        async_activity.submit(tag); // could be OpenCL/IO/Database/Network etc.
    }); // execution will be resumed after this function
});
```

```
// Dedicated user-managed activity:

// Signal to resume execution of the task referenced by the oneapi::tbb::task::suspend_
↪point
// from a dedicated user-managed activity
oneapi::tbb::task::resume(tag);
```

task_scheduler_handle

[scheduler.task_scheduler_handle]

The `oneapi::tbb::task_scheduler_handle` class and the `oneapi::tbb::finalize` function allow user to wait for completion of worker threads.

When the `oneapi::tbb::finalize` function is called with an `oneapi::tbb::task_scheduler_handle` instance, it blocks the calling thread until the completion of all worker threads that were implicitly created by the library.

```
// Defined in header <oneapi/tbb/global_control.h>

namespace oneapi {
    namespace tbb {

        class task_scheduler_handle {
        public:
            task_scheduler_handle() = default;
            task_scheduler_handle(oneapi::tbb::attach);
            ~task_scheduler_handle();

            task_scheduler_handle(const task_scheduler_handle& other) = delete;
            task_scheduler_handle(task_scheduler_handle&& other) noexcept;
            task_scheduler_handle& operator=(const task_scheduler_handle& other) =␣
↪delete;
            task_scheduler_handle& operator=(task_scheduler_handle&& other) noexcept;

            explicit operator bool() const noexcept;

            void release();
        };

        void finalize(task_scheduler_handle& handle);
        bool finalize(task_scheduler_handle& handle, const std::nothrow_t&) noexcept;
```

(continues on next page)

(continued from previous page)

```
    } // namespace tbb
} // namespace oneapi
```

Member Functions

task_scheduler_handle()

Effects: Creates an empty instance of the `task_scheduler_handle` class that does not contain any references to the task scheduler.

---

task_scheduler_handle(oneapi::tbb::attach)

Effects: Creates an instance of the `task_scheduler_handle` class that holds a reference to the task scheduler preventing its premature destruction.

~task_scheduler_handle()

Effects: Destroys an instance of the `task_scheduler_handle` class. If not empty, releases a reference to the task scheduler and deactivates an instance of the `task_scheduler_handle` class.

---

task_scheduler_handle(*task_scheduler_handle* &&other) noexcept

Effects: Creates an instance of the `task_scheduler_handle` class that references the task scheduler referenced by `other`. In turn, `other` releases its reference to the task scheduler.

---

task_scheduler_handle &**operator=**(*task_scheduler_handle* &&other) noexcept

Effects: If not empty, releases a reference to the task scheduler referenced by `this`. Adds a reference to the task scheduler referenced by `other`. In turn, `other` releases its reference to the task scheduler. **Returns**: A reference to `*this`.

---

explicit **operator bool**() const noexcept

Returns: `true` if `this` is not empty and refers to some task scheduler; `false` otherwise.

---

void **release**()

Effects: If not empty, releases a reference to the task scheduler and deactivates an instance of the `task_scheduler_handle` class; otherwise, does nothing. Non-blocking method.

Non-member Functions

void **finalize**(*task_scheduler_handle* &handle)

> **Effects**: If `handle` is not empty, blocks the program execution until all worker threads have been completed; otherwise, does nothing. Throws the `oneapi::tbb::unsafe_wait` exception if it is not safe to wait for the completion of the worker threads.

The following conditions should be met for finalization to succeed:

- No active, not yet terminated, instances of `task_arena` class exist in the whole program.
- `task_scheduler_handle::release` is called for each other active instance of `task_scheduler_handle` class, possibly by different application threads.

Under these conditions, it is guaranteed that at least one `finalize` call succeeds, at which point all worker threads have been completed. If calls are performed simultaneously, more than one call might succeed.

Note: If user knows how many active `task_scheduler_handle` instances exist in the program, it is necessary to `release` all but the last one, then call `finalize` for the last instance.

Caution: The method always fails if called within a task, a parallel algorithm, or a flow graph node.

---

bool **finalize**(*task_scheduler_handle* &handle, const std::nothrow_t&) noexcept

> **Effects**: If `handle` is not empty, blocks the program execution until all worker threads have been completed; otherwise, does nothing. The behavior is the same as `finalize(handle)` however, `false` is returned instead of exception or `true` if no exception.

Examples

```
#include <oneapi/tbb/global_control.h>
#include <oneapi/tbb/parallel_for.h>

#include <iostream>

int main() {
    oneapi::tbb::task_scheduler_handle handle;

    handle = oneapi::tbb::task_scheduler_handle{oneapi::tbb::attach{}};

    // Do some parallel work here, e.g.
    oneapi::tbb::parallel_for(0, 10000, [](int){});
    try {
        oneapi::tbb::finalize(handle);
        // oneTBB worker threads are terminated at this point.
    } catch (const oneapi::tbb::unsafe_wait&) {
        std::cerr << "Failed to terminate the worker threads." << std::endl;
    }
    return 0;
}
```

See also:

- *attach*

Task Group

task_group

[scheduler.task_group]

A `task_group` represents the concurrent execution of a group of tasks. You can dynamically add tasks to the group while it is executing. The thread executing `task_group::wait()` might participate in other tasks that are not related to the particular `task_group`.

```
// Defined in header <oneapi/tbb/task_group.h>

namespace oneapi {
namespace tbb {

    class task_group {
    public:
        task_group();
        task_group(task_group_context& context);

        ~task_group();

        template<typename Func>
        void run(Func&& f);

        template<typename Func>
        task_handle defer(Func&& f);

        void run(task_handle&& h);

        template<typename Func>
        task_group_status run_and_wait(const Func& f);

        task_group_status run_and_wait(task_handle&& h);

        task_group_status wait();
        void cancel();
    };

    bool is_current_task_group_canceling();

} // namespace tbb
} // namespace oneapi
```

Member functions

task_group()

Constructs an empty task_group.

task_group(*task_group_context* &context)

Constructs an empty task_group. All tasks added into the task_group are associated with the context.

~task_group()

Destroys the task_group.

Requires: Method wait must be called before destroying a task_group, otherwise, the destructor throws an exception.

template<typename **F**>
task_handle **defer**(*F* &&f)

Creates a deferred task to compute f() and returns task_handle pointing to it.

The task is not scheduled for the execution until it is explicitly requested, for example, with the task_group::run method. However, the task is still added into the task_group, thus the task_group::wait method waits until the task_handle is either scheduled or destroyed.

The F type must meet the *Function Objects* requirements described in the [function.objects] section of the ISO C++ standard.

Returns: task_handle object pointing to a task to compute f().

template<typename **Func**>
void **run**(*Func* &&f)

Adds a task to compute f() and returns immediately. The Func type must meet the *Function Objects* requirements described in the [function.objects] section of the ISO C++ standard.

void **run**(task_handle &&h)

Schedules the task object pointed by the h for the execution.

Note:

The failure to satisfy the following conditions leads to undefined behavior:

- h is not empty.
- *this is the same task_group that h is created with.

---

template<typename **Func**>
task_group_status **run_and_wait**(const *Func* &f)

Equivalent to {run(f); return wait();}. The Func type must meet the *Function Objects* requirements described in the [function.objects] section of the ISO C++ standard.

Returns: The status of task_group. See *task_group_status*.

task_group_status **wait**()

Waits for all tasks in the group to complete or be cancelled.

Returns: The status of task_group. See *task_group_status*.

void **cancel**()

Cancels all tasks in this task_group.

Non-member functions

bool **is_current_task_group_canceling**()

Returns true if an innermost `task_group` executing on this thread is cancelling its tasks.

task_group_status

[scheduler.task_group_status]

A `task_group_status` type represents the status of a `task_group`.

```
namespace oneapi {
namespace tbb {
    enum task_group_status {
        not_complete,
        complete,
        canceled
    };
} // namespace tbb
} // namespace oneapi
```

Member constants

not_complete

Not cancelled and not all tasks in a group have completed.

complete

Not cancelled and all tasks in a group have completed.

canceled

Task group received cancellation request.

task_handle

[scheduler.task_handle]

An instance of `task_handle` type owns a deferred task object.

```
namespace oneapi {
namespace tbb {

   class task_handle {
   public:
       task_handle();
       task_handle(task_handle&& src);

       ~task_handle();

       task_handle& operator=(task_handle&& src);

       explicit operator bool() const noexcept;
```

(continues on next page)

(continued from previous page)

```
    };

    bool operator==(task_handle const& h, std::nullptr_t) noexcept;
    bool operator==(std::nullptr_t, task_handle const& h) noexcept;

    bool operator!=(task_handle const& h, std::nullptr_t) noexcept;
    bool operator!=(std::nullptr_t, task_handle const& h) noexcept;

} // namespace tbb
} // namespace oneapi
```

Member Functions

task_handle()

Creates an empty `task_handle` object.

task_handle(*task_handle* &&src)

Constructs `task_handle` object with the content of `src` using move semantics. `src` becomes empty after the construction.

~task_handle()

Destroys the `task_handle` object and associated task if it exists.

task_handle &**operator=**(task_handle &&src)

Replaces the content of `task_handle` object with the content of `src` using move semantics. `src` becomes empty after the assignment. The previously associated task object, if any, is destroyed before the assignment.

Returns: Reference to `*this`.

explicit **operator bool**() const noexcept

Checks if `*this` has an associated task object.

Returns: `true` if `*this` is not empty, `false` otherwise.

Non-Member Functions

```
bool operator==(task_handle const& h, std::nullptr_t) noexcept
bool operator==(std::nullptr_t, task_handle const& h) noexcept
```

Returns: `true` if `h` is empty, `false` otherwise.

```
bool operator!=(task_handle const& h, std::nullptr_t) noexcept
bool operator!=(std::nullptr_t, task_handle const& h) noexcept
```

Returns: `true` if `h` is not empty, `false` otherwise.

Task Arena

task_arena

[scheduler.task_arena]

A class that represents an explicit, user-managed task scheduler arena.

```
// Defined in header <oneapi/tbb/task_arena.h>

namespace oneapi {
    namespace tbb {

        class task_arena {
        public:
            static const int automatic = /* unspecified */;
            static const int not_initialized = /* unspecified */;
            enum class priority : /* unspecified type */ {
                low = /* unspecified */,
                normal = /* unspecified */,
                high = /* unspecified */
            };

            struct constraints {
                constraints(numa_node_id numa_node_       = task_arena::automatic,
                            int          max_concurrency_ = task_arena::automatic);

                constraints& set_numa_id(numa_node_id id);
                constraints& set_max_concurrency(int maximal_concurrency);
                constraints& set_core_type(core_type_id id);
                constraints& set_max_threads_per_core(int threads_number);

                numa_node_id numa_id = task_arena::automatic;
                int max_concurrency = task_arena::automatic;
                core_type_id core_type = task_arena::automatic;
                int max_threads_per_core = task_arena::automatic;
            };

            task_arena(int max_concurrency = automatic, unsigned reserved_for_masters =␣
↪1,
                       priority a_priority = priority::normal);
            task_arena(constraints a_constraints, unsigned reserved_for_masters = 1,
                       priority a_priority = priority::normal);
            task_arena(const task_arena &s);
            explicit task_arena(oneapi::tbb::attach);
            ~task_arena();

            void initialize();
            void initialize(int max_concurrency, unsigned reserved_for_masters = 1,
                            priority a_priority = priority::normal);
            void initialize(constraints a_constraints, unsigned reserved_for_masters = 1,
                            priority a_priority = priority::normal);
            void initialize(oneapi::tbb::attach);
```

(continues on next page)

(continued from previous page)

```
            void terminate();

            bool is_active() const;
            int max_concurrency() const;

            template<typename F> auto execute(F&& f) -> decltype(f());
            template<typename F> void enqueue(F&& f);

            void enqueue(task_handle&& h);
        };

    } // namespace tbb
} // namespace oneapi
```

A `task_arena` class represents a place where threads may share and execute tasks.

The number of threads that may simultaneously execute tasks in a `task_arena` is limited by its concurrency level.

Each user thread that invokes any parallel construction outside an explicit `task_arena` uses an implicit task arena representation object associated with the calling thread.

The tasks spawned or enqueued into one arena cannot be executed in another arena.

Each `task_arena` has a `priority`. The tasks from `task_arena` with higher priority are given a precedence in execution over the tasks from `task_arena` with lower priority.

Note: The `task_arena` constructors do not create an internal task arena representation object. It may already exist in case of the "attaching" constructor; otherwise, it is created by an explicit call to `task_arena::initialize` or lazily on first use.

Member types and constants

static const int **`automatic`**

When passed as `max_concurrency` to the specific constructor, arena concurrency is automatically set based on the hardware configuration.

static const int **`not_initialized`**

When returned by a method or function, indicates that there is no active `task_arena` or that the `task_arena` object has not yet been initialized.

enum `priority::`**`low`**

When passed to a constructor or the `initialize` method, the initialized `task_arena` has a lowered priority.

enum `priority::`**`normal`**

When passed to a constructor or the `initialize` method, the initialized `task_arena` has regular priority.

enum `priority::`**`high`**

When passed to a constructor or the `initialize` method, the initialized `task_arena` has a raised priority.

struct **`constraints`**

Represents limitations applied to threads within `task_arena`.

Starting from C++20 this class should be an aggregate type to support the designated initialization.

numa_node_id *constraints*::**numa_id**

An integral logical index uniquely identifying a NUMA node. If set to non-automatic value, then this NUMA node will be considered as preferred for all the threads within the arena.

Note: NUMA node ID is considered valid if it was obtained through tbb::info::numa_nodes().

---

int *constraints*::**max_concurrency**

The maximum number of threads that can participate in work processing within the `task_arena` at the same time.

core_type_id *constraints*::**core_type**

An integral logical index uniquely identifying a core type. If set to non-automatic value, then this core type will be considered as preferred for all the threads within the arena.

Note: core type ID is considered valid if it was obtained through `tbb::info::core_types()`.

---

int *constraints*::**max_threads_per_core**

The maximum number of threads that can be scheduled to one core simultaneously.

constraints::**constraints**(numa_node_id numa_node_ = *task_arena*::automatic, int max_concurrency_ = *task_arena*::automatic)

Constructs the constraints object with the provided *numa_id* and *max_concurrency* settings.

Note: To support designated initialization this constructor is omitted starting from C++20. Aggregate initialization is supposed to be used instead.

---

constraints &*constraints*::**set_numa_id**(numa_node_id id)

Sets the *numa_id* to the provided `id`. Returns the reference to the updated constraints object.

constraints &*constraints*::**set_max_concurrency**(int maximal_concurrency)

Sets the *max_concurrency* to the provided `maximal_concurrency`. Returns the reference to the updated constraints object.

constraints &*constraints*::**set_core_type**(core_type_id id)

Sets the *core_type* to the provided `id`. Returns the reference to the updated constraints object.

constraints &*constraints*::**set_max_threads_per_core**(int threads_number)

Sets the *max_threads_per_core* to the provided `threads_number`. Returns the reference to the updated constraints object.

Member functions

task_arena(int max_concurrency = *automatic*, unsigned reserved_for_masters = 1, priority a_priority = priority::*normal*)

Creates a `task_arena` with a certain concurrency limit (`max_concurrency`) and priority (`a_priority`). Some portion of the limit can be reserved for application threads with `reserved_for_masters`. The amount for reservation cannot exceed the limit.

> **Caution:** If `max_concurrency` and `reserved_for_masters` are explicitly set to be equal and greater than 1, oneTBB worker threads will never join the arena. As a result, the execution guarantee for enqueued tasks is not valid in such arena. Do not use `task_arena::enqueue()` with an arena set to have no worker threads.

task_arena(*constraints* a_constraints, unsigned reserved_for_masters = 1, priority a_priority = priority::*normal*)

Creates a `task_arena` with a certain constraints(`a_constraints`) and priority (`a_priority`). Some portion of the limit can be reserved for application threads with `reserved_for_masters`. The amount for reservation cannot exceed the concurrency limit specified in `constraints`.

> **Caution:** If `constraints::max_concurrency` and `reserved_for_masters` are explicitly set to be equal and greater than 1, oneTBB worker threads will never join the arena. As a result, the execution guarantee for enqueued tasks is not valid in such arena. Do not use `task_arena::enqueue()` with an arena set to have no worker threads.

If `constraints::numa_node` is specified, then all threads that enter the arena are automatically pinned to corresponding NUMA node.

task_arena(const *task_arena*&)

Copies settings from another `task_arena` instance.

explicit **task_arena**(oneapi::tbb::attach)

Creates an instance of `task_arena` that is connected to the internal task arena representation currently used by the calling thread. If no such arena exists yet, creates a `task_arena` with default parameters.

Note: Unlike other constructors, this one automatically initializes the new `task_arena` when connecting to an already existing arena.

~task_arena()

Destroys the `task_arena` instance, but the destruction may not be synchronized with any task execution inside this `task_arena`. It means that an internal task arena representation associated with this `task_arena` instance can be destroyed later. Not thread-safe for concurrent invocations of other methods.

void **initialize**()

Performs actual initialization of internal task arena representation.

Note: After the call to `initialize`, the arena parameters are fixed and cannot be changed.

---

void **initialize**(int max_concurrency, unsigned reserved_for_masters = 1, priority a_priority = priority::*normal*)

Same as above, but overrides previous arena parameters.

void **initialize**(*constraints* a_constraints, unsigned reserved_for_masters = 1, priority a_priority = priority::*normal*)

Same as above.

void **initialize**(oneapi::tbb::attach)

If an internal task arena representation currently used by the calling thread, the method ignores arena parameters and connects `task_arena` to that internal task arena representation. The method has no effect when called for an already initialized `task_arena`.

void **terminate**()

Removes the reference to the internal task arena representation without destroying the task_arena object, which can then be re-used. Not thread safe for concurrent invocations of other methods.

bool **is_active**() const

Returns `true` if the `task_arena` has been initialized; `false`, otherwise.

int **max_concurrency**() const

Returns the concurrency level of the `task_arena`. Does not require the `task_arena` to be initialized and does not perform initialization.

template<F>
void **enqueue**(F &&f)

Enqueues a task into the `task_arena` to process the specified functor and immediately returns. The `F` type must meet the *Function Objects* requirements described in the [function.objects] section of the ISO C++ standard. The task is scheduled for eventual execution by a worker thread even if no thread ever explicitly waits for the task to complete. If the total number of worker threads is zero, a special additional worker thread is created to execute enqueued tasks.

Note: The method does not require the calling thread to join the arena; that is, any number of threads outside of the arena can submit work to it without blocking.

Caution: There is no guarantee that tasks enqueued into an arena execute concurrently with respect to any other tasks there.

Caution: An exception thrown and not caught in the functor results in undefined behavior.

template<F>
auto **execute**(F &&f) -> decltype(*f*())

Executes the specified functor in the `task_arena` and returns the value returned by the functor. The `F` type must meet the *Function Objects* requirements described in the [function.objects] section of the ISO C++ standard.

The calling thread joins the `task_arena` if possible, and executes the functor. Upon return it restores the previous task scheduler state and floating-point settings.

If joining the `task_arena` is not possible, the call wraps the functor into a task, enqueues it into the arena, waits using an OS kernel synchronization object for another opportunity to join, and finishes after the task completion.

An exception thrown in the functor will be captured and re-thrown from `execute`.

Note: Any number of threads outside of the arena can submit work to the arena and be blocked. However, only the maximal number of threads specified for the arena can participate in executing the work.

---

void **enqueue**(task_handle &&h)

Enqueues a task owned by `h` into the `task_arena` for processing.

The behavior of this function is identical to the generic version (`template<typename F> void task_arena::enqueue(F&& f)`), except parameter type.

Note: h should not be empty to avoid an undefined behavior.

Example

The example demonstrates task_arena NUMA support API. Each constructed task_arena is pinned to the corresponding NUMA node.

```
#include "oneapi/tbb/task_group.h"
#include "oneapi/tbb/task_arena.h"

#include <vector>

int main() {
    std::vector<oneapi::tbb::numa_node_id> numa_nodes = oneapi::tbb::info::numa_nodes();
    std::vector<oneapi::tbb::task_arena> arenas(numa_nodes.size());
    std::vector<oneapi::tbb::task_group> task_groups(numa_nodes.size());

    for (int i = 0; i < numa_nodes.size(); i++) {
        arenas[i].initialize(oneapi::tbb::task_arena::constraints(numa_nodes[i]));
    }

    for (int i = 0; i < numa_nodes.size(); i++) {
        arenas[i].execute([&task_groups, i] {
            task_groups[i].run([] {
                /* executed by the thread pinned to specified NUMA node */
            });
        });
    }

    for (int i = 0; i < numa_nodes.size(); i++) {
        arenas[i].execute([&task_groups, i] {
            task_groups[i].wait();
        });
    }

    return 0;
}
```

See also:

- *attach*
- *task_group*
- *task_scheduler_observer*

this_task_arena

[scheduler.this_task_arena]

The namespace for functions applicable to the current `task_arena`.

The namespace `this_task_arena` contains global functions for interaction with the `task_arena` currently used by the calling thread.

```
// Defined in header <oneapi/tbb/task_arena.h>

namespace oneapi {
namespace tbb {
    namespace this_task_arena {
        int current_thread_index();
        int max_concurrency();
        template<typename F> auto isolate(F&& f) -> decltype(f());

        void enqueue(task_handle&& h);

        template<typename F> void enqueue(F&& f) ;
    }
} // namespace tbb
} // namespace oneapi
```

int **current_thread_index**()

Returns the thread index in a `task_arena` currently used by the calling thread, or `task_arena::not_initialized` if the thread has not yet initialized the task scheduler.

A thread index is an integer number between 0 and the `task_arena` concurrency level. Thread indexes are assigned to both application threads and worker threads on joining an arena and are kept until exiting the arena. Indexes of threads that share an arena are unique, that is, no two threads within the arena can have the same index at the same time - but not necessarily consecutive.

Note: Since a thread may exit the arena at any time if it does not execute a task, the index of a thread may change between any two tasks, even those belonging to the same task group or algorithm.

Note: Threads that use different arenas may have the same current index value.

Note: Joining a nested arena in `execute()` may change current index value while preserving the index in the outer arena which will be restored on return.

---

int **max_concurrency**()

Returns the concurrency level of the `task_arena` currently used by the calling thread. If the thread has not yet initialized the task scheduler, returns the concurrency level determined automatically for the hardware configuration.

template<F>
auto **isolate**(F &&f) -> decltype(*f*())

Runs the specified functor in isolation by restricting the calling thread to process only tasks scheduled in the scope of the functor (also called the isolation region). The function returns the value returned by the functor.

The `F` type must meet the *Function Objects* requirements described in the [function.objects] section of the ISO C++ standard.

> **Caution:** The object returned by the functor cannot be a reference. `std::reference_wrapper` can be used instead.

template<typename **F**>
void **enqueue**(*F* &&f)

Enqueues a task into the `task_arena` currently used by the calling thread to process the specified functor, then returns immediately. The `F` type must meet the *Function Objects* requirements described in the [function.objects] section of the ISO C++ standard.

Behavior of this function is identical to `template<typename F> void task_arena::enqueue(F&& f)` applied to the `task_arena` object constructed with `attach` parameter.

void **enqueue**(task_handle &&h)

Enqueues a task owned by `h` into the `task_arena` that is currently used by the calling thread.

The behavior of this function is identical to the generic version (`template<typename F> void enqueue(F&& f)`), except the parameter type.

Note: `h` should not be empty to avoid an undefined behavior.

---

task_scheduler_observer

[scheduler.task_scheduler_observer]

Class that represents thread interest in task scheduling services.

```
// Defined in header <oneapi/tbb/task_scheduler_observer.h>

namespace oneapi {
namespace tbb {

   class task_scheduler_observer {
   public:
       task_scheduler_observer();
       explicit task_scheduler_observer( task_arena& a );
       virtual ~task_scheduler_observer();

       void observe( bool state=true );
       bool is_observing() const;

       virtual void on_scheduler_entry( bool is_worker ) {}
       virtual void on_scheduler_exit( bool is_worker } {}
   };

} // namespace tbb
} // namespace oneapi
```

A `task_scheduler_observer` permits clients to observe when a thread starts and stops processing tasks, either globally or in a certain task scheduler arena. You typically derive your own observer class from

`task_scheduler_observer`, and override virtual methods `on_scheduler_entry` or `on_scheduler_exit`. Observation can be enabled and disabled for an observer instance; it is disabled on creation. Remember to call `observe()` to enable observation.

Exceptions thrown and not caught in the overridden methods of `task_scheduler_observer` result in undefined behavior.

Member functions

`task_scheduler_observer()`

Constructs a `task_scheduler_observer` object in the inactive state (observation is disabled). For a created observer, entry/exit notifications are invoked whenever a worker thread joins/leaves the arena of the observer's owner thread. If a thread is already in the arena when the observer is activated, the entry notification is called before it executes the first stolen task.

explicit **`task_scheduler_observer`**(*task_arena*&)

Constructs a `task_scheduler_observer` object for a given arena in inactive state (observation is disabled). For created observer, entry/exit notifications are invoked whenever a thread joins/leaves arena. If a thread is already in the arena when the observer is activated, the entry notification is called before it executes the first stolen task.

Constructs a `task_scheduler_observer` object in the inactive state (observation is disabled), which receives notifications from threads entering and exiting the specified `task_arena`.

`~task_scheduler_observer()`

Disables observing and destroys the observer instance. Waits for extant invocations of `on_scheduler_entry` and `on_scheduler_exit` to complete.

void **`observe`**(bool state = true)

Enables observing if `state` is true; disables observing if `state` is false.

bool **`is_observing`**() const

Returns: True if observing is enabled; false, otherwise.

virtual void **`on_scheduler_entry`**(bool is_worker)

The task scheduler invokes this method for each thread that starts participating in oneTBB work or enters an arena after the observation is enabled. For threads that already execute tasks, the method is invoked before executing the first task stolen after enabling the observation.

If a thread enables the observation and then spawns a task, it is guaranteed that the task, as well as all the tasks it creates, will be executed by threads which have invoked `on_scheduler_entry`.

The flag `is_worker` is true if the thread was created by oneTBB; false, otherwise.

Effects: The default behavior does nothing.

virtual void **`on_scheduler_exit`**(bool is_worker)

The task scheduler invokes this method when a thread stops participating in task processing or leaves an arena.

> **Caution:** A process does not wait for the worker threads to clean up, and can terminate before `on_scheduler_exit` is invoked.

Effects: The default behavior does nothing.

Example

The following example sketches the code of an observer that pins oneTBB worker threads to hardware threads.

```
class pinning_observer : public oneapi::tbb::task_scheduler_observer {
public:
    affinity_mask_t m_mask; // HW affinity mask to be used for threads in an arena
    pinning_observer( oneapi::tbb::task_arena &a, affinity_mask_t mask )
        : oneapi::tbb::task_scheduler_observer(a), m_mask(mask) {
        observe(true); // activate the observer
    }
    void on_scheduler_entry( bool worker ) override {
        set_thread_affinity(oneapi::tbb::this_task_arena::current_thread_index(), m_
↪mask);
    }
    void on_scheduler_exit( bool worker ) override {
        restore_thread_affinity();
    }
};
```

Helper types

attach tag type

[scheduler.attach]

An `attach` tag type is specifically used with `task_arena` and `task_scheduler_handle` interfaces. It is guaranteed to be constructible by default.

```
namespace oneapi {
    namespace tbb {
        using attach = /* unspecified */
    }
}
```

See also:

- *task_arena*
- *task_scheduler_handle*

8.2.5 Containers

[containers]

The container classes provided by oneAPI Threading Building Blocks (oneTBB) permit multiple threads to simultaneously invoke certain methods on the same container.

Sequences

concurrent_vector

[containers.concurrent_vector]

`concurrent_vector` is a class template for a vector that can be concurrently grown and accessed.

Class Template Synopsis

```
// Defined in header <oneapi/tbb/concurrent_vector.h>

namespace oneapi {
    namespace tbb {

        template <typename T,
                  typename Allocator = cache_aligned_allocator<T>>
        class concurrent_vector {
            using value_type = T;
            using allocator_type = Allocator;

            using size_type = <implementation-defined unsigned integer type>;
            using difference_type = <implementation-defined signed integer type>;

            using reference = value_type&;
            using const_reference = const value_type&;

            using pointer = typename std::allocator_traits<allocator_type>::pointer;
            using const_pointer = typename std::allocator_traits<allocator_type>::const_
↪pointer;

            using iterator = <implementation-defined RandomAccessIterator>;
            using const_iterator = <implementation-defined constant RandomAccessIterator>
↪;

            using reverse_iterator = std::reverse_iterator<iterator>;
            using const_reverse_iterator = std::reverse_iterator<const_iterator>;

            using range_type = <implementation-defined ContainerRange>;
            using const_range_type = <implementation-defined constant ContainerRange>;

            // Construction, destruction, copying
            concurrent_vector();
            explicit concurrent_vector( const allocator_type& alloc ) noexcept;

            explicit concurrent_vector( size_type count, const value_type& value,
                                        const allocator_type& alloc = allocator_type() );

            explicit concurrent_vector( size_type count,
                                        const allocator_type& alloc = allocator_type() );

            template <typename InputIterator>
```

(continues on next page)

(continued from previous page)

```
            concurrent_vector( InputIterator first, InputIterator last,
                               const allocator_type& alloc = allocator_type() );

            concurrent_vector( std::initializer_list<value_type> init,
                               const allocator_type& alloc = allocator_type() );

            concurrent_vector( const concurrent_vector& other );
            concurrent_vector( const concurrent_vector& other, const allocator_type&␣
↪alloc );

            concurrent_vector( concurrent_vector&& other ) noexcept;
            concurrent_vector( concurrent_vector&& other, const allocator_type& alloc );

            ~concurrent_vector();

            concurrent_vector& operator=( const concurrent_vector& other );

            concurrent_vector& operator=( concurrent_vector&& other ) noexcept(/*See␣
↪details*/);

            concurrent_vector& operator=( std::initializer_list<value_type> init );

            void assign( size_type count, const value_type& value );

            template <typename InputIterator>
            void assign( InputIterator first, InputIterator last );

            void assign( std::initializer_list<value_type> init );

            // Concurrent growth
            iterator grow_by( size_type delta );
            iterator grow_by( size_type delta, const value_type& value );

            template <typename InputIterator>
            iterator grow_by( InputIterator first, InputIterator last );

            iterator grow_by( std::initializer_list<value_type> init );

            iterator grow_to_at_least( size_type n );
            iterator grow_to_at_least( size_type n, const value_type& value );

            iterator push_back( const value_type& value );
            iterator push_back( value_type&& value );

            template <typename... Args>
            iterator emplace_back( Args&&... args );

            // Element access
            value_type& operator[]( size_type index );
            const value_type& operator[]( size_type index ) const;

            value_type& at( size_type index );
```

(continues on next page)

(continued from previous page)

```
        const value_type& at( size_type index ) const;

        value_type& front();
        const value_type& front() const;

        value_type& back();
        const value_type& back() const;

        // Iterators
        iterator begin();
        const_iterator begin() const;
        const_iterator cbegin() const;

        iterator end();
        const_iterator end() const;
        const_iterator cend() const;

        reverse_iterator rbegin();
        const_reverse_iterator rbegin() const;
        const_reverse_iterator crbegin() const;

        reverse_iterator rend();
        const_reverse_iterator rend() const;
        const_reverse_iterator crend() const;

        // Size and capacity
        size_type size() const noexcept;

        bool empty() const noexcept;

        size_type max_size() const noexcept;

        size_type capacity() const noexcept;

        // Concurrently unsafe operations
        void reserve( size_type n );

        void resize( size_type n );
        void resize( size_type n, const value_type& value );

        void shrink_to_fit();

        void swap( concurrent_vector& other ) noexcept(/*See details*/);

        void clear();

        allocator_type get_allocator() const;

        // Parallel iteration
        range_type range( size_type grainsize = 1 );
        const_range_type range( size_type grainsize = 1 ) const;
    }; // class concurrent_vector
```

(continues on next page)

(continued from previous page)

```
    } // namespace tbb
} // namespace oneapi
```

Requirements

- The type `T` must meet the following requirements:
  - Requirements of `Erasable` from the [container.requirements] ISO C++ Standard section.
  - Its destructor must not throw an exception.
  - If its default constructor can throw an exception, the destructor must be non-virtual and work correctly on zero-filled memory.
  - Member functions can impose stricter requirements depending on the type of the operation.
- The type `Allocator` must meet the `Allocator` requirements from the [allocator.requirements] ISO C++ section.

Description

`oneapi::tbb::concurrent_vector` is a class template that represents a sequence container with the following features:

- Multiple threads can concurrently grow the container and append new elements.
- Random access by index. The index of the first element is zero.
- Growing the container does not invalidate any existing iterators or indices.

Exception Safety

Concurrent growing is fundamentally incompatible with ideal exception safety. Nonetheless, `oneapi::tbb::concurrent_vector` offers a practical level of exception safety.

Growth and vector assignment append a sequence of elements to a vector. If an exception occurs, the impact on the vector depends on the cause of the exception:

- If the exception is thrown by the constructor of an element, all subsequent elements in the appended sequence will be zero-filled.
- Otherwise, the exception is thrown by the vector allocator. The vector becomes broken. Each element in the appended sequence will be in one of three states:
  - constructed
  - zero-filled
  - unallocated in memory

Once a vector becomes broken, note the following when accessing it:

- Accessing an unallocated element with the method `at` causes an exception `std::range_error`. Accessing an unallocated element using any other method has undefined behavior.
- The values of `capacity()` and `size()` may be less than expected.
- Access to a broken vector via `back()` has undefined behavior.

However, the following guarantees hold for broken or unbroken vectors:

- Let `k` be an index of an unallocated element. Then `size() <= capacity() <= k`.
- Growth operations never cause `size()` or `capacity()` to decrease.

If a concurrent growth operation successfully completes, the appended sequence remains valid and accessible even if a subsequent growth operations fails.

Member functions

Construction, destruction, copying

Empty container constructors

```
concurrent_vector();

explicit concurrent_vector( const allocator_type& alloc );
```

Constructs an empty `concurrent_vector`.

If provided, uses the allocator `alloc` to allocate the memory.

Constructors from the sequence of elements

```
explicit concurrent_vector( size_type count, const value_type& value,
                            const allocator_type& alloc = allocator_type() );
```

Constructs a `concurrent_vector` containing `count` copies of the `value` using the allocator `alloc`.

```
explicit concurrent_vector( size_type count,
                            const allocator_type& alloc = allocator_type() );
```

Constructs a `concurrent_vector` containing `n` default constructed in-place elements using the allocator `alloc`.

```
template <typename InputIterator>
concurrent_vector( InputIterator first, InputIterator last,
                   const allocator_type& alloc = allocator_type() );
```

Constructs a `concurrent_vector` contains all elements from the half-open interval `[first, last)` using the allocator `alloc`.

Requirements: the type `InputIterator` must meet the requirements of `InputIterator` from the [input.iterators] ISO C++ Standard section.

```
concurrent_vector( std::initializer_list<value_type> init,
                   const allocator_type& alloc = allocator_type() );
```

Equivalent to `concurrent_vector(init.begin(), init.end(), alloc)`.

Copying constructors

```
concurrent_vector( const concurrent_vector& other );

concurrent_vector( const concurrent_vector& other,
                   const allocator_type& alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Moving constructors

```
concurrent_vector( concurrent_vector&& other );

concurrent_vector( concurrent_vector&& other,
                   const allocator_type& alloc );
```

Constructs a `concurrent_vector` with the contents of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by calling `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Destructor

```
~concurrent_vector();
```

Destroys the `concurrent_vector`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

Assignment operators

```
concurrent_vector& operator=( const concurrent_vector& other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_assignment::` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_vector& operator=( concurrent_vector&& other ) noexcept(/*See␣
↪below*/);
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment::` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

Exceptions: `noexcept` specification:

```
noexcept(std::allocator_traits<allocator_type>::propagate_on_container_
↪move_assignment::value ||
        std::allocator_traits<allocator_type>::is_always_equal::value)
```

```
concurrent_vector& operator=( std::initializer_list<value_type> init );
```

Replaces all elements in `*this` by the elements in `init`.

The behavior is undefined in case of concurrent operations with `*this`.

Returns: a reference to `*this`.

assign

```
void assign( size_type count, const value_type& value );
```

Replaces all elements in `*this` by `count` copies of `value`.

```
template <typename InputIterator>
void assign( InputIterator first, InputIterator last );
```

Replaces all elements in `*this` by the elements from the half-open interval `[first, last)`.

This overload only participates in overload resolution if the type `InputIterator` meets the requirements of *InputIterator* from the [input.iterators] ISO C++ Standard section.

```
void assign( std::initializer_list<value_type> init );
```

Equivalent to `assign(init.begin(), init.end())`.

get_allocator

```
allocator_type get_allocator() const;
```

Returns: a copy of the allocator associated with `*this`.

Concurrent growth

All member functions in this section can be performed concurrently with each other, element access methods and while traversing the container.

grow_by

```
iterator grow_by( size_type delta );
```

Appends a sequence comprising `delta` new default-constructed in-place elements to the end of the vector.

Returns: iterator to the beginning of the appended sequence.

Requirements: the type `value_type` must meet the `DefaultConstructible` and `EmplaceConstructible` requirements from [defaultconstructible] and [container.requirements] ISO C++ sections.

```
iterator grow_by( size_type delta, const value_type& value );
```

Appends a sequence comprising `delta` copies of `value` to the end of the vector.

Returns: iterator to the beginning of the appended sequence.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
template <typename InputIterator>
iterator grow_by( InputIterator first, InputIterator last );
```

Appends a sequence comprising all elements from the half-open interval `[first, last)` to the end of the vector.

Returns: iterator to the beginning of the appended sequence.

This overload participates in overload resolution only if the type `InputIterator` meets the requirements of *InputIterator* from the [input.iterators] ISO C++ Standard section.

```
iterator grow_by( std::initializer_list<value_type> init );
```

Equivalent to `grow_by(init.begin(), init.end())`.

grow_to_at_least

```
iterator grow_to_at_least( size_type n );
```

Appends minimal sequence of default constructed in-place elements such that `size() >= n`.

Returns: iterator to the beginning of the appended sequence.

Requirements: the type `value_type` must meet the `DefaultConstructible` and `EmplaceConstructible` requirements from [defaultconstructible] and [container.requirements] ISO C++ sections.

```
iterator grow_to_at_least( size_type n, const value_type& value );
```

Appends minimal sequence of comprising copies of `value` such that `size() >= n`.

Returns: iterator to the beginning of the appended sequence.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

push_back

```
iterator push_back( const value_type& value );
```

Appends a copy of `value` to the end of the vector.

Returns: iterator to the appended element.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
iterator push_back( value_type&& value );
```

Appends `value` to the end of the vector using move semantics.

`value` is left in a valid, but unspecified state.

Returns: iterator to the appended element.

Requirements: the type `value_type` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

emplace_back

```
template <typename... Args>
iterator emplace_back( Args&&... args );
```

Appends an element constructed in-place from `args` to the end of the vector.

Returns: iterator to the appended element.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ section.

Element access

All member functions in this section can be performed concurrently with each other, concurrent growth methods and while traversing the container.

In case of concurrent growth, the element returned by the access method can refer to the element that is under construction of the other thread.

Access by index

```
value_type& operator[]( size_type index );

const value_type& operator[]( size_type index ) const;
```

Returns: a reference to the element on the position `index`.

The behavior is undefined if `index() >= size()`.

```
value_type& at( size_type index );

const value_type& at( size_type index ) const;
```

Returns: a reference to the element on the position `index`.

Throws:

- `std::out_of_range` if `index >= size()`.
- `std::range_error` if the vector is broken and the element on the position `index` unallocated.

Access the first and the last element

```
value_type& front();

const value_type& front() const;
```

Returns: a reference to the first element in the vector.

```
value_type& back();

const value_type& back() const;
```

Returns: a reference to the last element in the vector.

Iterators

The types `concurrent_vector::iterator` and `concurrent_vector::const_iterator` meet the requirements of `RandomAccessIterator` from the [random.access.iterators] ISO C++ Standard section.

begin and cbegin

```
iterator begin();

const_iterator begin() const;

const_iterator cbegin() const;
```

Returns: an iterator to the first element in the vector.

end and cend

```
iterator end();

const_iterator end() const;

const_iterator cend() const;
```

Returns: an iterator to the element that follows the last element in the vector.

rbegin and crbegin

```
reverse_iterator rbegin();

const_reverse_iterator rbegin() const;

const_reverse_iterator crbegin() const;
```

Returns: a reverse iterator to the first element of the reversed vector.

rend and crend

```
reverse_iterator rend();

const_reverse_iterator rend() const;

const_reverse_iterator crend() const;
```

Returns: a reverse iterator that follows the last element of the reversed vector.

Size and capacity

size

```
size_type size() const noexcept;
```

Returns: the number of elements in the vector.

empty

```
bool empty() const noexcept;
```

Returns: `true` if the vector is empty; `false`, otherwise.

max_size

```
size_type max_size() const noexcept;
```

Returns: the maximum number of elements that the vector can hold.

capacity

```
size_type capacity() const noexcept;
```

Returns: the maximum number of elements that the vector can hold without allocating more memory.

Concurrently unsafe operations

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

Reserving

```
void reserve( size_type n );
```

Reserves memory for at least `n` elements.

Throws: `std::length_error` if `n > max_size()`.

Resizing

```
void resize( size_type n );
```

If `n < size()`, the vector is reduced to its first `n` elements.

Otherwise, appends `n - size()` new elements default-constructed in-place to the end of the vector.

```
void resize( size_type n, const value_type& value );
```

If `n < size()`, the vector is reduced to its first `n` elements.

Otherwise, appends `n - size()` copies of `value` to the end of the vector.

shrink_to_fit

```
void shrink_to_fit();
```

Removes the unused capacity of the vector.

Call for this method can also reorganize the internal vector representation in the memory.

clear

```
void clear();
```

Removes all elements from the container.

swap

```
void swap( concurrent_vector& other ) noexcept(/*See below*/);
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`.

Otherwise, if `get_allocator() != other.get_allocator()`, the behavior is undefined.

Exceptions: `noexcept` specification:

```
noexcept(std::allocator_traits<allocator_type>::propagate_on_container_
↪swap::value ||
        std::allocator_traits<allocator_type>::is_always_equal::value
```

Parallel iteration

Member types `concurrent_vector::range_type` and `concurrent_vector::const_range_type` meet the *ContainerRange requirements*.

These types differ only in that the bounds for a `concurrent_vector::const_range_type` are of type `concurrent_vector::const_iterator`, whereas the bounds for a `concurrent_vector::range_type` are of type `concurrent_vector::iterator`.

range member function

```
range_type range( size_type grainsize = 1 );

const_range_type range( size_type grainsize = 1 ) const;
```

Returns: a range object representing all elements in the container.

Non-member functions

These functions provide binary and lexicographical comparison and swap operations on `oneapi::tbb::concurrent_vector` objects.

The exact namespace where these functions are defined is unspecified, as long as they can be used in respective comparison operations. For example, an implementation can define the classes and functions in the same internal namespace and define `oneapi::tbb::concurrent_vector` as a type alias, for which the non-member functions are reachable only via argument-dependent lookup.

```
template <typename T, typename Allocator>
bool operator==( const concurrent_vector<T, Allocator>& lhs,
                 const concurrent_vector<T, Allocator>& rhs );

template <typename T, typename Allocator>
bool operator!=( const concurrent_vector<T, Allocator>& lhs,
                 const concurrent_vector<T, Allocator>& rhs );

template <typename T, typename Allocator>
bool operator<( const concurrent_vector<T, Allocator>& lhs,
                const concurrent_vector<T, Allocator>& rhs );

template <typename T, typename Allocator>
bool operator<=( const concurrent_vector<T, Allocator>& lhs,
                 const concurrent_vector<T, Allocator>& rhs );

template <typename T, typename Allocator>
bool operator>( const concurrent_vector<T, Allocator>& lhs,
                const concurrent_vector<T, Allocator>& rhs );

template <typename T, typename Allocator>
bool operator>=( const concurrent_vector<T, Allocator>& lhs,
                 const concurrent_vector<T, Allocator>& rhs );

template <typename T, typename Allocator>
```

(continues on next page)

(continued from previous page)

```
void swap( concurrent_vector<T, Allocator>& lhs,
           concurrent_vector<T, Allocator>& rhs );
```

Non-member binary comparisons

Two objects of `concurrent_vector` are equal if:

- they contains an equal number of elements.
- the elements on the same positions are equal.

```
template <typename T, typename Allocator>
bool operator==( const concurrent_vector<T, Allocator>& lhs,
                 const concurrent_vector<T, Allocator>& rhs );
```

Returns: `true` if `lhs` is equal to `rhs`, `false` otherwise.

```
template <typename T, typename Allocator>
bool operator!=( const concurrent_vector<T, Allocator>& lhs,
                 const concurrent_vector<T, Allocator>& rhs );
```

Returns: `true` if `lhs` is not equal to `rhs`, `false` otherwise.

Non-member lexicographical comparisons

```
template <typename T, typename Allocator>
bool operator<( const concurrent_vector<T, Allocator>& lhs,
                const concurrent_vector<T, Allocator>& rhs );
```

Returns: `true` if `lhs` is lexicographically *less* than `rhs`; `false`, otherwise.

```
template <typename T, typename Allocator>
bool operator<=( const concurrent_vector<T, Allocator>& lhs,
                 const concurrent_vector<T, Allocator>& rhs );
```

Returns: `true` if `lhs` is lexicographically *less or equal* than `rhs`; `false`, otherwise.

```
template <typename T, typename Allocator>
bool operator>( const concurrent_vector<T, Allocator>& lhs,
                const concurrent_vector<T, Allocator>& rhs );
```

Returns: `true` if `lhs` is lexicographically *greater* than `rhs`; `false`, otherwise.

```
template <typename T, typename Allocator>
bool operator>=( const concurrent_vector<T, Allocator>& lhs,
                 const concurrent_vector<T, Allocator>& rhs );
```

Returns: `true` if `lhs` is lexicographically *greater or equal* than `rhs`; `false`, otherwise.

Non-member swap

```
template <typename T, typename Allocator>
void swap( concurrent_vector<T, Allocator>& lhs,
           concurrent_vector<T, Allocator>& rhs );
```

Equivalent to `lhs.swap(rhs)`.

Other

Deduction guides

If possible, `concurrent_vector` constructors support class template argument deduction (since C++17). The following constructors provide implicitly-generated deduction guides:

- Copy and move constructors, including constructors with explicit `allocator_type` argument
- Constructors, accepting `std::initializer_list` as an argument

In addition, the following explicit deduction guide is provided:

```
template <typename InputIterator,
          typename Allocator = tbb::cache_aligned_allocator<iterator_value_t
↪<InputIterator>>>
concurrent_vector( InputIterator, InputIterator,
                   Allocator = Allocator() )
-> concurrent_vector<iterator_value_t<InputIterator>,
                     Allocator>;
```

Where type alias `iterator_value_t` defines as follows:

```
template <typename InputIterator>
using iterator_value_t = typename std::iterator_traits<InputIterator>::value_type;
```

This deduction guide only participate in the overload resolution if the following requirements are met:

- The `InputIterator` type meets the `InputIterator` requirements described in the [input.iterators] section of the ISO C++ Standard.
- The `Allocator` type meets the `Allocator` requirements described in the [allocator.requirements] section of the ISO C++ Standard.

Example

```
#include <oneapi/tbb/concurrent_vector.h>
#include <array>
#include <memory>

int main() {
    std::array<int, 100> arr;

    // Deduces cv1 as oneapi::tbb::concurrent_vector<int>
    oneapi::tbb::concurrent_vector cv1(arr.begin(), arr.end());

    std::allocator<int> alloc;
```

(continues on next page)

(continued from previous page)

```
    // Deduces cv2 as oneapi::tbb::concurrent_vector<int, std::allocator<int>>
    oneapi::tbb::concurrent_vector cv2(arr.begin(), arr.end(), alloc);
}
```

Queues

concurrent_queue

[containers.concurrent_queue]

`oneapi::tbb::concurrent_queue` is a class template for an unbounded first-in-first-out data structure that permits multiple threads to concurrently push and pop items.

Class Template Synopsis

```
// Defined in header <oneapi/tbb/concurrent_queue.h>

namespace oneapi {
    namespace tbb {

        template <typename T, typename Allocator = cache_aligned_allocator<T>>
        class concurrent_queue {
        public:
            using value_type = T;
            using reference = T&;
            using const_reference = const T&;
            using pointer = typename std::allocator_traits<Allocator>::pointer;
            using const_pointer = typename std::allocator_traits<Allocator>::const_
↪pointer;
            using allocator_type = Allocator;

            using size_type = <implementation-defined unsigned integer type>;
            using difference-type = <implementation-defined signed integer type>;

            using iterator = <implementation-defined ForwardIterator>;
            using const_iterator = <implementation-defined constant ForwardIterator>;

            // Construction, destruction, copying
            concurrent_queue();

            explicit concurrent_queue( const allocator_type& alloc );

            template <typename InputIterator>
            concurrent_queue( InputIterator first, InputIterator last,
                              const allocator_type& alloc = allocator_type() );

            concurrent_queue( std::initializer_list<value_type> init,
                              const allocator_type& alloc = allocator_type() );
```

(continues on next page)

(continued from previous page)

```
            concurrent_queue( const concurrent_queue& other );
            concurrent_queue( const concurrent_queue& other, const allocator_type& alloc␣
↪);

            concurrent_queue( concurrent_queue&& other );
            concurrent_queue( concurrent_queue&& other, const allocator_type& alloc );

            ~concurrent_queue();

            concurrent_queue& operator=( const concurrent_queue& other );
            concurrent_queue& operator=( concurrent_queue&& other );
            concurrent_queue& operator=( std::initializer_list<value_type> init );

            template <typename InputIterator>
            void assign( InputIterator first, InputIterator last );

            void assign( std::initializer_list<value_type> init );

            void swap( concurrent_queue& other );

            void push( const value_type& value );
            void push( value_type&& value );

            template <typename... Args>
            void emplace( Args&&... args );

            bool try_pop( value_type& result );

            allocator_type get_allocator() const;

            size_type unsafe_size() const;
            bool empty() const;

            void clear();

            iterator unsafe_begin();
            const_iterator unsafe_begin() const;
            const_iterator unsafe_cbegin() const;

            iterator unsafe_end();
            const_iterator unsafe_end() const;
            const_iterator unsafe_cend() const;
        }; // class concurrent_queue

    } // namespace tbb
} // namespace oneapi
```

Requirements:

- The type `T` must meet the `Erasable` requirements from the [container.requirements] ISO C++ Standard section. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Allocator` must meet the `Allocator` requirements from the [allocator.requirements] ISO C++ Stan-

dard section.

Member functions

Construction, destruction, copying

Empty container constructors

```
concurrent_queue();

explicit concurrent_queue( const allocator_type& alloc );
```

Constructs an empty `concurrent_queue`. If provided, uses the allocator `alloc` to allocate the memory.

Constructors from the sequence of elements

```
template <typename InputIterator>
concurrent_queue( InputIterator first, InputIterator last,
                  const allocator_type& alloc = allocator_type() );
```

Constructs a `concurrent_queue` containing all elements from the half-open interval `[first, last)` using the allocator `alloc` to allocate the memory.

Requirements: the type `InputIterator` must meet the *InputIterator* requirements from the `[input.iterators]` ISO C++ Standard section.

```
concurrent_queue( std::initializer_list<value_type> init,
                  const allocator_type& alloc = allocator_type() );
```

Equivalent to `concurrent_queue(init.begin(), init.end(), alloc)`.

Copying constructors

```
concurrent_queue( const concurrent_queue& other );

concurrent_queue( const concurrent_queue& other,
                  const allocator_type& alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by `std::allocator_traits<allocator_type>::select_on_contain get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Moving constructors

```
concurrent_queue( concurrent_queue&& other );

concurrent_queue( concurrent_queue&& other,
                  const allocator_type& alloc );
```

Constructs a `concurrent_queue` with the content of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Destructor

```
~concurrent_queue();
```

Destroys the `concurrent_queue`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

Assignment operators

```
concurrent_queue& operator=( const concurrent_queue& other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_assignment::`
is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_queue& operator=( concurrent_queue&& other );
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment:`
is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_queue& operator=( std::initializer_list<value_type> init );
```

Replaces all elements in `*this` by the elements in `init`.

The behavior is undefined in case of concurrent operations with `*this`.

Returns: a reference to `*this`.

assign

```
template <typename InputIterator>
void assign( InputIterator first, InputIterator last );
```

Replaces all elements in `*this` be the elements in the half-open interval `[first, last)`.

The behavior is undefined in case of concurrent operations with `*this`.

Requirements: the type `InputIterator` must meet the *InputIterator* requirements from the `[input.iterators]` ISO C++ Standard section.

```
void assign( std::initializer_list<value_type> init );
```

Equivalent to `assign(init.begin(), init.end())`.

Concurrently safe member functions

All member functions in this section can be performed concurrently with each other.

Pushing elements

```
void push( const value_type& value );
```

Pushes a copy of `value` into the container.

Requirements: the type `T` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
void push( value_type&& value );
```

Pushes `value` into the container using move semantics.

Requirements: the type `T` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

`value` is left in a valid, but unspecified state.

```
template <typename... Args>
void emplace( Args&&... args );
```

Pushes a new element constructed from `args` into the container.

Requirements: the type `T` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ Standard section.

Popping elements

```
bool try_pop( value_type& value );
```

If the container is empty, does nothing.

Otherwise, copies the last element from the container and assigns it to `value`. The popped element is destroyed.

Requirements: the type `T` must meet the `MoveAssignable` requirements from the [moveassignable] ISO C++ Standard section.

Returns: `true` if the element was popped; `false`, otherwise.

get_allocator

```
allocator_type get_allocator() const;
```

Returns: a copy of the allocator, associated with `*this`.

Concurrently unsafe member functions

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these methods with other (either concurrently safe) methods.

The number of elements

```
size_type unsafe_size() const;
```

Returns: the number of elements in the container.

```
bool empty() const;
```

Returns: `true` if the container is empty; `false`, otherwise.

clear

```
void clear();
```

Removes all elements from the container.

swap

```
void swap( concurrent_queue& other );
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`.

Otherwise if `get_allocator() != other.get_allocator()` the behavior is undefined.

Iterators

The types `concurrent_queue::iterator` and `concurrent_queue::const_iterator` meet the requirements of `ForwardIterator` from the [forward.iterators] ISO C++ Standard section.

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these methods with other (either concurrently safe) methods.

unsafe_begin and unsafe_cbegin

```
iterator unsafe_begin();

const_iterator unsafe_begin() const;

const_iterator unsafe_cbegin() const;
```

Returns: an iterator to the first element in the container.

unsafe_end and unsafe_cend

```
iterator unsafe_end();

const_iterator unsafe_end() const;

const_iterator unsafe_cend() const;
```

Returns: an iterator to the element that follows the last element in the container.

Non-member functions

These functions provides binary comparison and swap operations on `oneapi::tbb::concurrent_queue` objects.

The exact namespace where this function is defined is unspecified, as long as it may be used in respective operation. For example, an implementation may define the classes and functions in the same internal namespace and define `oneapi::tbb::concurrent_queue` as a type alias for which the non-member functions are reachable only via argument-dependent lookup.

```
template <typename T, typename Allocator>
void swap( concurrent_queue<T, Allocator>& lhs,
           concurrent_queue<T, Allocator>& rhs );

template <typename T, typename Allocator>
bool operator==( const concurrent_queue<T, Allocator>& lhs,
                 const concurrent_queue<T, Allocator>& rhs );

template <typename T, typename Allocator>
bool operator!=( const concurrent_queue<T, Allocator>& lhs,
                 const concurrent_queue<T, Allocator>& rhs );
```

Non-member swap

```
template <typename T, typename Allocator>
void swap( concurrent_queue<T, Allocator>& lhs,
           concurrent_queue<T, Allocator>& rhs );
```

Equivalent to `lhs.swap(rhs)`.

Non-member binary comparisons

```
template <typename T, typename Allocator>
bool operator==( const concurrent_queue<T, Allocator>& lhs,
                 const concurrent_queue<T, Allocator>& rhs );
```

Checks if `lhs` is equal to `rhs`, that is they have the same number of elements and `lhs` contains all elements from `rhs`.

Returns: `true` if `lhs` is equal to `rhs`; `false`, otherwise.

```
template <typename T, typename Allocator>
bool operator!=( const concurrent_queue<T, Allocator>& lhs,
                 const concurrent_queue<T, Allocator>& rhs );
```

Equivalent to `!(lhs == rhs)`.

Returns: `true` if `lhs` is not equal to `rhs`; `false`, otherwise.

Other

Deduction guides

If possible, `oneapi::tbb::concurrent_queue` constructors support class template argument deduction (since C++17). Copy and move constructors, including constructors with an explicit `allocator_type` argument, provide implicitly-generated deduction guides. In addition, the following explicit deduction guide is provided:

```
template <typename InputIterator,
          typename Allocator = tbb::cache_aligned_allocator<iterator_value_t
↪<InputIterator>>
concurrent_queue( InputIterator, InputIterator,
```

(continues on next page)

(continued from previous page)

```
                  Allocator = Allocator() )
-> concurrent_queue<iterator_value_t<InputIterator>,
                    Allocator>;
```

Where the type alias `iterator_value_t` is defined as follows:

```
template <typename InputIterator>
using iterator_value_t = typename std::iterator_traits<InputIterator>::value_type;
```

This deduction guide only participates in the overload resolution if the following requirements are met:

- The `InputIterator` type meets the `InputIterator` requirements described in the [input.iterators] section of the ISO C++ Standard.
- The `Allocator` type meets the `Allocator` requirements described in the [allocator.requirements] section of the ISO C++ Standard.

Example

```
#include <oneapi/tbb/concurrent_queue.h>
#include <vector>
#include <memory>

int main() {
    std::vector<int> vec;

    // Deduces cq1 as oneapi::tbb::concurrent_queue<int>
    oneapi::tbb::concurrent_queue cq1(vec.begin(), vec.end());

    // Deduces cq2 as oneapi::tbb::concurrent_queue<int, std::allocator<int>>
    oneapi::tbb::concurrent_queue cq2(vec.begin(), vec.end(), std::allocator<int>{})
}
```

concurrent_bounded_queue

[containers.concurrent_bounded_queue]

`oneapi::tbb::concurrent_bounded_queue` is a class template for a bounded first-in-first-out data structure that permits multiple threads to concurrently push and pop items.

Class Template Synopsis

```
// Defined in header <oneapi/tbb/concurrent_queue.h>

namespace oneapi {
    namespace tbb {

        template <typename T, typename Allocator = cache_aligned_allocator<T>>
        class concurrent_bounded_queue {
        public:
            using value_type = T;
            using reference = T&;
```

(continues on next page)

(continued from previous page)

```
            using const_reference = const T&;
            using pointer = typename std::allocator_traits<Allocator>::pointer;
            using const_pointer = typename std::allocator_traits<Allocator>::const_
↪pointer;

            using allocator_type = Allocator;

            using size_type = <implementation-defined signed integer type>;
            using difference_type = <implementation-defined signed integer type>;

            using iterator = <implementation-defined ForwardIterator>;
            using const_iterator = <implementation-defined constant ForwardIterator>;

            concurrent_bounded_queue();

            explicit concurrent_bounded_queue( const allocator_type& alloc );

            template <typename InputIterator>
            concurrent_bounded_queue( InputIterator first, InputIterator last,
                                      const allocator_type& alloc = allocator_type() );

            concurrent_bounded_queue( std::initializer_list<value_type> init,
                                      const allocator_type& alloc = allocator_type() );

            concurrent_bounded_queue( const concurrent_bounded_queue& other );
            concurrent_bounded_queue( const concurrent_bounded_queue& other,
                                      const allocator_type& alloc );

            concurrent_bounded_queue( concurrent_bounded_queue&& other );
            concurrent_bounded_queue( concurrent_bounded_queue&& other,
                                      const allocator_type& alloc );

            ~concurrent_bounded_queue();

            concurrent_bounded_queue& operator=( const concurrent_bounded_queue& other );
            concurrent_bounded_queue& operator=( concurrent_bounded_queue&& other );
            concurrent_bounded_queue& operator=( std::initializer_list<value_type> init␣
↪);

            template <typename InputIterator>
            void assign( InputIterator first, InputIterator last );

            void assign( std::initializer_list<value_type> init );

            void swap( concurrent_bounded_queue& other );

            allocator_type get_allocator() const;

            void push( const value_type& value );
            void push( value_type&& value );

            bool try_push( const value_type& value );
```

(continues on next page)

(continued from previous page)

```
            bool try_push( value_type&& value );

            template <typename... Args>
            void emplace( Args&&... args );

            template <typename... Args>
            bool try_emplace( Args&&... args );

            void pop( value_type& result );

            bool try_pop( value_type& result );

            void abort();

            size_type size() const;

            bool empty() const;

            size_type capacity() const;
            void set_capacity( size_type new_capacity );

            void clear();

            iterator unsafe_begin();
            const_iterator unsafe_begin() const;
            const_iterator unsafe_cbegin() const;

            iterator unsafe_end();
            const_iterator unsafe_end() const;
            const_iterator unsafe_cend() const;
        }; // class concurrent_bounded_queue

    } // namespace tbb
} // namespace oneapi
```

Requirements:

- The type `T` must meet the `Erasable` requirements from the [container.requirements] ISO C++ Standard section. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Allocator` must meet the `Allocator` requirements from the [allocator.requirements] ISO C++ Standard section.

Member functions

Construction, destruction, copying

Empty container constructors

```
concurrent_bounded_queue();
```

(continues on next page)

(continued from previous page)

```
explicit concurrent_bounded_queue( const allocator_type& alloc );
```

Constructs an empty `concurrent_bounded_queue` with an unbounded capacity. If provided, uses the allocator `alloc` to allocate the memory.

Constructors from the sequence of elements

```
template <typename InputIterator>
concurrent_bounded_queue( InputIterator first, InputIterator last,
                          const allocator_type& alloc = allocator_type() );
```

Constructs a `concurrent_bounded_queue` with an unbounded capacity and containing all elements from the half-open interval `[first, last)` using the allocator `alloc` to allocate the memory.

Requirements: the type `InputIterator` must meet the *InputIterator* requirements from the `[input.iterators]` ISO C++ Standard section.

```
concurrent_bounded_queue( std::initializer_list<value_type> init,
                          const allocator_type& alloc = allocator_type() );
```

Equivalent to `concurrent_bounded_queue(init.begin(), init.end(), alloc)`.

Copying constructors

```
concurrent_bounded_queue( const concurrent_bounded_queue& other );

concurrent_bounded_queue( const concurrent_bounded_queue& other,
                          const allocator_type& alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by `std::allocator_traits<allocator_type>::select_on_contain get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Moving constructors

```
concurrent_bounded_queue( concurrent_bounded_queue&& other );

concurrent_bounded_queue( concurrent_bounded_queue&& other,
                  const allocator_type& alloc );
```

Constructs a `concurrent_bounded_queue` with the content of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Destructor

```
~concurrent_bounded_queue();
```

Destroys the `concurrent_bounded_queue`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

Assignment operators

```
concurrent_bounded_queue& operator=( const concurrent_bounded_queue& other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_assignment::` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_bounded_queue& operator=( concurrent_bounded_queue&& other );
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment:` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_bounded_queue& operator=( std::initializer_list<value_type> init );
```

Replaces all elements in `*this` by the elements in `init`.

The behavior is undefined in case of concurrent operations with `*this`.

Returns: a reference to `*this`.

assign

```
template <typename InputIterator>
void assign( InputIterator first, InputIterator last );
```

Replaces all elements in `*this` be the elements in the half-open interval `[first, last)`.

The behavior is undefined in case of concurrent operations with `*this`.

Requirements: the type `InputIterator` must meet the *InputIterator* requirements from the `[input.iterators]` ISO C++ Standard section.

```
void assign( std::initializer_list<value_type> init );
```

Equivalent to `assign(init.begin(), init.end())`.

Concurrently safe member functions

All member functions in this section can be performed concurrently with each other.

Pushing elements

```
void push( const value_type& value );
```

Waits until the number of items in the queue is less than the capacity and pushes a copy of `value` into the container.

Requirements: the type `T` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
bool try_push( const value_type& value );
```

If the number of items in the queue is less than the capacity, pushes a copy of `value` into the container.

Requirements: the type `T` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

Returns: `true` if the item was pushed; `false`, otherwise.

```
void push( value_type&& value );
```

Waits until the number of items in the queue is less than `capacity()` and pushes `value` into the container using move semantics.

Requirements: the type `T` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

`value` is left in a valid, but unspecified state.

```
bool try_push( value_type&& value );
```

If the number of items in the queue is less than the capacity, pushes `value` into the container using move semantics.

Requirements: the type `T` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

`value` is left in a valid, but unspecified state.

Returns: `true` if the item was pushed; `false`, otherwise.

```
template <typename... Args>
void emplace( Args&&... args );
```

Waits until the number of items in the queue is less than `capacity()` and pushes a new element constructed from `args` into the container.

Requirements: the type `T` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ Standard section.

```
template <typename... Args>
bool try_emplace( Args&&... args );
```

If the number of items in the queue is less than the capacity, pushes a new element constructed from `args` into the container.

Requirements: the type `T` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ Standard section.

Returns: `true` if the item was pushed; `false`, otherwise.

Popping elements

```
void pop( value_type& value );
```

Waits until the item becomes available, copies it from the container, and assigns it to the `value`. The popped element is destroyed.

Requirements: the type `T` must meet the `MoveAssignable` requirements from the [moveassignable] ISO C++ Standard section.

```
bool try_pop( value_type& value );
```

If the container is empty, does nothing.

Otherwise, copies the last element from the container and assigns it to the `value`. The popped element is destroyed.

Requirements: the type `T` must meet the `MoveAssignable` requirements from the [moveassignable] ISO C++ Standard section.

Returns: `true` if the element was popped; `false`, otherwise.

abort

```
void abort();
```

Wakes up any threads that are waiting on the queue via `push`, `pop`, or `emplace` operations and raises the `oneapi::tbb::user_abort` exception on those threads.

Capacity of the queue

```
size_type capacity() const;
```

Returns: the maximum number of items that the queue can hold.

```
void set_capacity( size_type new_capacity ) const;
```

Sets the maximum number of items that the queue can hold to `new_capacity`.

get_allocator

```
allocator_type get_allocator() const;
```

Returns: a copy of the allocator, associated with `*this`.

Concurrently unsafe member functions

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these methods with other (either concurrently safe) methods.

The number of elements

```
size_type size() const;
```

Returns: the number of elements in the container.

```
bool empty() const;
```

Returns: `true` if the container is empty; `false`, otherwise.

clear

```
void clear();
```

Removes all elements from the container.

swap

```
void swap( concurrent_bounded_queue& other );
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`.

Otherwise if `get_allocator() != other.get_allocator()` the behavior is undefined.

Iterators

The types `concurrent_bounded_queue::iterator` and `concurrent_bounded_queue::const_iterator` meet the requirements of `ForwardIterator` from the [forward.iterators] ISO C++ Standard section.

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these methods with other (either concurrently safe) methods.

unsafe_begin and unsafe_cbegin

```
iterator unsafe_begin();

const_iterator unsafe_begin() const;

const_iterator unsafe_cbegin() const;
```

Returns: an iterator to the first element in the container.

unsafe_end and unsafe_cend

```
iterator unsafe_end();

const_iterator unsafe_end() const;

const_iterator unsafe_cend() const;
```

Returns: an iterator to the element that follows the last element in the container.

Non-member functions

These functions provides binary comparison and swap operations on `oneapi::tbb::concurrent_bounded_queue` objects.

The exact namespace where this function is defined is unspecified, as long as it may be used in respective operation. For example, an implementation may define the classes and functions in the same internal namespace and define `oneapi::tbb::concurrent_bounded_queue` as a type alias for which the non-member functions are reachable only via argument-dependent lookup.

```
template <typename T, typename Allocator>
void swap( concurrent_bounded_queue<T, Allocator>& lhs,
           concurrent_bounded_queue<T, Allocator>& rhs );

template <typename T, typename Allocator>
bool operator==( const concurrent_bounded_queue<T, Allocator>& lhs,
                 const concurrent_bounded_queue<T, Allocator>& rhs );

template <typename T, typename Allocator>
bool operator!=( const concurrent_bounded_queue<T, Allocator>& lhs,
                 const concurrent_bounded_queue<T, Allocator>& rhs );
```

Non-member swap

```
template <typename T, typename Allocator>
void swap( concurrent_bounded_queue<T, Allocator>& lhs,
           concurrent_bounded_queue<T, Allocator>& rhs );
```

Equivalent to `lhs.swap(rhs)`.

Non-member binary comparisons

```
template <typename T, typename Allocator>
bool operator==( const concurrent_bounded_queue<T, Allocator>& lhs,
                 const concurrent_bounded_queue<T, Allocator>& rhs );
```

Checks if `lhs` is equal to `rhs`, that is they have the same number of elements and `lhs` contains all elements from `rhs`.

Returns: `true` if `lhs` is equal to `rhs`; `false`, otherwise.

```
template <typename T, typename Allocator>
bool operator!=( const concurrent_bounded_queue<T, Allocator>& lhs,
                 const concurrent_bounded_queue<T, Allocator>& rhs );
```

Equivalent to `!(lhs == rhs)`.

Returns: `true` if `lhs` is not equal to `rhs`; `false`, otherwise.

Other

Deduction guides

If possible, `concurrent_bounded_queue` constructors support class template argument deduction (since C++17). Copy and move constructors, including constructors with an explicit `allocator_type` argument, provide implicitly-generated deduction guides. In addition, the following explicit deduction guide is provided:

```
template <typename InputIterator,
          typename Allocator = tbb::cache_aligned_allocator<iterator_value_t
↪<InputIterator>>
concurrent_bounded_queue( InputIterator, InputIterator,
                          Allocator = Allocator() )
-> concurrent_bounded_queue<iterator_value_t<InputIterator>,
                            Allocator>;
```

Where the type alias `iterator_value_t` is defined as follows:

```
template <typename InputIterator>
using iterator_value_t = typename std::iterator_traits<InputIterator>::value_type;
```

This deduction guides only participate in the overload resolution if the following requirements are met:

- The `InputIterator` type meets the `InputIterator` requirements described in the [input.iterators] section of the ISO C++ Standard.
- The `Allocator` type meets the `Allocator` requirements described in the [allocator.requirements] section of the ISO C++ Standard.

Example

```
#include <oneapi/tbb/concurrent_queue.h>
#include <vector>
#include <memory>

int main() {
    std::vector<int> vec;

    // Deduces cq1 as oneapi::tbb::concurrent_bounded_queue<int>
    oneapi::tbb::concurrent_bounded_queue cq1(vec.begin(), vec.end());

    // Deduces cq2 as oneapi::tbb::concurrent_bounded_queue<int, std::allocator<int>>
    oneapi::tbb::concurrent_bounded_queue cq2(vec.begin(), vec.end(), std::allocator<int>
↪{})
}
```

concurrent_priority_queue

[containers.concurrent_priority_queue]

`oneapi::tbb::concurrent_priority_queue` is a class template for an unbounded priority queue that permits multiple threads to concurrently push and pop items. Items are popped in a priority order.

Class Template Synopsis

```
namespace oneapi {
    namespace tbb {

        template <typename T, typename Compare = std::less<T>,
                  typename Allocator = cache_aligned_allocator<T>>
        class concurrent_priority_queue {
        public:
            using value_type = T;
            using reference = T&;
            using const_reference = const T&;
            using size_type = <implementation-defined unsigned integer type>;
            using difference_type = <implementation-defined signed integer type>;
            using allocator_type = Allocator;

            concurrent_priority_queue();
            explicit concurrent_priority_queue( const allocator_type& alloc );

            explicit concurrent_priority_queue( const Compare& compare,
                                                const allocator_type& alloc = allocator_
↪type() );

            explicit concurrent_priority_queue( size_type init_capacity, const allocator_
↪type& alloc = allocator_type() );

            explicit concurrent_priority_queue( size_type init_capacity, const Compare&␣
```

(continues on next page)

(continued from previous page)

```
↪compare,
                                                const allocator_type& alloc = allocator_
↪type() );

            template <typename InputIterator>
            concurrent_priority_queue( InputIterator first, InputIterator last,
                                       const allocator_type& alloc = allocator_type() );

            template <typename InputIterator>
            concurrent_priority_queue( InputIterator first, InputIterator last,
                                       const Compare& compare, const allocator_type&␣
↪alloc = allocator_type() );

            concurrent_priority_queue( std::initializer_list<value_type> init,
                                       const allocator_type& alloc = allocator_type() );

            concurrent_priority_queue( std::initializer_list<value_type> init,
                                       const Compare& compare, const allocator_type&␣
↪alloc = allocator_type() );

            concurrent_priority_queue( const concurrent_priority_queue& other );
            concurrent_priority_queue( const concurrent_priority_queue& other, const␣
↪allocator_type& alloc );

            concurrent_priority_queue( concurrent_priority_queue&& other );
            concurrent_priority_queue( concurrent_priority_queue&& other, const␣
↪allocator_type& alloc );

            ~concurrent_priority_queue();

            concurrent_priority_queue& operator=( const concurrent_priority_queue& other␣
↪);
            concurrent_priority_queue& operator=( concurrent_priority_queue&& other );
            concurrent_priority_queue& operator=( std::initializer_list<value_type> init␣
↪);

            template <typename InputIterator>
            void assign( InputIterator first, InputIterator last );

            void assign( std::initializer_list<value_type> init );

            void swap( concurrent_priority_queue& other );

            allocator_type get_allocator() const;

            void clear();

            bool empty() const;
            size_type size() const;

            void push( const value_type& value );
            void push( value_type&& value );
```

(continues on next page)

(continued from previous page)

```
            template <typename... Args>
            void emplace( Args&&... args );

            bool try_pop( value_type& value );
        }; // class concurrent_priority_queue

    }; // namespace tbb
} // namespace oneapi
```

Requirements:

- The type `T` must meet the `Erasable` requirements from [container.requirements] ISO C++ Standard section. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Compare` must meet the `Compare` requirements from [alg.sorting] ISO C++ Standard section.
- The type `Allocator` must meet the `Allocator` requirements from [allocator.requirements] ISO C++ Standard section.

Member functions

Construction, destruction, copying

Empty container constructors

```
concurrent_priority_queue();

explicit concurrent_priority_queue( const allocator_type& alloc );

explicit concurrent_priority_queue( const Compare& compare, const allocator_
↪type& alloc );
```

Constructs an empty `concurrent_priority_queue`. The initial capacity is unspecified. If provided, uses the predicate `compare` for priority comparisons and the allocator `alloc` to allocate the memory.

```
concurrent_priority_queue( size_type init_capacity,
                           const allocator_type& alloc = allocator_type() );

concurrent_priority_queue( size_type init_capacity,
                           const Compare& compare,
                           const allocator_type& alloc = allocator_type() );
```

Constructs an empty `concurrent_priority_queue` with the initial capacity `init_capacity`. If provided, uses the predicate `compare` for priority comparisons and the allocator `alloc` to allocate the memory.

Constructors from the sequence of elements

```
template <typename InputIterator>
concurrent_priority_queue( InputIterator first, InputIterator last,
                           const allocator_type& alloc = allocator_type() );

template <typename InputIterator>
concurrent_priority_queue( InputIterator first, InputIterator last,
                           const Compare& compare,
                           const allocator_type& alloc = allocator_type() );
```

Constructs a `concurrent_priority_queue` containing all elements from the half-open interval `[first, last)`.

If provided, uses the predicate `compare` for priority comparisons and the allocator `alloc` to allocate the memory.

Requirements: the type `InputIterator` must meet the *InputIterator* requirements from the `[input.iterators]` ISO C++ Standard section.

```
concurrent_priority_queue( std::initializer_list<value_type> init,
                           const allocator_type& alloc = allocator_type() );
```

Equivalent to `concurrent_priority_queue(init.begin(), init.end(), alloc)`.

```
concurrent_priority_queue( std::initializer_list<value_type> init,
                           const Compare& compare,
                           const allocator_type& alloc = allocator_type() );
```

Equivalent to `concurrent_priority_queue(init.begin(), init.end(), compare, alloc)`.

Copying constructors

```
concurrent_priority_queue( const concurrent_priority_queue& other );

concurrent_priority_queue( const concurrent_priority_queue& other,
                           const allocator_type& alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by `std::allocator_traits<allocator_type>::select_on_contain get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Moving constructors

```
concurrent_priority_queue( concurrent_priority_queue&& other );

concurrent_priority_queue( concurrent_priority_queue&& other,
                           const allocator_type& alloc );
```

Constructs a copy of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Destructor

```
~concurrent_priority_queue();
```

Destroys the `concurrent_priority_queue`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

Assignment operators

```
concurrent_priority_queue& operator=( const concurrent_priority_queue& other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_assignment::`
is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_priority_queue& operator=( concurrent_priority_queue&& other );
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment:`
is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_priority_queue& operator=( std::initializer_list<value_type> init );
```

Replaces all elements in `*this` by the elements in `init`.

The behavior is undefined in case of concurrent operations with `*this`.

Returns: a reference to `*this`.

assign

```
template <typename InputIterator>
void assign( InputIterator first, InputIterator last );
```

Replaces all elements in `*this` be the elements in the half-open interval `[first, last)`.

The behavior is undefined in case of concurrent operations with `*this`.

Requirements: the type `InputIterator` must meet the *InputIterator* requirements from the `[input.iterators]` ISO C++ Standard section.

```
void assign( std::initializer_list<value_type> init );
```

Equivalent to `assign(init.begin(), init.end())`.

Size and capacity

empty

```
bool empty() const;
```

Returns: `true` if the container is empty; `false`, otherwise.

The result may differ from the actual container state in case of pending concurrent `push` or `try_pop` operations.

size

```
size_type size() const;
```

Returns: the number of elements in the container.

The result may differ from the actual number of elements in case of pending concurrent `push` or `try_pop` operations.

Concurrently safe modifiers

All member functions in this section can be performed concurrently with each other.

Pushing elements

```
void push( const value_type& value );
```

Pushes a copy of `value` into the container.

Requirements: the type `T` must meet the `CopyInsertable` requirements from [container.requirements] and the `CopyAssignable` requirements from [copyassignable] ISO C++ Standard sections.

```
void push( value_type&& value );
```

Pushes `value` into the container using move semantics.

Requirements: the type `T` must meet the `MoveInsertable` requirements from [container.requirements] and the `MoveAssignable` requirements from [moveassignable] ISO C++ Standard sections.

`value` is left in a valid, but unspecified state.

```
template <typename... Args>
void emplace( Args&&... args );
```

Pushes a new element constructed from `args` into the container.

Requirements: the type `T` must meet the `EmplaceConstructible` requirements from [container.requirements] and the `MoveAssignable` requirements from [moveassignable] ISO C++ Standard sections.

Popping elements

```
bool try_pop( value_type& value )
```

If the container is empty, does nothing.

Otherwise, copies the highest priority element from the container and assigns it to `value`. The popped element is destroyed.

Requirements: the type `T` must meet the `MoveAssignable` requirements from the [moveassignable] ISO C++ Standard section.

Returns: `true` if the element was popped; `false`, otherwise.

Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these methods with other (either concurrently safe) methods.

clear

```
void clear();
```

Removes all elements from the container.

swap

```
void swap( concurrent_priority_queue& other );
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`.

Otherwise if `get_allocator() != other.get_allocator()` the behavior is undefined.

Non-member functions

These functions provides binary comparison and swap operations on `oneapi::tbb::concurrent_priority_queue` objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define `oneapi::tbb::concurrent_priority_queue` as a type alias for which the non-member functions are reachable only via argument-dependent lookup.

```
template <typename T, typename Compare, typename Allocator>
void swap( concurrent_priority_queue<T, Compare, Allocator>& lhs,
           concurrent_priority_queue<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator==( const concurrent_priority_queue<T, Compare, Allocator>& lhs,
                 const concurrent_priority_queue<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator!=( const concurrent_priority_queue<T, Compare, Allocator>& lhs,
                 const concurrent_priority_queue<T, Compare, Allocator>& rhs );
```

Non-member swap

```
template <typename T, typename Compare, typename Allocator>
void swap( concurrent_priority_queue<T, Compare, Allocator>& lhs,
           concurrent_priority_queue<T, Compare, Allocator>& rhs );
```

Equivalent to `lhs.swap(rhs)`.

Non-member binary comparisons

```
template <typename T, typename Compare, typename Allocator>
bool operator==( const concurrent_priority_queue<T, Compare, Allocator>& lhs,
                 const concurrent_priority_queue<T, Compare, Allocator>& rhs );
```

Checks if `lhs` is equal to `rhs`, that is they have the same number of elements and `lhs` contains all elements from `rhs` with the same priority.

Returns: `true` if `lhs` is equal to `rhs`; `false`, otherwise.

```
template <typename T, typename Compare, typename Allocator>
bool operator!=( const concurrent_priority_queue<T, Compare, Allocator>& lhs,
                 const concurrent_priority_queue<T, Compare, Allocator>& rhs );
```

Equivalent to `!(lhs == rhs)`.

Returns: `true` if `lhs` is not equal to `rhs`; `false`, otherwise.

Other

Deduction guides

If possible, `oneapi::tbb::concurrent_priority_queue` constructors support class template argument deduction (since C++17). Copy and move constructors, including constructors with an explicit `allocator_type` argument, provide implicitly-generated deduction guides. In addition, the following explicit deduction guides are provided:

```
template <typename InputIterator,
          typename Compare = std::less<iterator_value_t<InputIterator>>,
          typename Allocator = tbb::cache_aligned_allocator<iterator_value_t
↪<InputIterator>>>
concurrent_priority_queue( InputIterator, InputIterator,
                           Compare = Compare(),
                           Allocator = Allocator() )
-> concurrent_priority_queue<iterator_value_t<InputIterator>,
                             Compare,
                             Allocator>;

template <typename InputIterator,
          typename Allocator>
concurrent_priority_queue( InputIterator, InputIterator,
                           Allocator )
-> concurrent_priority_queue<iterator_value_t<InputIterator>,
                             std::less<iterator_value_t<InputIterator>>,
                             Allocator>;

template <typename T,
          typename Compare = std::less<T>,
          typename Allocator = tbb::cache_aligned_allocator<T>>
concurrent_priority_queue( std::initializer_list<T>,
                           Compare = Compare(),
                           Allocator = Allocator() )
-> concurrent_priority_queue<T,
                             Compare,
                             Allocator>;

template <typename T,
          typename Allocator>
concurrent_priority_queue( std::initializer_list<T>,
                           Allocator )
-> concurrent_priority_queue<T,
                             std::less<T>,
                             Allocator>;
```

Where the type alias `iterator_value_t` is defined as follows:

```
template <typename InputIterator>
using iterator_value_t = typename std::iterator_traits<InputIterator>::value_type;
```

These deduction guides only participate in the overload resolution if the following requirements are met:

- The `InputIterator` type meets the `InputIterator` requirements described in the [input.iterators] section of the ISO C++ Standard.

- The `Allocator` type meets the `Allocator` requirements described in the [allocator.requirements] section of the ISO C++ Standard.
- The `Compare` type does not meet the `Allocator` requirements.

Example

```
#include <oneapi/tbb/concurrent_priority_queue.h>
#include <vector>
#include <functional>

int main() {
    std::vector<int> vec;

    // Deduces cpq1 as oneapi::tbb::concurrent_priority_queue<int>
    oneapi::tbb::concurrent_priority_queue cpq1(vec.begin(), vec.end());

    // Deduces cpq2 as oneapi::tbb::concurrent_priority_queue<int, std::greater>
    oneapi::tbb::concurrent_priority_queue cpq2(vec.begin(), vec.end(), std::greater{});
}
```

Unordered associative containers

concurrent_hash_map

[containers.concurrent_hash_map]

`concurrent_hash_map` is a class template for an unordered associative container that holds key-value pairs with unique keys and supports concurrent insertion, lookup, and erasure.

Class Template Synopsis

```
// Defined in header <oneapi/tbb/concurrent_hash_map.h>

namespace oneapi {
    namespace tbb {

        template <typename Key, typename T,
                  typename HashCompare = tbb_hash_compare<Key>,
                  typename Allocator = tbb_allocator<std::pair<const Key, T>>>
        class concurrent_hash_map {
        public:
              using key_type = Key;
              using mapped_type = T;
              using value_type = std::pair<const Key, T>;

              using reference = value_type&;
              using const_reference = const value_type&;
              using pointer = typename std::allocator_traits<Allocator>
 ↪::pointer;
              using const_pointer = typename std::allocator_traits<Allocator>
 ↪::const_pointer;
```

(continues on next page)

(continued from previous page)

```
            using hash_compare_type = HashCompare;
            using allocator_type = Allocator;

            using size_type = <implementation-defined unsigned integer type>;
            using difference_type = <implementation-defined signed integer␣
↪type>;

            using iterator = <implementation-defined ForwardIterator>;
            using const_iterator = <implementation-defined constant␣
↪ForwardIterator>;

            using range_type = <implementation-defined ContainerRange>;
            using const_range_type = <implementation-defined constant␣
↪ContainerRange>;

            class accessor;
            class const_accessor;

            // Construction, destruction, copying
            concurrent_hash_map();

            explicit concurrent_hash_map( const hash_compare_type& compare,
                                          const allocator_type& alloc =␣
↪allocator_type() );

            explicit concurrent_hash_map( const allocator_type& alloc );

            concurrent_hash_map( size_type n, const hash_compare_type&␣
↪compare,
                                 const allocator_type& alloc = allocator_
↪type() );

            concurrent_hash_map( size_type n, const allocator_type& alloc =␣
↪allocator_type() );

            template <typename InputIterator>
            concurrent_hash_map( InputIterator first, InputIterator last,
                                 const hash_compare_type& compare,
                                 const allocator_type& alloc = allocator_
↪type() );

            template <typename InputIterator>
            concurrent_hash_map( InputIterator first, InputIterator last,
                                 const allocator_type& alloc = allocator_
↪type() );

            concurrent_hash_map( std::initializer_list<value_type> init,
                                 const hash_compare_type& compare = hash_
↪compare_type(),
                                 const allocator_type& alloc = allocator_
↪type() );
```

(continues on next page)

(continued from previous page)

```
            concurrent_hash_map( std::initializer_list<value_type> init,
                                 const allocator_type& alloc );

            concurrent_hash_map( const concurrent_hash_map& other );
            concurrent_hash_map( const concurrent_hash_map& other,
                                 const allocator_type& alloc );

            concurrent_hash_map( concurrent_hash_map&& other );
            concurrent_hash_map( concurrent_hash_map&& other,
                                 const allocator_type& alloc );

            ~concurrent_hash_map();

            concurrent_hash_map& operator=( const concurrent_hash_map& other␣
↪);
            concurrent_hash_map& operator=( concurrent_hash_map&& other );
            concurrent_hash_map& operator=( std::initializer_list<value_type>␣
↪init );

            allocator_type get_allocator() const;

            // Concurrently unsafe modifiers
            void clear();

            void swap( concurrent_hash_map& other );

            // Hash policy
            void rehash( size_type sz = 0 );
            size_type bucket_count() const;

            // Size and capacity
            size_type size() const;
            bool empty() const;
            size_type max_size() const;

            // Lookup
            bool find( const_accessor& result, const key_type& key ) const;
            bool find( accessor& result, const key_type& key );

            template <typename K>
            bool find( const_accessor& result, const K& key ) const;

            template <typename K>
            bool find( accessor& result, const K& key );

            size_type count( const key_type& key ) const;

            template <typename K>
            size_type count( const K& key ) const;

            // Modifiers
```

(continues on next page)

(continued from previous page)

```
            bool insert( const_accessor& result, const key_type& key );
            bool insert( accessor& result, const key_type& key );

            template <typename K>
            bool insert( const_accessor& result, const K& key );

            template <typename K>
            bool insert( accessor& result, const K& key );

            bool insert( const_accessor& result, const value_type& value );
            bool insert( accessor& result, const value_type& value );
            bool insert( const_accessor& result, value_type&& value );
            bool insert( accessor& result, value_type&& value );

            bool insert( const value_type& value );
            bool insert( value_type&& value );

            template <typename InputIterator>
            void insert( InputIterator first, InputIterator last );

            void insert( std::initializer_list<value_type> init );

            template <typename... Args>
            bool emplace( const_accessor& result, Args&&... args );

            template <typename... Args>
            bool emplace( accessor& result, Args&&... args );

            template <typename... Args>
            bool emplace( Args&&... args );

            bool erase( const key_type& key );

            template <typename K>
            bool erase( const K& key );

            bool erase( const_accessor& item_accessor );
            bool erase( accessor& item_accessor );

            // Iterators
            iterator begin();
            const_iterator begin() const;
            const_iterator cbegin() const;

            iterator end();
            const_iterator end() const;
            const_iterator cend() const;

            std::pair<iterator, iterator> equal_range( const key_type& key );
            std::pair<const_iterator, const_iterator> equal_range( const key_
↪type& key ) const;
```

(continues on next page)

(continued from previous page)

```
            template <typename K>
            std::pair<iterator, iterator> equal_range( const K& key );

            template <typename K>
            std::pair<const_iterator, const_iterator> equal_range( const K&␣
↪key ) const;

            // Parallel iteration
            range_type range( std::size_t grainsize = 1 );
            const_range_type range( std::size_t grainsize = 1 ) const;
        }; // class concurrent_hash_map

    } // namespace tbb
} // namespace oneapi
```

Requirements:

- The expression `std::allocator_type<Allocator>::destroy(m, val)`, where `m` is an object of the type `Allocator` and `val` is an object of type `value_type`, must be well-formed. Member functions can impose stricter requirements depending on the type of the operation.
- The type `HashCompare` must meet the *HashCompare requirements*.
- The type `Allocator` must meet the `Allocator` requirements from the [allocator.requirements] ISO C++ Standard section.

Member classes

accessor and const_accessor

Member classes `concurrent_hash_map::accessor` and `concurrent_hash_map::const_accessor` are called *accessors*. Accessors allow multiple threads to concurrently access the key-value pairs in `concurrent_hash_map`. An accessor is called *empty* if it does not point to any item.

`accessor` member class

Member class `concurrent_hash_map::accessor` provides read-write access to the key-value pair in `concurrent_hash_map`.

```
namespace oneapi {
    namespace tbb {

        template <typename Key, typename T, typename HashCompare, typename Allocator>
        class concurrent_hash_map<Key, T, HashCompare, Allocator>::accessor {
            using value_type = std::pair<const Key, T>;

            accessor();
            ~accessor();

            bool empty() const;
            value_type& operator*() const;
```

(continues on next page)

(continued from previous page)

```
            value_type* operator->() const;

            void release();
        }; // class accessor

    } // namespace tbb
} // namespace oneapi
```

const_accessor member class

Member class concurrent_hash_map::const_accessor provides read only access to the key-value pair in concurrent_hash_map.

```
namespace oneapi {
    namespace tbb {

        template <typename Key, typename T, typename HashCompare, typename Allocator>
        class concurrent_hash_map<Key, T, HashCompare, Allocator>::const_accessor {
            using value_type = const std::pair<const Key, T>;

            const_accessor();
            ~const_accessor();

            bool empty() const;
            value_type& operator*() const;
            value_type* operator->() const;

            void release();
        }; // class const_accessor

    } // namespace tbb
} // namespace oneapi
```

Member functions

Construction and destruction

```
accessor();

const_accessor();
```

Constructs an empty accessor.

```
~accessor();

~const_accessor();
```

Destroys the accessor. If *this is not empty, releases the ownership of the element.

Emptiness

```
bool empty() const;
```

Returns: `true` if the accessor is empty; `false`, otherwise.

Key-value pair access

```
value_type& operator*() const;
```

Returns: a reference to the key-value pair to which the accessor points.

The behavior is undefined if the accessor is empty.

```
value_type* operator->() const;
```

Returns: a pointer to the key-value pair to which the accessor points.

The behavior is undefined if the accessor is empty.

Releasing

```
void release();
```

If `*this` is not empty, releases the ownership of the element. `*this` becomes empty.

Member functions

Construction, destruction, copying

Empty container constructors

```
concurrent_hash_map();

explicit concurrent_hash_map( const hash_compare_type& compare,
                              const allocator_type& alloc = allocator_type() );

explicit concurrent_hash_map( const allocator_type& alloc );
```

Constructs an empty `concurrent_hash_map`. The initial number of buckets is unspecified.

If provided, uses the comparator `compare` to calculate hash codes and compare `key_type` objects for equality and the allocator `alloc` to allocate the memory.

```
concurrent_hash_map( size_type n, const hash_compare_type& compare,
                     const allocator_type& alloc = allocator_type() );

concurrent_hash_map( size_type n, const allocator_type& alloc = allocator_
↪type() );
```

Constructs an empty `concurrent_hash_map` with `n` preallocated buckets.

If provided, uses the comparator `compare` to calculate hash codes and compare `key_type` objects for equality and the allocator `alloc` to allocate the memory.

Constructors from the sequence of elements

```
template <typename InputIterator>
concurrent_hash_map( InputIterator first, InputIterator last,
                     const hash_compare_type& compare,
                     const allocator_type& alloc = allocator_type() );

template <typename InputIterator>
concurrent_hash_map( InputIterator first, InputIterator last,
                     const allocator_type& alloc = allocator_type() );
```

Constructs the `concurrent_hash_map` which contains the elements from the half-open interval `[first, last)`.

If the range `[first, last)` contains multiple elements with equal keys, it is unspecified which element would be inserted.

If provided, uses the comparator `compare` to calculate hash codes and compare `key_type` objects for equality and the allocator `alloc` to allocate the memory.

Requirements: the type `InputIterator` must meet the requirements of `InputIterator` from the [input.iterators] ISO C++ Standard section.

```
concurrent_hash_map( std::initializer_list<value_type> init,
                     const hash_compare_type& compare = hash_compare_type(),
                     const allocator_type& alloc = allocator_type() );
```

Equivalent to `concurrent_hash_map(init.begin(), init.end(), compare, alloc)`.

```
concurrent_hash_map( std::initializer_list<value_type> init,
                     const allocator_type& alloc );
```

Equivalent to `concurrent_hash_map(init.begin(), init.end(), alloc)`.

Copying constructors

```
concurrent_hash_map( const concurrent_hash_map& other );

concurrent_hash_map( const concurrent_hash_map& other,
                     const allocator_type& alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Moving constructors

```
concurrent_hash_map( concurrent_hash_map&& other );

concurrent_hash_map( concurrent_hash_map&& other,
                     const allocator_type& alloc );
```

Constructs a `concurrent_hash_map` with the content of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by calling `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Destructor

```
~concurrent_hash_map();
```

Destroys the `concurrent_hash_map`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

Assignment operators

```
concurrent_hash_map& operator=( const concurrent_hash_map& other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_assignment::` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_hash_map& operator=( concurrent_hash_map&& other );
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment:`
is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_hash_map& operator=( std::initializer_list<value_type> init );
```

Replaces all elements in `*this` by the elements in `init`.

If `init` contains multiple elements with equal keys, it is unspecified which element is inserted.

The behavior is undefined in case of concurrent operations with `*this`.

Returns: a reference to `*this`.

get_allocator

```
allocator_type get_allocator() const;
```

Returns: a copy of the allocator associated with `*this`.

Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

clear

```
void clear();
```

Removes all elements from the container.

swap

```
void swap( concurrent_hash_map& other );
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value`
is `true`.

Otherwise, if `get_allocator() != other.get_allocator()`, the behavior is undefined.

Hash policy

Rehashing

```
void rehash( size_type n = 0 );
```

If `n > 0`, sets the number of buckets to the value that is not less than `n`.

bucket_count

```
size_type bucket_count() const;
```

Returns: the number of buckets in the container.

Size and capacity

empty

```
bool empty() const;
```

Returns: `true` if the container is empty; `false`, otherwise.

The result may differ with the actual container state in case of pending concurrent insertions or erasures.

size

```
size_type size() const;
```

Returns: the number of elements in the container.

The result may differ with the actual container state in case of pending concurrent insertions or erasures.

max_size

```
size_type max_size() const;
```

Returns: The maximum number of elements that container can hold.

Lookup

All methods in this section can be executed concurrently with each other and concurrently-safe modifiers.

find

```
bool find( const_accessor& result, const key_type& key ) const;

bool find( accessor& result, const key_type& key );
```

If the `result` accessor is not empty, releases the `result`.

If an element with the key that is equivalent to `key` exists, sets the `result` to provide access to this element.

Returns: `true` if an element with the key equivalent to `key` is found; `false` otherwise.

```
template <typename K>
bool find( const_accessor& result, const K& key ) const;

template <typename K>
bool find( accessor& result, const K& key );
```

If the `result` accessor is not empty, releases the `result`.

If an element with the key that compares equivalent to the value `key` exists, sets the `result` to provide access to this element.

Returns: `true` if an element with the key that compares equivalent to the value `key` is found; `false` otherwise.

This overload only participates in the overload resolution if qualified-id `hash_compare_type::is_transparent` is valid and denotes a type.

count

```
size_type count( const key_type& key ) const;
```

Returns: 1 if an element with the key equivalent to `key` exists; `0` otherwise.

```
template <typename K>
size_type count( const K& key ) const;
```

Returns: 1 if an element with the key that compares equivalent to the value `key` exists; `0` otherwise.

This overload only participates in the overload resolution if qualified-id `hash_compare_type::is_transparent` is valid and denotes a type.

Concurrently safe modifiers

All methods in this section can be executed concurrently with each other and lookup methods.

Inserting values

```
bool insert( const_accessor& result, const key_type& key );

bool insert( accessor& result, const key_type& key );
```

If the `result` accessor is not empty, releases the `result` and attempts to insert the value constructed from `key, mapped_type()` into the container.

Sets the `result` to provide access to the inserted element or to the element with equal key, which was already presented in the container.

Requirements:

- the `value_type` type must meet the `EmplaceConstructible` requirements described in the [container.requirements] section of the ISO C++ Standard.
- the `mapped_type` type must meet the `DefaultConstructible` requirements described in the [defaultconstructible] section of the ISO C++ Standard.

Returns: `true` if an element is inserted; `false` otherwise.

```
template <typename K>
bool insert( const_accessor& result, const K& key );

template <typename K>
bool insert( accessor& result, const K& key );
```

If the `result` accessor is not empty, releases the `result` and attempts to insert the value constructed from `key, mapped_type()` into the container.

Sets the `result` to provide access to the inserted element or to the element with the key, that compares equivalent to the value `key`, which was already presented in the container.

This overload only participates in the overload resolution if:

- qualified-id `hash_compare_type::is_transparent` is valid and denotes a type
- `std::is_constructible<key_type, const K&>::value` is `true`

Requirements: the `mapped_type` type must meet the `DefaultConstructible` requirements described in the [defaultconstructible] section of the ISO C++ Standard.

Returns: `true` if an element is inserted; `false` otherwise.

```
bool insert( const_accessor& result, const value_type& value );

bool insert( accessor& result, const value_type& value );
```

If the `result` accessor is not empty, releases the `result` and attempts to insert the value `value` into the container.

Sets the `result` to provide access to the inserted element or to the element with equal key, which was already presented in the container.

Requirements: the `value_type` type must meet the `CopyInsertable` requirements described in the [container.requirements] section of the ISO C++ Standard.

Returns: `true` if an element is inserted; `false` otherwise.

```
bool insert( const value_type& value );
```

Attempts to insert the value `value` into the container.

Requirements: the `value_type` type must meet the `CopyInsertable` requirements described in the [container.requirements] section of the ISO C++ Standard.

Returns: `true` if an element is inserted; `false` otherwise.

```
bool insert( const_accessor& result, value_type&& value );

bool insert( accessor& result, value_type&& value );
```

If the `result` accessor is not empty, releases the `result` and attempts to insert the value `value` into the container using move semantics.

Sets the `result` to provide access to the inserted element or to the element with equal key, which was already presented in the container.

`value` is left in a valid, but unspecified state.

Requirements: the `value_type` type must meet the `MoveInsertable` requirements described in the [container.requirements] section of the ISO C++ Standard.

Returns: `true` if an element is inserted; `false` otherwise.

```
bool insert( value_type&& value );
```

Attempts to insert the value `value` into the container using move semantics.

Requirements: the `value_type` type must meet the `MoveInsertable` requirements described in the [container.requirements] section of the ISO C++ Standard.

Returns: `true` if an element is inserted; `false` otherwise.

Inserting sequences of elements

```
template <typename InputIterator>
void insert( InputIterator first, InputIterator last );
```

Attempts to insert all items from the half-open interval `[first, last)` into the container.

If the interval `[first, last)` contains multiple elements with equal keys, it is unspecified which element should be inserted.

Requirements: the `InputIterator` type must meet the requirements of *InputIterator* described in the `[input.iterators]` section of the ISO C++ Standard.

```
void insert( std::initializer_list<value_type> init );
```

Equivalent to `insert(init.begin(), init.end())`.

Emplacing elements

```
template <typename... Args>
bool emplace( const_accessor& result, Args&&... args );

template <typename... Args>
bool emplace( accessor& result, Args&&... args );
```

If the `result` accessor is not empty, releases the `result` and attempts to insert an element constructed in-place from `args` into the container.

Sets the `result` to provide access to the inserted element or to the element with equal key, which was already presented in the container.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements described in the [container.requirements] section of the ISO C++ Standard.

Returns: `true` if an element is inserted; `false` otherwise

```
template <typename... Args>
bool emplace( Args&&... args );
```

Attempts to insert an element constructed in-place from `args` into the container.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements described in the [container.requirements] section of the ISO C++ Standard.

Returns: `true` if an element is inserted; `false` otherwise

Erasing elements

```
bool erase( const key_type& key );
```

If an element with the key equivalent to `key` exists, removes it from the container.

Returns: `true` if an element is removed; `false` otherwise.

```
template <typename K>
bool erase( const K& key );
```

If an element with the key that compares equivalent to the value `key` exists, removes it from the container.

This overload only participates in the overload resolution if qualified-id `hash_compare_type::is_transparent` is valid and denotes a type.

Returns: `true` if an element is removed; `false` otherwise.

```
bool erase( const_accessor& item_accessor );
bool erase( accessor& item_accessor );
```

Removes an element owned by `item_accessor` from the container.

Requirements: `item_accessor` should not be empty.

Returns: `true` if an element is removed by the current thread; `false` if it is removed by another thread.

Iterators

The types `concurrent_hash_map::iterator` and `concurrent_hash_map::const_iterator` meet the requirements of `ForwardIterator` from the [forward.iterators] ISO C++ Standard section.

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

begin and cbegin

```
iterator begin();

const_iterator begin() const;

const_iterator cbegin() const;
```

Returns: an iterator to the first element in the container.

end and cend

```
iterator end();

const_iterator end() const;

const_iterator cend() const;
```

Returns: an iterator to the element that follows the last element in the container.

equal_range

```
std::pair<iterator, iterator> equal_range( const key_type& key );

std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;
```

Returns: a range containing an element that is equivalent to `key`. If there is no such element in the container, returns `{end(), end()}` .

```
template <typename K>
std::pair<iterator, iterator> equal_range( const K& key );

template <typename K>
std::pair<const_iterator, const_iterator> equal_range( const K& key ) const;
```

Returns: a range containing an element which compares equivalent to the value `key`. If there is no such element in the container, returns `{end(), end()}`.

This overload only participates in the overload resolution if qualified-id `hash_compare_type::is_transparent` is valid and denotes a type.

Parallel iteration

Member types `concurrent_hash_map::range_type` and `concurrent_hash_map::const_range_type` meet the *ContainerRange requirements*.

These types differ only in that the bounds for a `concurrent_hash_map::const_range_type` are of type `concurrent_hash_map::const_iterator`, whereas the bounds for a `concurrent_hash_map::range_type` are of type `concurrent_hash_map::iterator`.

Traversing the `concurrent_hash_map` is not thread safe. The behavior is undefined in case of concurrent execution of any member functions while traversing the `range_type` or `const_range_type`.

range member function

```
range_type range( std::size_t grainsize = 1 );

const_range_type range( std::size_t grainsize = 1 ) const;
```

Returns: a range object representing all elements in the container.

Non-member functions

These functions provide binary comparison and swap operations on `oneapi::tbb::concurrent_hash_map` objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define `oneapi::tbb::concurrent_hash_map` as a type alias for which the non-member functions are reachable only via argument-dependent lookup.

```
template <typename Key, typename T, typename HashCompare, typename Allocator>
bool operator==( const concurrent_hash_map<Key, T, HashCompare, Allocator>& lhs,
                 const concurrent_hash_map<Key, T, HashCompare, Allocator>& rhs );

template <typename Key, typename T, typename HashCompare, typename Allocator>
bool operator!=( const concurrent_hash_map<Key, T, HashCompare, Allocator>& lhs,
                 const concurrent_hash_map<Key, T, HashCompare, Allocator>& rhs );

template <typename Key, typename T, typename HashCompare, typename Allocator>
void swap( concurrent_hash_map<Key, T, HashCompare, Allocator>& lhs,
           concurrent_hash_map<Key, T, HashCompare, Allocator>& rhs );
```

Non-member swap

```
template <typename Key, typename T, typename HashCompare, typename Allocator>
void swap( concurrent_hash_map<Key, T, HashCompare, Allocator>& lhs,
           concurrent_hash_map<Key, T, HashCompare, Allocator>& rhs );
```

Equivalent to `lhs.swap(rhs)`.

Non-member binary comparisons

Two objects of `concurrent_hash_map` are equal if the following conditions are `true`:

- They contain equal number of elements.
- Each element from one container is also available in the other.

```
template <typename Key, typename T, typename HashCompare, typename Allocator>
bool operator==( const concurrent_hash_map<Key, T, HashCompare, Allocator>& lhs,
                 const concurrent_hash_map<Key, T, HashCompare, Allocator>& rhs );
```

Returns: `true` if `lhs` is equivalent to `rhs`; `false`, otherwise.

```
template <typename Key, typename T, typename HashCompare, typename Allocator>
bool operator!=( const concurrent_hash_map<Key, T, HashCompare, Allocator>& lhs,
                 const concurrent_hash_map<Key, T, HashCompare, Allocator>& rhs );
```

Equivalent to `!(lhs == rhs)`.

Returns: `true` if `lhs` is not equal to `rhs`; `false`, otherwise.

Other

Deduction guides

If possible, `concurrent_hash_map` constructors support class template argument deduction (since C++17). Copy and move constructors, including constructors with an explicit `allocator_type` argument, provide implicitly-generated deduction guides. In addition, the following explicit deduction guides are provided:

```
template <typename InputIterator,
          typename HashCompare = tbb_hash_compare<iterator_key_t<InputIterator>>,
          typename Allocator = tbb_allocator<iterator_alloc_value_t<InputIterator>>>
concurrent_hash_map( InputIterator, InputIterator,
                     HashCompare = HashCompare(),
                     Allocator = Allocator() )
-> concurrent_hash_map<iterator_key_t<InputIterator>,
                       iterator_mapped_t<InputIterator>,
                       HashCompare,
                       Allocator>;

template <typename InputIterator,
          typename Allocator>
```

(continues on next page)

(continued from previous page)

```
concurrent_hash_map( InputIterator, InputIterator, Allocator )
-> concurrent_hash_map<iterator_key_t<InputIterator>,
                       iterator_mapped_t<InputIterator>,
                       tbb_hash_compare<iterator_key_t<InputIterator>>,
                       Allocator>;

template <typename Key, typename T,
          typename HashCompare = tbb_hash_compare<std::remove_const_t<Key>>,
          typename Allocator = tbb_allocator<std::pair<const Key, T>>>
concurrent_hash_map( std::initializer_list<std::pair<Key, T>>,
                     HashCompare = HashCompare(),
                     Allocator = Allocator() )
-> concurrent_hash_map<std::remove_const_t<Key>,
                       T,
                       HashCompare,
                       Allocator>;

template <typename Key, typename T,
          typename Allocator>
concurrent_hash_map( std::initializer_list<std::pair<Key, T>>,
                     Allocator )
-> concurrent_hash_map<std::remove_const_t<Key>,
                       T,
                       tbb_hash_compare<std::remove_const_t<Key>>,
                       Allocator>;
```

Where the type aliases `iterator_key_t`, `iterator_mapped_t`, and `iterator_alloc_value_t` are defined as follows:

```
template <typename InputIterator>
using iterator_key_t = std::remove_const_t<typename std::iterator_traits<InputIterator>
↪::value_type::first_type>;

template <typename InputIterator>
using iterator_mapped_t = typename std::iterator_traits<InputIterator>::value_
↪type::second_type;

template <typename InputIterator>
using iterator_alloc_value_t = std::pair<std::add_const_t<iterator_key_t<InputIterator>,
                                         iterator_mapped_t<InputIterator>>>;
```

These deduction guides only participate in the overload resolution if the following requirements are met:

- The `InputIterator` type meets the `InputIterator` requirements described in the [input.iterators] section of the ISO C++ Standard.
- The `Allocator` type meets the `Allocator` requirements described in the [allocator.requirements] section of the ISO C++ Standard.
- The `HashCompare` type does not meet the `Allocator` requirements.

Example

```
#include <oneapi/tbb/concurrent_hash_map.h>
#include <vector>

int main() {
    std::vector<std::pair<const int, float>> v;

    // Deduces chmap1 as oneapi::tbb::concurrent_hash_map<int, float>
    oneapi::tbb::concurrent_hash_map chmap1(v.begin(), v.end());

    std::allocator<std::pair<const int, float>> alloc;
    // Deduces chmap2 as oneapi::tbb::concurrent_hash_map<int, float,
    //                                                    tbb_hash_compare<int>,
    //                                                    std::allocator<std::pair<const int, float>>>
    oneapi::tbb::concurrent_hash_map chmap2(v.begin(), v.end(), alloc);
}
```

concurrent_unordered_map

[containers.concurrent_unordered_map]

`oneapi::tbb::concurrent_unordered_map` is a class template that represents an unordered associative container. It stores key-value pairs with unique keys and supports concurrent insertion, lookup, and traversal, but does not support concurrent erasure.

Class Template Synopsis

```
// Defined in header <oneapi/tbb/concurrent_unordered_map.h>

namespace oneapi {
    namespace tbb {

        template <typename Key,
                  typename T,
                  typename Hash = std::hash<Key>,
                  typename KeyEqual = std::equal_to<Key>,
                  typename Allocator = tbb_allocator<std::pair<const Key, T>>>
        class concurrent_unordered_map {
        public:
            using key_type = Key;
            using mapped_type = T;
            using value_type = std::pair<const Key, T>;

            using size_type = <implementation-defined unsigned integer type>;
            using difference_type = <implementation-defined signed integer type>;

            using hasher = Hash;
            using key_equal = /*See below*/;

            using allocator_type = Allocator;
```

(continues on next page)

(continued from previous page)

```
            using reference = value_type&;
            using const_reference = const value_type&;

            using pointer = typename std::allocator_traits<Allocator>::pointer;
            using const_pointer = typename std::allocator_traits<Allocator>::const_
↪pointer;

            using iterator = <implementation-defined ForwardIterator>;
            using const_iterator = <implementation-defined constant ForwardIterator>;

            using local_iterator = <implementation-defined ForwardIterator>;
            using const_local_iterator = <implementation-defined constant␣
↪ForwardIterator>;

            using node_type = <implementation-defined node handle>;

            using range_type = <implementation-defined ContainerRange>;
            using const_range_type = <implementation-defined constant ContainerRange>;

            // Construction, destruction, copying
            concurrent_unordered_map();

            explicit concurrent_unordered_map( size_type bucket_count, const hasher&␣
↪hash = hasher(),
                                               const key_equal& equal = key_equal(),
                                               const allocator_type& alloc = allocator_
↪type() );

            concurrent_unordered_map( size_type bucket_count, const allocator_type&␣
↪alloc );

            concurrent_unordered_map( size_type bucket_count, const hasher& hash,
                                      const allocator_type& alloc );

            explicit concurrent_unordered_map( const allocator_type& alloc );

            template <typename InputIterator>
            concurrent_unordered_map( InputIterator first, InputIterator last,
                                      size_type bucket_count = /*implementation-defined*/
↪,
                                      const hasher& hash = hasher(),
                                      const key_equal& equal = key_equal(),
                                      const allocator_type& alloc = allocator_type() );

            template <typename Inputiterator>
            concurrent_unordered_map( InputIterator first, InputIterator last,
                                  size_type bucket_count, const allocator_type& alloc );

            template <typename InputIterator>
            concurrent_unordered_map( InputIterator first, InputIterator last,
                                      size_type bucket_count, const hasher& hash,
```

(continues on next page)

(continued from previous page)

```
                                    const allocator_type& alloc );

            concurrent_unordered_map( std::initializer_list<value_type> init,
                                    size_type bucket_count = /*implementation-defined*/
↪,
                                    const hasher& hash = hasher(),
                                    const key_equal& equal = key_equal(),
                                    const allocator_type& alloc = allocator_type() );

            concurrent_unordered_map( std::initializer_list<value_type> init,
                                    size_type bucket_count, const allocator_type&␣
↪alloc );

            concurrent_unordered_map( std::initializer_list<value_type> init,
                                    size_type bucket_count, const hasher& hash,
                                    const allocator_type& alloc );

            concurrent_unordered_map( const concurrent_unordered_map& other );
            concurrent_unordered_map( const concurrent_unordered_map& other,
                                    const allocator_type& alloc );

            concurrent_unordered_map( concurrent_unordered_map&& other );
            concurrent_unordered_map( concurrent_unordered_map&& other,
                                    const allocator_type& alloc );

            ~concurrent_unordered_map();

            concurrent_unordered_map& operator=( const concurrent_unordered_map& other );
            concurrent_unordered_map& operator=( concurrent_unordered_map&& other )␣
↪noexcept(/*See details*/);

            concurrent_unordered_map& operator=( std::initializer_list<value_type> init␣
↪);

            allocator_type get_allocator() const;

            // Iterators
            iterator begin() noexcept;
            const_iterator begin() const noexcept;
            const_iterator cbegin() const noexcept;

            iterator end() noexcept;
            const_iterator end() const noexcept;
            const_iterator cend() const noexcept;

            // Size and capacity
            bool empty() const noexcept;
            size_type size() const noexcept;
            size_type max_size() const noexcept;

            // Concurrently safe modifiers
            std::pair<iterator, bool> insert( const value_type& value );
```

(continues on next page)

(continued from previous page)

```
            iterator insert( const_iterator hint, const value_type& value );

            template <typename P>
            std::pair<iterator, bool> insert( P&& value );

            template <typename P>
            iterator insert( const_iterator hint, P&& value );

            std::pair<iterator, bool> insert( value_type&& value );
            iterator insert( const_iterator hint, value_type&& value );

            template <typename InputIterator>
            void insert( InputIterator first, InputIterator last );

            void insert( std::initializer_list<value_type> init );

            std::pair<iterator, bool> insert( node_type&& nh );
            iterator insert( const_iterator hint, node_type&& nh );

            template <typename... Args>
            std::pair<iterator, bool> emplace( Args&&... args );

            template <typename... Args>
            iterator emplace_hint( const_iterator hint, Args&&... args );

            template <typename SrcHash, typename SrcKeyEqual>
            void merge( concurrent_unordered_map<Key, T, SrcHash, SrcKeyEqual, Allocator>
↪& source );

            template <typename SrcHash, typename SrcKeyEqual>
            void merge( concurrent_unordered_map<Key, T, SrcHash, SrcKeyEqual, Allocator>
↪&& source );

            template <typename SrcHash, typename SrcKeyEqual>
            void merge( concurrent_unordered_multimap<Key, T, SrcHash, SrcKeyEqual,␣
↪Allocator>& source );

            template <typename SrcHash, typename SrcKeyEqual>
            void merge( concurrent_unordered_multimap<Key, T, SrcHash, SrcKeyEqual,␣
↪Allocator>&& source );

            // Concurrently unsafe modifiers
            void clear() noexcept;

            iterator unsafe_erase( const_iterator pos );
            iterator unsafe_erase( iterator pos );

            iterator unsafe_erase( const_iterator first, const_iterator last );

            size_type unsafe_erase( const key_type& key );

            template <typename K>
```

(continues on next page)

(continued from previous page)

```
            size_type unsafe_erase( const K& key );

            node_type unsafe_extract( const_iterator pos );
            node_type unsafe_extract( iterator pos );

            node_type unsafe_extract( const key_type& key );

            template <typename K>
            node_type unsafe_extract( const K& key );

            void swap( concurrent_unordered_map& other );

            // Element access
            mapped_type& at( const key_type& key );
            const mapped_type& at( const key_type& key ) const;

            mapped_type& operator[]( const key_type& key );
            mapped_type& operator[]( key_type&& key );

            // Lookup
            size_type count( const key_type& key ) const;

            template <typename K>
            size_type count( const K& key ) const;

            iterator find( const key_type& key );
            const_iterator find( const key_type& key ) const;

            template <typename K>
            iterator find( const K& key );

            template <typename K>
            const_iterator find( const K& key ) const;

            bool contains( const key_type& key ) const;

            template <typename K>
            bool contains( const K& key ) const;

            std::pair<iterator, iterator> equal_range( const key_type& key );
            std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;

            template <typename K>
            std::pair<iterator, iterator> equal_range( const K& key );

            template <typename K>
            std::pair<const_iterator, const_iterator> equal_range( const K& key ) const;

            // Bucket interface
            local_iterator unsafe_begin( size_type n );
            const_local_iterator unsafe_begin( size_type n ) const;
```

(continues on next page)

(continued from previous page)

```
            const_local_iterator unsafe_cbegin( size_type n ) const;

            local_iterator unsafe_end( size_type n );
            const_local_iterator unsafe_end( size_type n ) const;
            const_local_iterator unsafe_cend( size_type n ) const;

            size_type unsafe_bucket_count() const;
            size_type unsafe_max_bucket_bount() const;

            size_type unsafe_bucket_size( size_type n ) const;

            size_type unsafe_bucket( const key_type& key ) const;

            // Hash policy
            float load_factor() const;

            float max_load_factor() const;
            void max_load_factor( float ml );

            void rehash( size_type count );

            void reserve( size_type count );

            // Observers
            hasher hash_function() const;
            key_equal key_eq() const;

            // Parallel iteration
            range_type range();
            const_range_type range() const;
        }; // class concurrent_unordered_map

    } // namespace tbb
} // namespace oneapi
```

Requirements:

- The expression `std::allocator_type<Allocator>::destroy(m, val)`, where `m` is an object of the type `Allocator` and `val` is an object of type `value_type`, must be well-formed. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Hash` must meet the `Hash` requirements from the [hash] ISO C++ Standard section.
- The type `KeyEqual` must meet the `BinaryPredicate` requirements from the [algorithms.general] ISO C++ Standard section.
- The type `Allocator` must meet the `Allocator` requirements from the [allocator.requirements] ISO C++ Standard section.

Description

`oneapi::tbb::concurrent_unordered_map` is an unordered associative container, which elements are organized into buckets. The value of the hash function `Hash` for a `Key` object determines the number of the bucket in which the corresponding element will be placed.

If the qualified-id `Hash::transparent_key_equal` is valid and denotes a type, the member type `concurrent_unordered_map::key_equal` is defined as the value of this qualified-id. In this case, the program is ill-formed if any of the following conditions are met:

- The template parameter `KeyEqual` is different from `std::equal_to<Key>`.
- Qualified-id `Hash::transparent_key_equal::is_transparent` is not valid or does not denote a type.

Otherwise, the member type `concurrent_unordered_map::key_equal` is defined as the value of the template parameter `KeyEqual`.

Member functions

Construction, destruction, copying

Empty container constructors

```
concurrent_unordered_map();

explicit concurrent_unordered_map( const allocator_type& alloc );
```

Constructs an empty `concurrent_unordered_map`. The initial number of buckets is unspecified.

If provided, uses the allocator `alloc` to allocate the memory.

```
explicit concurrent_unordered_map( size_type bucket_count,
                                   const hasher& hash = hasher(),
                                   const key_equal& equal = key_equal(),
                                   const allocator_type& alloc = allocator_
↪type() );

concurrent_unordered_map( size_type bucket_count, const allocator_type& alloc␣
↪);

concurrent_unordered_map( size_type bucket_count, const hasher& hash,
                          const allocator_type& alloc );
```

Constructs an empty `concurrent_unordered_map` with `bucket_count` buckets.

If provided, uses the hash function `hasher`, predicate `equal` to compare `key_type` objects for equality, and the allocator `alloc` to allocate the memory.

Constructors from the sequence of elements

```
template <typename InputIterator>
concurrent_unordered_map( InputIterator first, InputIterator last,
                          size_type bucket_count = /*implementation-defined*/,
                          const hasher& hash = hasher(),
                          const key_equal& equal = key_equal(),
                          const allocator_type& alloc = allocator_type() );

template <typename Inputiterator>
concurrent_unordered_map( InputIterator first, InputIterator last,
                          size_type bucket_count, const allocator_type& alloc␣
↪);

template <typename InputIterator>
concurrent_unordered_map( InputIterator first, InputIterator last,
                          size_type bucket_count, const hasher& hash,
                          const allocator_type& alloc );
```

Constructs the `concurrent_unordered_map` that contains the elements from the half-open interval `[first, last)`.

If the range `[first, last)` contains multiple elements with equal keys, it is unspecified which element would be inserted.

If provided, uses the hash function `hasher`, predicate `equal` to compare `key_type` objects for equality, and the allocator `alloc` to allocate the memory.

Requirements: the type `InputIterator` must meet the requirements of `InputIterator` from the [input.iterators] ISO C++ Standard section.

```
concurrent_unordered_map( std::initializer_list<value_type> init,
                          size_type bucket_count = /*implementation-defined*/,
                          const hasher& hash = hasher(),
                          const key_equal& equal = key_equal(),
                          const allocator_type& alloc = allocator_type() );
```

Equivalent to `concurrent_unordered_map(init.begin(), init.end(), bucket_count, hash, equal, alloc)`.

```
concurrent_unordered_map( std::initializer_list<value_type> init,
                          size_type bucket_count, const allocator_type& alloc␣
↪);
```

Equivalent to `concurrent_unordered_map(init.begin(), init.end(), bucket_count, alloc)`.

```
concurrent_unordered_map( std::initializer_list<value_type> init,
                          size_type bucket_count, const hasher& hash,
                          const allocator_type& alloc );
```

Equivalent to `concurrent_unordered_map(init.begin(), init.end(), bucket_count, hash, alloc)`.

Copying constructors

```
concurrent_unordered_map( const concurrent_unordered_map& other );

concurrent_unordered_map( const concurrent_unordered_map& other,
                          const allocator_type& alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Moving constructors

```
concurrent_unordered_map( concurrent_unordered_map&& other );

concurrent_unordered_map( concurrent_unordered_map&& other,
                          const allocator_type& alloc );
```

Constructs a `concurrent_unordered_map` with the contents of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by calling `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Destructor

```
~concurrent_unordered_map();
```

Destroys the `concurrent_unordered_map`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

Assignment operators

```
concurrent_unordered_map& operator=( const concurrent_unordered_map& other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_assignment::` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_unordered_map& operator=( concurrent_unordered_map&& other )␣
↪noexcept(/*See below*/);
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment:` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

Exceptions: `noexcept` specification:

```
noexcept(std::allocator_traits<allocator_type>::is_always_equal::value␣
↪&&
         std::is_nothrow_move_assignable<hasher>::value &&
         std::is_nothrow_move_assignable<key_equal>::value)
```

```
concurrent_unordered_map& operator=( std::initializer_list<value_type> init );
```

Replaces all elements in `*this` by the elements in `init`.

If `init` contains multiple elements with equal keys, it is unspecified which element would be inserted.

The behavior is undefined in case of concurrent operations with `*this`.

Returns: a reference to `*this`.

Iterators

The types `concurrent_unordered_map::iterator` and `concurrent_unordered_map::const_iterator` meet the requirements of `ForwardIterator` from the [forward.iterators] ISO C++ Standard section.

begin and cbegin

```
iterator begin();

const_iterator begin() const;

const_iterator cbegin() const;
```

Returns: an iterator to the first element in the container.

end and cend

```
iterator end();

const_iterator end() const;

const_iterator cend() const;
```

Returns: an iterator to the element that follows the last element in the container.

Size and capacity

empty

```
bool empty() const;
```

Returns: `true` if the container is empty; `false`, otherwise.

The result may differ from the actual container state in case of pending concurrent insertions.

size

```
size_type size() const;
```

Returns: the number of elements in the container.

The result may differ from the actual container size in case of pending concurrent insertions.

max_size

```
size_type max_size() const;
```

Returns: the maximum number of elements that container can hold.

Concurrently safe modifiers

All member functions in this section can be performed concurrently with each other, lookup methods and while traversing the container.

Emplacing elements

```
template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args );
```

Attempts to insert an element constructed in-place from `args` into the container.

Returns: `std::pair<iterator, bool>` where `iterator` points to the inserted element or to an existing element with equal key. Boolean value is `true` if insertion took place; `false`, otherwise.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ Standard section.

```
template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args );
```

Attempts to insert an element constructed in-place from `args` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

Returns: an `iterator` to the inserted element or to an existing element with equal key.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ Standard section.

Inserting values

```
std::pair<iterator, bool> insert( const value_type& value );
```

Attempts to insert `value` into the container.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element or to an existing element with equal key. Boolean value is `true` if insertion took place; `false`, otherwise.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, const value_type& value );
```

Attempts to insert `value` into the container.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

Returns: an `iterator` to the inserted element or to an existing element with equal key.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
template <typename P>
std::pair<iterator, bool> insert( P&& value );
```

Equivalent to `emplace(std::forward<P>(value))`.

This overload only participates in overload resolution if `std::is_constructible<value_type, P&&>::value` is `true`.

```
template <typename P>
iterator insert( const_iterator hint, P&& value );
```

Equivalent to `emplace_hint(hint, std::forward<P>(value))`.

This overload only participates in overload resolution if `std::is_constructible<value_type, P&&>::value` is `true`.

```
std::pair<iterator, bool> insert( value_type&& value );
```

Attempts to insert `value` into the container using move semantics.

`value` is left in a valid, but unspecified state.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element or to an existing element with equal key. Boolean value is `true` if insertion took place; `false`, otherwise.

Requirements: the type `value_type` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, value_type&& other );
```

Attempts to insert `value` into the container using move semantics.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

`value` is left in a valid, but unspecified state.

Returns: an `iterator` to the inserted element or to an existing element with equal key.

Requirements: the type `value_type` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

Inserting sequences of elements

```
template <typename InputIterator>
void insert( InputIterator first, InputIterator last );
```

Attempts to insert all items from the half-open interval `[first, last)` into the container.

If the interval `[first, last)` contains multiple elements with equal keys, it is unspecified which element should be inserted.

Requirements: the type `InputIterator` must meet the requirements of *InputIterator* from the `[input.iterators]` ISO C++ Standard section.

```
void insert( std::initializer_list<value_type> init );
```

Equivalent to `insert(init.begin(), init.end())`.

Inserting nodes

```
std::pair<iterator, bool> insert( node_type&& nh );
```

If the node handle `nh` is empty, does nothing.

Otherwise, attempts to insert the node owned by `nh` into the container.

If the insertion fails, node handle `nh` keeps ownership of the node.

Otherwise, `nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element or to an existing element with key equal to `nh.key()`. Boolean value is `true` if insertion took place; `false`, otherwise.

```
iterator insert( const_iterator hint, node_type&& nh );
```

If the node handle `nh` is empty, does nothing.

Otherwise, attempts to insert the node owned by `nh` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

If the insertion fails, node handle `nh` keeps ownership of the node.

Otherwise, `nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: an iterator pointing to the inserted element or to an existing element with key equal to `nh.key()`.

Merging containers

```
template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_map<Key, T, SrcHash, SrcKeyEqual, Allocator>&␣
↪source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_map<Key, T, SrcHash, SrcKeyEqual, Allocator>&&
↪ source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_multimap<Key, T, SrcHash, SrcKeyEqual,␣
↪Allocator>& source );
```

(continues on next page)

(continued from previous page)

```
template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_multimap<Key, T, SrcHash, SrcKeyEqual,␣
↪Allocator>&& source );
```

Transfers those elements from `source` which keys do not exist in the container.

In case of merging with the container with multiple elements with equal keys, it is unspecified which element would be transferred.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `get_allocator() != source.get_allocator()`.

Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

Clearing

```
void clear();
```

Removes all elements from the container.

Erasing elements

```
iterator unsafe_erase( const_iterator pos );

iterator unsafe_erase( iterator pos );
```

Removes the element pointed to by `pos` from the container.

Invalidates all iterators and references to the removed element.

Returns: `iterator` that follows the removed element.

Requirements: the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
size_type unsafe_erase( const key_type& key );
```

Removes the element with the key equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed element.

Returns: 1 if an element with the key equivalent to `key` exists; `0`, otherwise.

```
template <typename K>
size_type unsafe_erase( const K& key );
```

Removes the element with the key equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed element.

This overload only participates in overload resolution if all of the following conditions are met:

- The qualified-id `hasher::transparent_key_equal` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: 1 if an element with the key equivalent to `key` exists; `0`, otherwise.

Erasing sequences

```
iterator unsafe_erase( const_iterator first, const_iterator last );
```

Removes all elements from the half-open interval `[first, last)` from the container.

Returns: `iterator` that follows the last removed element.

Requirements: the range `[first, last)` must be a valid subrange in `*this`.

Extracting nodes

```
node_type unsafe_extract( iterator pos );

node_type unsafe_extract( const_iterator pos );
```

Transfers ownership of the element pointed to by `pos` from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element.

Requirements: the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
node_type unsafe_extract( const key_type& key );
```

If an element with the key equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element or an empty node handle if an element with the key equivalent to `key` was not found.

```
template <typename K>
node_type unsafe_extract( const K& key );
```

If an element with the key equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

This overload only participates in overload resolution if all of the following conditions are met:

- The qualified-id `hasher::transparent_key_equal` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: the node handle that owns the extracted element or an empty node handle if an element with the key equivalent to `key` was not found.

swap

```
void swap( concurrent_unordered_map& other ) noexcept(/*See below*/);
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`.

Otherwise, if `get_allocator() != other.get_allocator()`, the behavior is undefined.

Exceptions: `noexcept` specification:

```
noexcept(std::allocator_traits<allocator_type>::is_always_equal::value␣
↪&&
         std::is_nothrow_swappable<hasher>::value &&
         std::is_nothrow_swappable<key_equal>::value
```

Element access

at

```
value_type& at( const key_type& key );

const value_type& at( const key_type& key ) const;
```

Returns: a reference to `item.second`, where `item` is the element with the key equivalent to `key`.

Throws: `std::out_of_range` exception if the element with the key equivalent to `key` is not presented in the container.

operator[]

```
value_type& operator[]( const key_type& key );
```

If the element with the key equivalent to `key` is not presented in the container, inserts a new element constructed in-place from `std::piecewise_construct, std::forward_as_tuple(key), std::tuple<>()`.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ Standard section.

Returns: a reference to `item.second`, where `item` is the element with the key equivalent to `key`.

```
value_type& operator[]( key_type&& key );
```

If the element with the key equivalent to `key` is not presented in the container, inserts a new element constructed in-place from `std::piecewise_construct, std::forward_as_tuple(std::move(key)), std::tuple<>()`.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ Standard section.

Returns: a reference to `item.second` where `item` is the element with the key equivalent to `key`.

Lookup

All methods in this section can be executed concurrently with each other, concurrently-safe modifiers and while traversing the container.

count

```
size_type count( const key_type& key );
```

Returns: the number of elements with the key equivalent to `key`.

```
template <typename K>
size_type count( const K& key );
```

Returns: the number of elements with the key that is equivalent to `key`.

This overload only participates in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

find

```
iterator find( const key_type& key );

const_iterator find( const key_type& key ) const;
```

Returns: an iterator to the element with the key equivalent to `key`, or `end()` if no such element exists.

```
template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;
```

Returns: an iterator to the element with the key that is equivalent to `key`, or `end()` if no such element exists.

These overloads only participate in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

contains

```
bool contains( const key_type& key ) const;
```

Returns: `true` if an element with the key equivalent to `key` exists in the container; `false`, otherwise.

```
template <typename K>
bool contains( const K& key ) const;
```

Returns: `true` if an element with the key equivalent to `key` exists in the container; `false`, otherwise.

This overload only participates in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

equal_range

```
std::pair<iterator, iterator> equal_range( const key_type& key );

std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;
```

Returns: if an element with the key equivalent to `key` exists, a pair of iterators `{f, l}`, where `f` is an iterator to this element, `l` is `std::next(f)`. Otherwise, `{end(), end()}`.

```
template <typename K>
std::pair<iterator, iterator> equal_range( const K& key )

template <typename K>
std::pair<const_iterator, const_iterator> equal_range( const K& key )
```

Returns: if an element with the key equivalent to `key` exists, a pair of iterators `{f, l}`, where `f` is an iterator to this element, `l` is `std::next(f)`. Otherwise, `{end(), end()}`.

These overloads only participate in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

Bucket interface

The types `concurrent_unordered_map::local_iterator` and `concurrent_unordered_map::const_local_iterator` meet the requirements of `ForwardIterator` from the [forward.iterators] ISO C++ Standard section.

Use these iterators to traverse a certain bucket.

All methods in this section can only be executed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

Bucket begin and bucket end

```
local_iterator unsafe_begin( size_type n );

const_local_iterator unsafe_begin( size_type n ) const;

const_local_iterator unsafe_cbegin( size_type n ) const;
```

Returns: an iterator to the first element in the bucket number `n`.

```
local_iterator unsafe_end( size_type n );

const_local_iterator unsafe_end( size_type n ) const;

const_local_iterator unsafe_cend( size_type n ) const;
```

Returns: an iterator to the element that follows the last element in the bucket number `n`.

The number of buckets

```
size_type unsafe_bucket_count() const;
```

Returns: the number of buckets in the container.

```
size_type unsafe_max_bucket_count() const;
```

Returns: the maximum number of buckets that container can hold.

Size of the bucket

```
size_type unsafe_bucket_size( size_type n ) const;
```

Returns: the number of elements in the bucket number `n`.

Bucket number

```
size_type unsafe_bucket( const key_type& key ) const;
```

Returns: the number of the bucket in which the element with the key `key` is stored.

Hash policy

Hash policy of `concurrent_unordered_map` manages the number of buckets in the container and the allowed maximum number of elements per bucket (load factor). If the maximum load factor is exceeded, the container can automatically increase the number of buckets.

Load factor

```
float load_factor() const;
```

Returns: the average number of elements per bucket, which is `size()/unsafe_bucket_count()`.

```
float max_load_factor() const;
```

Returns: the maximum number of elements per bucket.

```
void max_load_factor( float ml );
```

Sets the maximum number of elements per bucket to `ml`.

Manual rehashing

```
void rehash( size_type n );
```

Sets the number of buckets to `n` and rehashes the container.

```
void reserve( size_type n );
```

Sets the number of buckets to the value that is needed to store `n` elements.

Observers

get_allocator

```
allocator_type get_allocator() const;
```

Returns: a copy of the allocator associated with `*this`.

hash_function

```
hasher hash_function() const;
```

Returns: a copy of the hash function associated with `*this`.

key_eq

```
key_equal key_eq() const;
```

Returns: a copy of the key equality predicate associated with `*this`.

Parallel iteration

Member types `concurrent_unordered_map::range_type` and `concurrent_unordered_map::const_range_type` meet the *ContainerRange requirements*.

These types differ only in that the bounds for a `concurrent_unordered_map::const_range_type` are of type `concurrent_unordered_map::const_iterator`, whereas the bounds for a `concurrent_unordered_map::range_type` are of type `concurrent_unordered_map::iterator`.

range member function

```
range_type range();

const_range_type range() const;
```

Returns: a range object representing all elements in the container.

Non-member functions

These functions provide binary comparison and swap operations on `oneapi::tbb::concurrent_unordered_map` objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define `oneapi::tbb::concurrent_unordered_map` as a type alias for which the non-member functions are reachable only via argument-dependent lookup.

```
template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
void swap( concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>& lhs,
           concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>& rhs );

template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>& lhs,
                 const concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>& rhs␣
↪);

template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>& lhs,
                 const concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>& rhs␣
↪);
```

Non-member swap

```
template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
void swap( concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>& lhs,
           concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>& rhs )␣
↪noexcept(noexcept(lhs.swap(rhs)));
```

Equivalent to `lhs.swap(rhs)`.

Non-member binary comparisons

Two objects of `concurrent_unordered_map` are equal if the following conditions are `true`:

- They contains an equal number of elements.
- Each element from the one container is also available in the other.

```
template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>& lhs,
                 const concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>& rhs␣
↪);
```

Returns: `true` if `lhs` is equal to `rhs`; `false`, otherwise.

```
template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator!=( const concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>& lhs,
                 const concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>& rhs␣
↪);
```

Equivalent to `!(lhs == rhs)`.

Returns: `true` if `lhs` is not equal to `rhs`; `false`, otherwise.

Other

Deduction guides

If possible, `concurrent_unordered_map` constructors support class template argument deduction (since C++17). Copy and move constructors, including constructors with an explicit `allocator_type` argument, provide implicitly-generated deduction guides. In addition, the following explicit deduction guides are provided:

```
template <typename InputIterator,
          typename Hash = std::hash<iterator_key_t<InputIterator>>,
          typename KeyEqual = std::equal_to<iterator_key_t<InputIterator>>,
          typename Allocator = tbb::tbb_allocator<iterator_alloc_value_t<InputIterator>>>
concurrent_unordered_map( InputIterator, InputIterator,
                          map_size_type = {},
                          Hash = Hash(),
                          KeyEqual = KeyEqual(),
                          Allocator = Allocator() )
-> concurrent_unordered_map<iterator_key_t<InputIterator>,
                            iterator_mapped_t<InputIterator>,
                            Hash,
                            KeyEqual,
                            Allocator>;

template <typename InputIterator,
          typename Hash,
          typename Allocator>
concurrent_unordered_map( InputIterator, InputIterator,
                          map_size_type,
                          Hash,
                          Allocator )
-> concurrent_unordered_map<iterator_key_t<InputIterator>,
                            iterator_mapped_t<InputIterator>,
                            Hash,
                            std::equal_to<iterator_key_t<InputIterator>>,
                            Allocator>;

template <typename InputIterator,
          typename Allocator>
concurrent_unordered_map( InputIterator, InputIterator,
                          map_size_type,
                          Allocator )
-> concurrent_unordered_map<iterator_key_t<InputIterator>,
                            iterator_mapped_t<InputIterator>,
                            std::hash<iterator_key_t<InputIterator>>,
                            std::equal_to<iterator_key_t<InputIterator>>,
                            Allocator>;

template <typename Key,
          typename T,
          typename Hash = std::hash<std::remove_const_t<Key>>,
          typename KeyEqual = std::equal_to<std::remove_const_t<Key>>,
          typename Allocator = tbb::tbb_allocator<std::pair<const Key, T>>>
concurrent_unordered_map( std::initializer_list<std::pair<Key, T>>,
```

(continues on next page)

(continued from previous page)

```
                          map_size_type = {},
                          Hash = Hash(),
                          KeyEqual = KeyEqual(),
                          Allocator = Allocator() )
-> concurrent_unordered_map<std::remove_const_t<Key>,
                            T,
                            Hash,
                            KeyEqual,
                            Allocator>;

template <typename Key,
          typename T,
          typename Allocator>
concurrent_unordered_map( std::initializer_list<std::pair<Key, T>>,
                          map_size_type,
                          Allocator )
-> concurrent_unordered_map<std::remove_const_t<Key>,
                            T,
                            std::hash<std::remove_const_t<Key>>,
                            std::equal_to<std::remove_const_t<Key>>,
                            Allocator>;

template <typename Key,
          typename T,
          typename Allocator>
concurrent_unordered_map( std::initializer_list<std::pair<Key, T>>,
                          Allocator )
-> concurrent_unordered_map<std::remove_const_t<Key>,
                            T,
                            std::hash<std::remove_const_t<Key>>,
                            std::equal_to<std::remove_const_t<Key>>,
                            Allocator>;

template <typename Key,
          typename T,
          typename Hash,
          typename Allocator>
concurrent_unordered_map( std::initializer_list<std::pair<Key, T>>,
                          map_size_type,
                          Hash,
                          Allocator )
-> concurrent_unordered_map<std::remove_const_t<Key>,
                            T,
                            Hash,
                            std::equal_to<std::remove_const_t<Key>>,
                            Allocator>;
```

Where the `map_size_type` type refers to the `size_type` member type of the deduced `concurrent_unordered_map` and the type aliases `iterator_key_t`, `iterator_mapped_t`, and `iterator_alloc_value_t` are defined as follows:

```
template <typename InputIterator>
using iterator_key_t = std::remove_const_t<typename std::iterator_traits<InputIterator>
```

(continues on next page)

(continued from previous page)

```
↪::value_type::first_type>;

template <typename InputIterator>
using iterator_mapped_t = typename std::iterator_traits<InputIterator>::value_
↪type::second_type;

template <typename InputIterator>
using iterator_alloc_value_t = std::pair<std::add_const_t<iterator_key_t<InputIterator>,
                                         iterator_mapped_t<InputIterator>>>;
```

These deduction guides only participate in the overload resolution if the following requirements are met:

- The `InputIterator` type meets the `InputIterator` requirements described in the [input.iterators] section of the ISO C++ Standard.
- The `Allocator` type meets the `Allocator` requirements described in the [allocator.requirements] section of the ISO C++ Standard.
- The `Hash` type does not meet the `Allocator` requirements.
- The `KeyEqual` type does not meet the `Allocator` requirements.

Example

```
#include <oneapi/tbb/concurrent_unordered_map.h>
#include <vector>
#include <functional>

struct CustomHasher {...};

int main() {
    std::vector<std::pair<int, float>> v;

    // Deduces m1 as concurrent_unordered_map<int, float>
    oneapi::tbb::concurrent_unordered_map m1(v.begin(), v.end());

    // Deduces m2 as concurrent_unordered_map<int, float, CustomHasher>;
    oneapi::tbb::concurrent_unordered_map m2(v.begin(), v.end(), CustomHasher{});
}
```

concurrent_unordered_multimap

[containers.concurrent_unordered_multimap]

`oneapi::tbb::concurrent_unordered_multimap` is a class template that represents an unordered associative container. It stores key-value pairs and supports concurrent insertion, lookup, and traversal, but does not support concurrent erasure. In this container, multiple elements with equal keys can be stored.

Class Template Synopsis

```
// Defined in header <oneapi/tbb/concurrent_unordered_map.h>

namespace oneapi {
    namespace tbb {
        template <typename Key,
                  typename T,
                  typename Hash = std::hash<Key>,
                  typename KeyEqual = std::equal_to<Key>,
                  typename Allocator = tbb_allocator<std::pair<const Key, T>>>
        class concurrent_unordered_multimap {
        public:
            using key_type = Key;
            using mapped_type = T;
            using value_type = std::pair<const Key, T>;

            using size_type = <implementation-defined unsigned integer type>;
            using difference_type = <implementation-defined signed integer type>;

            using hasher = Hash;
            using key_equal = /*See below*/;

            using allocator_type = Allocator;

            using reference = value_type&;
            using const_reference = const value_type&;

            using pointer = typename std::allocator_traits<Allocator>::pointer;
            using const_pointer = typename std::allocator_traits<Allocator>::const_
↪pointer;

            using iterator = <implementation-defined ForwardIterator>;
            using const_iterator = <implementation-defined constant ForwardIterator>;

            using local_iterator = <implementation-defined ForwardIterator>;
            using const_local_iterator = <implementation-defined constant␣
↪ForwardIterator>;

            using node_type = <implementation-defined node handle>;

            using range_type = <implementation-defined ContainerRange>;
            using const_range_type = <implementation-defined constant ContainerRange>;

            // Construction, destruction, copying
            concurrent_unordered_multimap();

            explicit concurrent_unordered_multimap( size_type bucket_count, const hasher&
↪ hash = hasher(),
                                                    const key_equal& equal = key_equal(),
                                                    const allocator_type& alloc =␣
↪allocator_type() );
```

(continues on next page)

(continued from previous page)

```
            concurrent_unordered_multimap( size_type bucket_count, const allocator_type&␣
↪alloc );

            concurrent_unordered_multimap( size_type bucket_count, const hasher& hash,
                                           const allocator_type& alloc );

            explicit concurrent_unordered_multimap( const allocator_type& alloc );

            template <typename InputIterator>
            concurrent_unordered_multimap( InputIterator first, InputIterator last,
                                           size_type bucket_count = /*implementation-
↪defined*/,
                                           const hasher& hash = hasher(),
                                           const key_equal& equal = key_equal(),
                                           const allocator_type& alloc = allocator_
↪type() );

            template <typename Inputiterator>
            concurrent_unordered_multimap( InputIterator first, InputIterator last,
                                           size_type bucket_count, const allocator_type&␣
↪alloc );

            template <typename InputIterator>
            concurrent_unordered_multimap( InputIterator first, InputIterator last,
                                           size_type bucket_count, const hasher& hash,
                                           const allocator_type& alloc );

            concurrent_unordered_multimap( std::initializer_list<value_type> init,
                                           size_type bucket_count = /*implementation-
↪defined*/,
                                           const hasher& hash = hasher(),
                                           const key_equal& equal = key_equal(),
                                           const allocator_type& alloc = allocator_
↪type() );

            concurrent_unordered_multimap( std::initializer_list<value_type> init,
                                           size_type bucket_count, const allocator_type&␣
↪alloc );

            concurrent_unordered_multimap( std::initializer_list<value_type> init,
                                           size_type bucket_count, const hasher& hash,
                                           const allocator_type& alloc );

            concurrent_unordered_multimap( const concurrent_unordered_multimap& other );
            concurrent_unordered_multimap( const concurrent_unordered_multimap& other,
                                           const allocator_type& alloc );

            concurrent_unordered_multimap( concurrent_unordered_multimap&& other );
            concurrent_unordered_multimap( concurrent_unordered_multimap&& other,
                                           const allocator_type& alloc );

            ~concurrent_unordered_multimap();
```

(continues on next page)

(continued from previous page)

```
            concurrent_unordered_multimap& operator=( const concurrent_unordered_
↪multimap& other );
            concurrent_unordered_multimap& operator=( concurrent_unordered_multimap&&␣
↪other ) noexcept(/*See details*/);

            concurrent_unordered_multimap& operator=( std::initializer_list<value_type>␣
↪init );

            allocator_type get_allocator() const;

            // Iterators
            iterator begin() noexcept;
            const_iterator begin() const noexcept;
            const_iterator cbegin() const noexcept;

            iterator end() noexcept;
            const_iterator end() const noexcept;
            const_iterator cend() const noexcept;

            // Size and capacity
            bool empty() const noexcept;
            size_type size() const noexcept;
            size_type max_size() const noexcept;

            // Concurrently safe modifiers
            std::pair<iterator, bool> insert( const value_type& value );
            iterator insert( const_iterator hint, const value_type& value );

            template <typename P>
            std::pair<iterator, bool> insert( P&& value );

            template <typename P>
            iterator insert( const_iterator hint, P&& value );

            std::pair<iterator, bool> insert( value_type&& value );
            iterator insert( const_iterator hint, value_type&& value );

            template <typename InputIterator>
            void insert( InputIterator first, InputIterator last );

            void insert( std::initializer_list<value_type> init );

            std::pair<iterator, bool> insert( node_type&& nh );
            iterator insert( const_iterator hint, node_type&& nh );

            template <typename... Args>
            std::pair<iterator, bool> emplace( Args&&... args );

            template <typename... Args>
            iterator emplace_hint( const_iterator hint, Args&&... args );
```

(continues on next page)

(continued from previous page)

```
            template <typename SrcHash, typename SrcKeyEqual>
            void merge( concurrent_unordered_map<Key, T, SrcHash, SrcKeyEqual, Allocator>
↪& source );

            template <typename SrcHash, typename SrcKeyEqual>
            void merge( concurrent_unordered_map<Key, T, SrcHash, SrcKeyEqual, Allocator>
↪&& source );

            template <typename SrcHash, typename SrcKeyEqual>
            void merge( concurrent_unordered_multimap<Key, T, SrcHash, SrcKeyEqual,␣
↪Allocator>& source );

            template <typename SrcHash, typename SrcKeyEqual>
            void merge( concurrent_unordered_multimap<Key, T, SrcHash, SrcKeyEqual,␣
↪Allocator>&& source );

            // Concurrently unsafe modifiers
            void clear() noexcept;

            iterator unsafe_erase( const_iterator pos );
            iterator unsafe_erase( iterator pos );

            iterator unsafe_erase( const_iterator first, const_iterator last );

            size_type unsafe_erase( const key_type& key );

            template <typename K>
            size_type unsafe_erase( const K& key );

            node_type unsafe_extract( const_iterator pos );
            node_type unsafe_extract( iterator pos );

            node_type unsafe_extract( const key_type& key );

            template <typename K>
            node_type unsafe_extract( const K& key );

            void swap( concurrent_unordered_multimap& other );

            // Lookup
            size_type count( const key_type& key ) const;

            template <typename K>
            size_type count( const K& key ) const;

            iterator find( const key_type& key );
            const_iterator find( const key_type& key ) const;

            template <typename K>
            iterator find( const K& key );

            template <typename K>
```

(continues on next page)

(continued from previous page)

```
            const_iterator find( const K& key ) const;

            bool contains( const key_type& key ) const;

            template <typename K>
            bool contains( const K& key ) const;

            std::pair<iterator, iterator> equal_range( const key_type& key );
            std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;

            template <typename K>
            std::pair<iterator, iterator> equal_range( const K& key );

            template <typename K>
            std::pair<const_iterator, const_iterator> equal_range( const K& key ) const;

            // Bucket interface
            local_iterator unsafe_begin( size_type n );
            const_local_iterator unsafe_begin( size_type n ) const;
            const_local_iterator unsafe_cbegin( size_type n ) const;

            local_iterator unsafe_end( size_type n );
            const_local_iterator unsafe_end( size_type n ) const;
            const_local_iterator unsafe_cend( size_type n ) const;

            size_type unsafe_bucket_count() const;
            size_type unsafe_max_bucket_bount() const;

            size_type unsafe_bucket_size( size_type n ) const;

            size_type unsafe_bucket( const key_type& key ) const;

            // Hash policy
            float load_factor() const;

            float max_load_factor() const;
            void max_load_factor( float ml );

            void rehash( size_type count );

            void reserve( size_type count );

            // Observers
            hasher hash_function() const;
            key_equal key_eq() const;

            // Parallel iteration
            range_type range();
            const_range_type range() const;
        }; // class concurrent_unordered_multimap
    } // namespace tbb
```

(continues on next page)

(continued from previous page)

```
} // namespace oneapi
```

Requirements:

- The expression `std::allocator_type<Allocator>::destroy(m, val)`, where `m` is an object of the type `Allocator` and `val` is an object of type `value_type`, must be well-formed. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Hash` must meet the `Hash` requirements from the [hash] ISO C++ Standard section.
- The type `KeyEqual` must meet the `BinaryPredicate` requirements from the [algorithms.general] ISO C++ Standard section.
- The type `Allocator` must meet the `Allocator` requirements from the [allocator.requirements] ISO C++ Standard section.

Description

`oneapi::tbb::concurrent_unordered_multimap` is an unordered associative container, which elements are organized into buckets. The value of the hash function `Hash` for a `Key` object determines the number of the bucket in which the corresponding element will be placed.

If the qualified-id `Hash::transparent_key_equal` is valid and denotes a type, the member type `concurrent_unordered_multimap::key_equal` is defined as the value of this qualified-id. In this case, the program is ill-formed if any of the following conditions are met:

- The template parameter `KeyEqual` is different from `std::equal_to<Key>`.
- Qualified-id `Hash::transparent_key_equal::is_transparent` is not valid or does not denote a type.

Otherwise, the type `concurrent_unordered_multimap::key_equal` is defined as the value of the template parameter `KeyEqual`.

Member functions

Construction, destruction, copying

Empty container constructors

```
concurrent_unordered_multimap();

explicit concurrent_unordered_multimap( const allocator_type& alloc );
```

Constructs an empty `concurrent_unordered_multimap`. The initial number of buckets is unspecified.

If provided uses the allocator `alloc` to allocate the memory.

```
explicit concurrent_unordered_multimap( size_type bucket_count,
                                        const hasher& hash = hasher(),
                                        const key_equal& equal = key_equal(),
                                        const allocator_type& alloc =␣
↪allocator_type() );
```

(continues on next page)

(continued from previous page)

```
concurrent_unordered_multimap( size_type bucket_count, const allocator_type&␣
↪alloc );

concurrent_unordered_multimap( size_type bucket_count, const hasher& hash,
                               const allocator_type& alloc );
```

Constructs an empty `concurrent_unordered_multimap` with `bucket_count` buckets.

If provided uses the hash function `hasher`, predicate `equal` to compare `key_type` objects for equality, and the allocator `alloc` to allocate the memory.

Constructors from the sequence of elements

```
template <typename InputIterator>
concurrent_unordered_multimap( InputIterator first, InputIterator last,
                               size_type bucket_count = /*implementation-
↪defined*/,
                               const hasher& hash = hasher(),
                               const key_equal& equal = key_equal(),
                               const allocator_type& alloc = allocator_type()␣
↪);

template <typename Inputiterator>
concurrent_unordered_multimap( InputIterator first, InputIterator last,
                               size_type bucket_count, const allocator_type&␣
↪alloc );

template <typename InputIterator>
concurrent_unordered_multimap( InputIterator first, InputIterator last,
                               size_type bucket_count, const hasher& hash,
                               const allocator_type& alloc );
```

Constructs the `concurrent_unordered_multimap` that contains all elements from the half-open interval `[first, last)`.

If provided, uses the hash function `hasher`, predicate `equal` to compare `key_type` objects for equality and the allocator `alloc` to allocate the memory.

Requirements: the type `InputIterator` must meet the requirements of `InputIterator` from the [input.iterators] ISO C++ Standard section.

```
concurrent_unordered_multimap( std::initializer_list<value_type> init,
                               size_type bucket_count = /*implementation-
↪defined*/,
                               const hasher& hash = hasher(),
                               const key_equal& equal = key_equal(),
                               const allocator_type& alloc = allocator_type()␣
↪);
```

Equivalent to `concurrent_unordered_multimap(init.begin(), init.end(), bucket_count, hash, equal, alloc)`.

```
concurrent_unordered_multimap( std::initializer_list<value_type> init,
                               size_type bucket_count, const allocator_type&␣
↪alloc );
```

Equivalent to `concurrent_unordered_multimap(init.begin(), init.end(), bucket_count, alloc)`.

```
concurrent_unordered_multimap( std::initializer_list<value_type> init,
                               size_type bucket_count, const hasher& hash,
                               const allocator_type& alloc );
```

Equivalent to `concurrent_unordered_multimap(init.begin(), init.end(), bucket_count, hash, alloc)`.

Copying constructors

```
concurrent_unordered_multimap( const concurrent_unordered_multimap& other );

concurrent_unordered_multimap( const concurrent_unordered_multimap& other,
                               const allocator_type& alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Moving constructors

```
concurrent_unordered_multimap( concurrent_unordered_multimap&& other );

concurrent_unordered_multimap( concurrent_unordered_multimap&& other,
                               const allocator_type& alloc );
```

Constructs a `concurrent_unordered_multimap` with the contents of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by calling `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Destructor

```
~concurrent_unordered_multimap();
```

Destroys the `concurrent_unordered_multimap`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

Assignment operators

```
concurrent_unordered_multimap& operator=( const concurrent_unordered_multimap&␣
↪other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_assignment::` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_unordered_multimap& operator=( concurrent_unordered_multimap&&␣
↪other ) noexcept(/*See below*/);
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment:` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

Exceptions: `noexcept` specification:

```
noexcept(std::allocator_traits<allocator_type>::is_always_equal::value␣
↪&&
         std::is_nothrow_move_assignable<hasher>::value &&
         std::is_nothrow_move_assignable<key_equal>::value)
```

```
concurrent_unordered_multimap& operator=( std::initializer_list<value_type>␣
↪init );
```

Replaces all elements in `*this` by the elements in `init`.

If `init` contains multiple elements with equal keys, it is unspecified which element would be inserted.

The behavior is undefined in case of concurrent operations with `*this`.

Returns: a reference to `*this`.

Iterators

The types `concurrent_unordered_multimap::iterator` and `concurrent_unordered_multimap::const_iterator` meet the requirements of `ForwardIterator` from the [forward.iterators] ISO C++ Standard section.

begin and cbegin

```
iterator begin();

const_iterator begin() const;

const_iterator cbegin() const;
```

Returns: an iterator to the first element in the container.

end and cend

```
iterator end();

const_iterator end() const;

const_iterator cend() const;
```

Returns: an iterator to the element that follows the last element in the container.

Size and capacity

empty

```
bool empty() const;
```

Returns: `true` if the container is empty; `false`, otherwise.

The result may differ from the actual container state in case of pending concurrent insertions.

size

```
size_type size() const;
```

Returns: the number of elements in the container.

The result may differ from the actual container size in case of pending concurrent insertions.

max_size

```
size_type max_size() const;
```

Returns: the maximum number of elements that container can hold.

Concurrently safe modifiers

All member functions in this section can be performed concurrently with each other, lookup methods and while traversing the container.

Emplacing elements

```
template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args )
```

Inserts an element constructed in-place from `args` into the container.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element. Boolean value is always `true`.

```
template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args )
```

Inserts an element constructed in-place from `args` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

Returns: an `iterator` to the inserted element.

Inserting values

```
std::pair<iterator, bool> insert( const value_type& value )
```

Inserts the value `value` into the container.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element. Boolean value is always `true`.

```
iterator insert( const_iterator hint, const value_type& other )
```

Inserts the value `value` into the container.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

Returns: an `iterator` to the inserted element.

```
template <typename P>
std::pair<iterator, bool> insert( P&& value )
```

Equivalent to `emplace(std::forward<P>(value))`.

This overload only participates in overload resolution if `std::is_constructible<value_type, P&&>::value` is `true`.

```
template <typename P>
iterator insert( const_iterator hint, P&& value )
```

Equivalent to `emplace_hint(hint, std::forward<P>(value))`.

This overload only participates in overload resolution if `std::is_constructible<value_type, P&&>::value` is `true`.

```
std::pair<iterator, bool> insert( value_type&& value )
```

Inserts the value `value` into the container using move semantics.

`value` is left in a valid, but unspecified state.

Returns: `std::pair<iterator, bool>` where `iterator` points to the inserted element. Boolean value is always `true`.

```
iterator insert( const_iterator hint, value_type&& other )
```

Inserts the value `value` into the container using move semantics.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

`value` is left in a valid, but unspecified state.

Returns: an `iterator` to the inserted element.

Inserting sequences of elements

```
template <typename InputIterator>
void insert( InputIterator first, InputIterator last )
```

Inserts all items from the half-open interval `[first, last)` into the container.

Requirements: the type `InputIterator` must meet the requirements of *InputIterator* from the `[input.iterators]` ISO C++ Standard section.

```
void insert( std::initializer_list<value_type> init )
```

Equivalent to `insert(init.begin(), init.end())`.

Inserting nodes

```
std::pair<iterator, bool> insert( node_type&& nh )
```

If the node handle `nh` is empty, does nothing.

Otherwise, inserts the node owned by `nh` into the container.

`nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element. Boolean value is always `true`.

```
iterator insert( const_iterator hint, node_type&& nh )
```

If the node handle `nh` is empty, does nothing.

Otherwise, inserts the node owned by `nh` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

`nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: an iterator pointing to the inserted element.

Merging containers

```
template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_map<Key, T, SrcHash, SrcKeyEqual, Allocator>&␣
↪source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_map<Key, T, SrcHash, SrcKeyEqual, Allocator>&&
↪ source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_multimap<Key, T, SrcHash, SrcKeyEqual,␣
↪Allocator>& source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_multimap<Key, T, SrcHash, SrcKeyEqual,␣
↪Allocator>&& source );
```

Transfers all elements from `source` to `*this`.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `get_allocator() != source.get_allocator()`.

Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

Clearing

```
void clear();
```

Removes all elements from the container.

Erasing elements

```
iterator unsafe_erase( const_iterator pos );

iterator unsafe_erase( iterator pos );
```

Removes the element pointed to by `pos` from the container.

Invalidates all iterators and references to the removed element.

Returns: `iterator` that follows the removed element.

Requirements: the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
size_type unsafe_erase( const key_type& key );
```

Removes all elements with the key equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed elements.

Returns: the number of removed elements.

```
template <typename K>
size_type unsafe_erase( const K& key );
```

Removes all elements with the key equivalent to `key` if they exist in the container.

Invalidates all iterators and references to the removed elements.

This overload only participates in overload resolution if all of the following statements are `true`:

- The qualified-id `hasher::transparent_key_equal` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: the number of removed elements.

Erasing sequences

```
iterator unsafe_erase( const_iterator first, const_iterator last );
```

Removes all elements from the half-open interval `[first, last)` from the container.

Returns: `iterator` that follows the last removed element.

Requirements: the range `[first, last)` must be a valid subrange in `*this`.

Extracting nodes

```
node_type unsafe_extract( iterator pos );

node_type unsafe_extract( const_iterator pos );
```

Transfers ownership of the element pointed to by `pos` from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element.

Requirements: the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
node_type unsafe_extract( const key_type& key );
```

If at least one element with the key equivalent to `key` exists, transfers ownership of one of these element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

If there are multiple elements with the key equivalent to `key`, it is unspecified which element should be transferred.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element or an empty node handle if an element with the key equivalent to `key` was not found.

```
template <typename K>
node_type unsafe_extract( const K& key );
```

If at least one element with the key equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

If there are multiple elements with the key equivalent to `key` exists, it is unspecified which element should be transferred.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

This overload only participates in overload resolution if all of the following statements are `true`:

- The qualified-id `hasher::transparent_key_equal` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: the node handle that owns the extracted element or an empty node handle if an element with the key equivalent to `key` was not found.

swap

```
void swap( concurrent_unordered_multimap& other ) noexcept(/*See below*/);
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`.

Otherwise, if `get_allocator() != other.get_allocator()`, the behavior is undefined.

Exceptions: `noexcept` specification:

```
noexcept(std::allocator_traits<allocator_type>::is_always_equal::value␣
↪&&
         std::is_nothrow_swappable<hasher>::value &&
         std::is_nothrow_swappable<key_equal>::value
```

Lookup

All methods in this section can be executed concurrently with each other, concurrently-safe modifiers and while traversing the container.

count

```
size_type count( const key_type& key );
```

Returns: the number of elements with the key equivalent to `key`.

```
template <typename K>
size_type count( const K& key );
```

Returns: the number of elements with the key equivalent to `key`.

This overload only participates in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

find

```
iterator find( const key_type& key );

const_iterator find( const key_type& key ) const;
```

Returns: an iterator to the element with the key equivalent to `key`, or `end()` if no such element exists.

If there are multiple elements with the key equivalent to `key`, it is unspecified which element should be found.

```
template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;
```

Returns: an iterator to the element with the key equivalent to `key`, or `end()` if no such element exists.

If there are multiple elements with the key equivalent to `key`, it is unspecified which element should be found.

These overloads only participates in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

contains

```
bool contains( const key_type& key ) const;
```

Returns: `true` if at least one element with the key equivalent to `key` exists in the container; `false`, otherwise.

```
template <typename K>
bool contains( const K& key ) const;
```

Returns: `true` if at least one element with the key equivalent to `key` exists in the container; `false`, otherwise.

This overload only participates in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

equal_range

```
std::pair<iterator, iterator> equal_range( const key_type& key );

std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;
```

Returns: if at least one element with the key equivalent to `key` exists, a pair of iterators `{f, l}`, where `f` is an iterator to the first element with the key equivalent to `key`, `l` is an iterator to the element which follows the last element with the key equivalent to `key`. Otherwise, `{end(), end()}`.

```
template <typename K>
std::pair<iterator, iterator> equal_range( const K& key )

template <typename K>
std::pair<const_iterator, const_iterator> equal_range( const K& key )
```

Returns: if at least one element with the key equivalent to `key` exists - a pair of iterators `{f, l}`, where `f` is an iterator to the first element with the key equivalent to `key`, `l` is an iterator to the element that follows the last element with the key equivalent to `key`. Otherwise,``{end(), end()}``.

These overloads only participates in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

Bucket interface

The types `concurrent_unordered_multimap::local_iterator` and `concurrent_unordered_multimap::const_local_iter` meet the requirements of `ForwardIterator` from the [forward.iterators] ISO C++ Standard section.

These iterators are used to traverse the certain bucket.

All methods in this section can only be executed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

Bucket begin and bucket end

```
local_iterator unsafe_begin( size_type n );

const_local_iterator unsafe_begin( size_type n ) const;

const_local_iterator unsafe_cbegin( size_type n ) const;
```

Returns: an iterator to the first element in the bucket number `n`.

```
local_iterator unsafe_end( size_type n );

const_local_iterator unsafe_end( size_type n ) const;

const_local_iterator unsafe_cend( size_type n ) const;
```

Returns: an iterator to the element that follows the last element in the bucket number `n`.

The number of buckets

```
size_type unsafe_bucket_count() const;
```

Returns: the number of buckets in the container.

```
size_type unsafe_max_bucket_count() const;
```

Returns: the maximum number of buckets that container can hold.

Size of the bucket

```
size_type unsafe_bucket_size( size_type n ) const;
```

Returns: the number of elements in the bucket number `n`.

Bucket number

```
size_type unsafe_bucket( const key_type& key ) const;
```

Returns: the number of the bucket in which the element with the key `key` is stored.

Hash policy

Hash policy of `concurrent_unordered_multimap` manages the number of buckets in the container and the allowed maximum number of elements per bucket (load factor). If the maximum load factor is exceeded, the container can automatically increase the number of buckets.

Load factor

```
float load_factor() const;
```

Returns: the average number of elements per bucket, which is `size()/unsafe_bucket_count()`.

```
float max_load_factor() const;
```

Returns: the maximum number of elements per bucket.

```
void max_load_factor( float ml );
```

Sets the maximum number of elements per bucket to `ml`.

Manual rehashing

```
void rehash( size_type n );
```

Sets the number of buckets to `n` and rehashes the container.

```
void reserve( size_type n );
```

Sets the number of buckets to the value that is needed to store `n` elements.

Observers

get_allocator

```
allocator_type get_allocator() const;
```

Returns: a copy of the allocator associated with `*this`.

hash_function

```
hasher hash_function() const;
```

Returns: a copy of the hash function associated with `*this`.

key_eq

```
key_equal key_eq() const;
```

Returns: a copy of the key equality predicate associated with `*this`.

Parallel iteration

Member types `concurrent_unordered_multimap::range_type` and `concurrent_unordered_multimap::const_range_type` meet the *ContainerRange requirements*.

These types differ only in that the bounds for a `concurrent_unordered_multimap::const_range_type` are of type `concurrent_unordered_multimap::const_iterator`, whereas the bounds for a `concurrent_unordered_multimap::range_type` are of type `concurrent_unordered_multimap::iterator`.

range member function

```
range_type range();

const_range_type range() const;
```

Returns: a range object representing all elements in the container.

Non-member functions

These functions provides binary comparison and swap operations on `oneapi::tbb::concurrent_unordered_multimap` objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define `oneapi::tbb::concurrent_unordered_multimap` as a type alias for which the non-member functions are reachable only via argument-dependent lookup.

```
template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
void swap( concurrent_unordered_multimap<Key, T, Hash, KeyEqual, Allocator>& lhs,
           concurrent_unordered_multimap<Key, T, Hash, KeyEqual, Allocator>& rhs );

template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_multimap<Key, T, Hash, KeyEqual, Allocator>&␣
↪lhs,
                 const concurrent_unordered_multimap<Key, T, Hash, KeyEqual, Allocator>&␣
↪rhs );

template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_multimap<Key, T, Hash, KeyEqual, Allocator>&␣
↪lhs,
                 const concurrent_unordered_multimap<Key, T, Hash, KeyEqual, Allocator>&␣
↪rhs );
```

Non-member swap

```
template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
void swap( concurrent_unordered_multimap<Key, T, Hash, KeyEqual, Allocator>& lhs,
           concurrent_unordered_multimap<Key, T, Hash, KeyEqual, Allocator>& rhs )␣
↪noexcept(noexcept(lhs.swap(rhs)));
```

Equivalent to `lhs.swap(rhs)`.

Non-member binary comparisons

Two objects of `concurrent_unordered_multimap` are equal if the following conditions are `true`:

- They contain an equal number of elements.
- Each group of elements with the same key in one container has the corresponding group of equivalent elements in the other container (not necessary in the same order).

```
template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_multimap<Key, T, Hash, KeyEqual, Allocator>&␣
↪lhs,
                 const concurrent_unordered_multimap<Key, T, Hash, KeyEqual, Allocator>&␣
↪rhs );
```

Returns: `true` if `lhs` is equal to `rhs`; `false`, otherwise.

```
template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator!=( const concurrent_unordered_multimap<Key, T, Hash, KeyEqual, Allocator>&␣
↪lhs,
                 const concurrent_unordered_multimap<Key, T, Hash, KeyEqual, Allocator>&␣
↪rhs );
```

Equivalent to `!(lhs == rhs)`.

Returns: `true` if `lhs` is not equal to `rhs`; `false`, otherwise.

Other

Deduction guides

If possible, `concurrent_unordered_multimap` constructors support class template argument deduction (since C++17). Copy and move constructors, including constructors with an explicit `allocator_type` argument, provide implicitly-generated deduction guides. In addition, the following explicit deduction guides are provided:

```
template <typename InputIterator,
          typename Hash = std::hash<iterator_key_t<InputIterator>>,
          typename KeyEqual = std::equal_to<iterator_key_t<InputIterator>>,
          typename Allocator = tbb::tbb_allocator<iterator_alloc_value_t<InputIterator>>>
concurrent_unordered_multimap( InputIterator, InputIterator,
                               map_size_type = {},
                               Hash = Hash(),
                               KeyEqual = KeyEqual(),
                               Allocator = Allocator() )
-> concurrent_unordered_multimap<iterator_key_t<InputIterator>,
                                 iterator_mapped_t<InputIterator>,
                                 Hash,
                                 KeyEqual,
                                 Allocator>;
```

(continues on next page)

(continued from previous page)

```
template <typename InputIterator,
          typename Hash,
          typename Allocator>
concurrent_unordered_multimap( InputIterator, InputIterator,
                               map_size_type,
                               Hash,
                               Allocator )
-> concurrent_unordered_multimap<iterator_key_t<InputIterator>,
                                 iterator_mapped_t<InputIterator>,
                                 Hash,
                                 std::equal_to<iterator_key_t<InputIterator>>,
                                 Allocator>;

template <typename InputIterator,
          typename Allocator>
concurrent_unordered_multimap( InputIterator, InputIterator,
                               map_size_type,
                               Allocator )
-> concurrent_unordered_multimap<iterator_key_t<InputIterator>,
                                 iterator_mapped_t<InputIterator>,
                                 std::hash<iterator_key_t<InputIterator>>,
                                 std::equal_to<iterator_key_t<InputIterator>>,
                                 Allocator>;

template <typename Key,
          typename T,
          typename Hash = std::hash<std::remove_const_t<Key>>,
          typename KeyEqual = std::equal_to<std::remove_const_t<Key>>,
          typename Allocator = tbb::tbb_allocator<std::pair<const Key, T>>>
concurrent_unordered_multimap( std::initializer_list<std::pair<Key, T>>,
                               map_size_type = {},
                               Hash = Hash(),
                               KeyEqual = KeyEqual(),
                               Allocator = Allocator() )
-> concurrent_unordered_multimap<std::remove_const_t<Key>,
                                 T,
                                 Hash,
                                 KeyEqual,
                                 Allocator>;

template <typename Key,
          typename T,
          typename Allocator>
concurrent_unordered_multimap( std::initializer_list<std::pair<Key, T>>,
                               map_size_type,
                               Allocator )
-> concurrent_unordered_multimap<std::remove_const_t<Key>,
                                 T,
                                 std::hash<std::remove_const_t<Key>>,
                                 std::equal_to<std::remove_const_t<Key>>,
                                 Allocator>;
```

(continues on next page)

(continued from previous page)

```
template <typename Key,
          typename T,
          typename Allocator>
concurrent_unordered_multimap( std::initializer_list<std::pair<Key, T>>,
                               Allocator )
-> concurrent_unordered_multimap<std::remove_const_t<Key>,
                                 T,
                                 std::hash<std::remove_const_t<Key>>,
                                 std::equal_to<std::remove_const_t<Key>>,
                                 Allocator>;

template <typename Key,
          typename T,
          typename Hash,
          typename Allocator>
concurrent_unordered_multimap( std::initializer_list<std::pair<Key, T>>,
                               map_size_type,
                               Hash,
                               Allocator )
-> concurrent_unordered_multimap<std::remove_const_t<Key>,
                                 T,
                                 Hash,
                                 std::equal_to<std::remove_const_t<Key>>,
                                 Allocator>;
```

Where the `map_size_type` type refers to the `size_type` member type of the deduced `concurrent_unordered_multimap` and the type aliases `iterator_key_t`, `iterator_mapped_t`, and `iterator_alloc_value_t` are defined as follows:

```
template <typename InputIterator>
using iterator_key_t = std::remove_const_t<typename std::iterator_traits<InputIterator>
↪::value_type::first_type>;

template <typename InputIterator>
using iterator_mapped_t = typename std::iterator_traits<InputIterator>::value_
↪type::second_type;

template <typename InputIterator>
using iterator_alloc_value_t = std::pair<std::add_const_t<iterator_key_t<InputIterator>,
                                         iterator_mapped_t<InputIterator>>>;
```

These deduction guides only participate in the overload resolution if the following requirements are met:

- The `InputIterator` type meets the `InputIterator` requirements described in the [input.iterators] section of the ISO C++ Standard.
- The `Allocator` type meets the `Allocator` requirements described in the [allocator.requirements] section of the ISO C++ Standard.
- The `Hash` type does not meet the `Allocator` requirements.
- The `KeyEqual` type does not meet the `Allocator` requirements.

Example

```
#include <oneapi/tbb/concurrent_unordered_map.h>
#include <vector>
#include <functional>

struct CustomHasher {...};

int main() {
    std::vector<std::pair<int, float>> v;

    // Deduces m1 as concurrent_unordered_multimap<int, float>
    oneapi::tbb::concurrent_unordered_multimap m1(v.begin(), v.end());

    // Deduces m2 as concurrent_unordered_multimap<int, float, CustomHasher>;
    oneapi::tbb::concurrent_unordered_multimap m2(v.begin(), v.end(), CustomHasher{});
}
```

concurrent_unordered_set

[containers.concurrent_unordered_set]

`oneapi::tbb::concurrent_unordered_set` is a class template that represents an unordered sequence of unique elements. It supports concurrent insertion, lookup, and traversal, but does not support concurrent erasure.

Class Template Synopsis

```
// Defined in header <oneapi/tbb/concurrent_unordered_set.h>

namespace oneapi {
    namespace tbb {
        template <typename T,
                  typename Hash = std::hash<Key>,
                  typename KeyEqual = std::equal_to<Key>,
                  typename Allocator = tbb_allocator<std::pair<const Key, T>>>
        class concurrent_unordered_set {
        public:
            using key_type = Key;
            using value_type = Key;

            using size_type = <implementation-defined unsigned integer type>;
            using difference_type = <implementation-defined signed integer type>;

            using hasher = Hash;
            using key_equal = /*See below*/;

            using allocator_type = Allocator;

            using reference = value_type&;
            using const_reference = const value_type&;

            using pointer = typename std::allocator_traits<Allocator>::pointer;
```

(continues on next page)

(continued from previous page)

```
            using const_pointer = typename std::allocator_traits<Allocator>::const_
↪pointer;

            using iterator = <implementation-defined ForwardIterator>;
            using const_iterator = <implementation-defined constant ForwardIterator>;

            using local_iterator = <implementation-defined ForwardIterator>;
            using const_local_iterator = <implementation-defined constant␣
↪ForwardIterator>;

            using node_type = <implementation-defined node handle>;

            using range_type = <implementation-defined ContainerRange>;
            using const_range_type = <implementation-defined constant ContainerRange>;

            // Construction, destruction, copying
            concurrent_unordered_set();

            explicit concurrent_unordered_set( size_type bucket_count, const hasher&␣
↪hash = hasher(),
                                               const key_equal& equal = key_equal(),
                                               const allocator_type& alloc = allocator_
↪type() );

            concurrent_unordered_set( size_type bucket_count, const allocator_type&␣
↪alloc );

            concurrent_unordered_set( size_type bucket_count, const hasher& hash,
                                      const allocator_type& alloc );

            explicit concurrent_unordered_set( const allocator_type& alloc );

            template <typename InputIterator>
            concurrent_unordered_set( InputIterator first, InputIterator last,
                                      size_type bucket_count = /*implementation-defined*/
↪,
                                      const hasher& hash = hasher(),
                                      const key_equal& equal = key_equal(),
                                      const allocator_type& alloc = allocator_type() );

            template <typename Inputiterator>
            concurrent_unordered_set( InputIterator first, InputIterator last,
                                      size_type bucket_count, const allocator_type&␣
↪alloc );

            template <typename InputIterator>
            concurrent_unordered_set( InputIterator first, InputIterator last,
                                      size_type bucket_count, const hasher& hash,
                                      const allocator_type& alloc );

            concurrent_unordered_set( std::initializer_list<value_type> init,
                                      size_type bucket_count = /*implementation-defined*/
```

(continues on next page)

(continued from previous page)

```
↪,
                                    const hasher& hash = hasher(),
                                    const key_equal& equal = key_equal(),
                                    const allocator_type& alloc = allocator_type() );

            concurrent_unordered_set( std::initializer_list<value_type> init,
                                    size_type bucket_count, const allocator_type&␣
↪alloc );

            concurrent_unordered_set( std::initializer_list<value_type> init,
                                    size_type bucket_count, const hasher& hash,
                                    const allocator_type& alloc );

            concurrent_unordered_set( const concurrent_unordered_set& other );
            concurrent_unordered_set( const concurrent_unordered_set& other,
                                    const allocator_type& alloc );

            concurrent_unordered_set( concurrent_unordered_set&& other );
            concurrent_unordered_set( concurrent_unordered_set&& other,
                                    const allocator_type& alloc );

            ~concurrent_unordered_set();

            concurrent_unordered_set& operator=( const concurrent_unordered_set& other );
            concurrent_unordered_set& operator=( concurrent_unordered_set&& other )␣
↪noexcept(/*See details*/);

            concurrent_unordered_set& operator=( std::initializer_list<value_type> init␣
↪);

            allocator_type get_allocator() const;

            // Iterators
            iterator begin() noexcept;
            const_iterator begin() const noexcept;
            const_iterator cbegin() const noexcept;

            iterator end() noexcept;
            const_iterator end() const noexcept;
            const_iterator cend() const noexcept;

            // Size and capacity
            bool empty() const noexcept;
            size_type size() const noexcept;
            size_type max_size() const noexcept;

            // Concurrently safe modifiers
            std::pair<iterator, bool> insert( const value_type& value );
            iterator insert( const_iterator hint, const value_type& value );

            std::pair<iterator, bool> insert( value_type&& value );
            iterator insert( const_iterator hint, value_type&& value );
```

(continues on next page)

(continued from previous page)

```
            template <typename InputIterator>
            void insert( InputIterator first, InputIterator last );

            void insert( std::initializer_list<value_type> init );

            std::pair<iterator, bool> insert( node_type&& nh );
            iterator insert( const_iterator hint, node_type&& nh );

            template <typename... Args>
            std::pair<iterator, bool> emplace( Args&&... args );

            template <typename... Args>
            iterator emplace_hint( const_iterator hint, Args&&... args );

            template <typename SrcHash, typename SrcKeyEqual>
            void merge( concurrent_unordered_set<T, SrcHash, SrcKeyEqual, Allocator>&␣
↪source );

            template <typename SrcHash, typename SrcKeyEqual>
            void merge( concurrent_unordered_set<T, SrcHash, SrcKeyEqual, Allocator>&&␣
↪source );

            template <typename SrcHash, typename SrcKeyEqual>
            void merge( concurrent_unordered_multiset<T, SrcHash, SrcKeyEqual, Allocator>
↪& source );

            template <typename SrcHash, typename SrcKeyEqual>
            void merge( concurrent_unordered_multiset<T, SrcHash, SrcKeyEqual, Allocator>
↪&& source );

            // Concurrently unsafe modifiers
            void clear() noexcept;

            iterator unsafe_erase( const_iterator pos );
            iterator unsafe_erase( iterator pos );

            iterator unsafe_erase( const_iterator first, const_iterator last );

            size_type unsafe_erase( const key_type& key );

            template <typename K>
            size_type unsafe_erase( const K& key );

            node_type unsafe_extract( const_iterator pos );
            node_type unsafe_extract( iterator pos );

            node_type unsafe_extract( const key_type& key );

            template <typename K>
            node_type unsafe_extract( const K& key );
```

(continues on next page)

(continued from previous page)

```
            void swap( concurrent_unordered_set& other );

            // Lookup
            size_type count( const key_type& key ) const;

            template <typename K>
            size_type count( const K& key ) const;

            iterator find( const key_type& key );
            const_iterator find( const key_type& key ) const;

            template <typename K>
            iterator find( const K& key );

            template <typename K>
            const_iterator find( const K& key ) const;

            bool contains( const key_type& key ) const;

            template <typename K>
            bool contains( const K& key ) const;

            std::pair<iterator, iterator> equal_range( const key_type& key );
            std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;

            template <typename K>
            std::pair<iterator, iterator> equal_range( const K& key );

            template <typename K>
            std::pair<const_iterator, const_iterator> equal_range( const K& key ) const;

            // Bucket interface
            local_iterator unsafe_begin( size_type n );
            const_local_iterator unsafe_begin( size_type n ) const;
            const_local_iterator unsafe_cbegin( size_type n ) const;

            local_iterator unsafe_end( size_type n );
            const_local_iterator unsafe_end( size_type n ) const;
            const_local_iterator unsafe_cend( size_type n ) const;

            size_type unsafe_bucket_count() const;
            size_type unsafe_max_bucket_bount() const;

            size_type unsafe_bucket_size( size_type n ) const;

            size_type unsafe_bucket( const key_type& key ) const;

            // Hash policy
            float load_factor() const;

            float max_load_factor() const;
```

(continues on next page)

(continued from previous page)

```
            void max_load_factor( float ml );

            void rehash( size_type count );

            void reserve( size_type count );

            // Observers
            hasher hash_function() const;
            key_equal key_eq() const;

            // Parallel iteration
            range_type range();
            const_range_type range() const;
        }; // class concurrent_unordered_set
    } // namespace tbb
} // namespace oneapi
```

Requirements:

- The expression `std::allocator_type<Allocator>::destroy(m, val)`, where `m` is an object of the type `Allocator` and `val` is an object of type `value_type`, must be well-formed. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Hash` must meet the `Hash` requirements from the [hash] ISO C++ Standard section.
- The type `KeyEqual` must meet the `BinaryPredicate` requirements from the [algorithms.general] ISO C++ Standard section.
- The type `Allocator` must meet the `Allocator` requirements from the [allocator.requirements] ISO C++ Standard section.

Description

`oneapi::tbb::concurrent_unordered_set` is an unordered sequence, which elements are organized into buckets. The value of the hash function `Hash` for `Key` object determines the number of the bucket in which the corresponding element will be placed.

If the qualified-id `Hash::transparent_key_equal` is valid and denotes a type, the member type `concurrent_unordered_set::key_equal` is defined as the value of this qualified-id. In this case, the program is ill-formed if any of the following conditions are met:

- The template parameter `KeyEqual` is different from `std::equal_to<Key>`.
- Qualified-id `Hash::transparent_key_equal::is_transparent` is not valid or does not denote a type.

Otherwise, the member type `concurrent_unordered_set::key_equal` is defined as the value of the template parameter `KeyEqual`.

Member functions

Construction, destruction, copying

Empty container constructors

```
concurrent_unordered_set();

explicit concurrent_unordered_set( const allocator_type& alloc );
```

Constructs an empty `concurrent_unordered_set`. The initial number of buckets is unspecified.

If provided, uses the allocator `alloc` to allocate the memory.

```
explicit concurrent_unordered_set( size_type bucket_count,
                                   const hasher& hash = hasher(),
                                   const key_equal& equal = key_equal(),
                                   const allocator_type& alloc = allocator_
↪type() );

concurrent_unordered_set( size_type bucket_count, const allocator_type& alloc␣
↪);

concurrent_unordered_set( size_type bucket_count, const hasher& hash,
                          const allocator_type& alloc );
```

Constructs an empty `concurrent_unordered_set` with `bucket_count` buckets.

If provided, uses the hash function `hasher`, predicate `equal` to compare `key_type` objects for equality, and the allocator `alloc` to allocate the memory.

Constructors from the sequence of elements

```
template <typename InputIterator>
concurrent_unordered_set( InputIterator first, InputIterator last,
                          size_type bucket_count = /*implementation-defined*/,
                          const hasher& hash = hasher(),
                          const key_equal& equal = key_equal(),
                          const allocator_type& alloc = allocator_type() );

template <typename Inputiterator>
concurrent_unordered_set( InputIterator first, InputIterator last,
                          size_type bucket_count, const allocator_type& alloc␣
↪);

template <typename InputIterator>
concurrent_unordered_set( InputIterator first, InputIterator last,
                          size_type bucket_count, const hasher& hash,
                          const allocator_type& alloc );
```

Constructs the `concurrent_unordered_set` that contains the elements from the half-open interval `[first, last)`.

If the range `[first, last)` contains multiple equal elements, it is unspecified which element would be inserted.

If provided, uses the hash function `hasher`, predicate `equal` to compare `key_type` objects for equality, and the allocator `alloc` to allocate the memory.

Requirements: the type `InputIterator` must meet the requirements of `InputIterator` from the [input.iterators] ISO C++ Standard section.

```
concurrent_unordered_set( std::initializer_list<value_type> init,
                          size_type bucket_count = /*implementation-defined*/,
                          const hasher& hash = hasher(),
                          const key_equal& equal = key_equal(),
                          const allocator_type& alloc = allocator_type() );
```

Equivalent to `concurrent_unordered_set(init.begin(), init.end(), bucket_count, hash, equal, alloc)`.

```
concurrent_unordered_set( std::initializer_list<value_type> init,
                          size_type bucket_count, const allocator_type& alloc␣
↪);
```

Equivalent to `concurrent_unordered_set(init.begin(), init.end(), bucket_count, alloc)`.

```
concurrent_unordered_set( std::initializer_list<value_type> init,
                          size_type bucket_count, const hasher& hash,
                          const allocator_type& alloc );
```

Equivalent to `concurrent_unordered_set(init.begin(), init.end(), bucket_count, hash, alloc)`.

Copying constructors

```
concurrent_unordered_set( const concurrent_unordered_set& other );

concurrent_unordered_set( const concurrent_unordered_set& other,
                          const allocator_type& alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Moving constructors

```
concurrent_unordered_set( concurrent_unordered_set&& other );

concurrent_unordered_set( concurrent_unordered_set&& other,
                          const allocator_type& alloc );
```

Constructs a *concurrent_unordered_set* with the contents of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by calling `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Destructor

```
~concurrent_unordered_set();
```

Destroys the `concurrent_unordered_set`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

Assignment operators

```
concurrent_unordered_set& operator=( const concurrent_unordered_set& other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_assignment::` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_unordered_set& operator=( concurrent_unordered_set&& other )␣
↪noexcept(/*See below*/);
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment:` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

Exceptions: `noexcept` specification:

```
noexcept(std::allocator_traits<allocator_type>::is_always_equal::value␣
↪&&
         std::is_nothrow_move_assignable<hasher>::value &&
         std::is_nothrow_move_assignable<key_equal>::value)
```

```
concurrent_unordered_set& operator=( std::initializer_list<value_type> init );
```

Replaces all elements in `*this` by the elements in `init`.

If `init` contains multiple equal elements, it is unspecified which element would be inserted.

The behavior is undefined in case of concurrent operations with `*this`.

Returns: a reference to `*this`.

Iterators

The types `concurrent_unordered_set::iterator` and `concurrent_unordered_set::const_iterator` meet the requirements of `ForwardIterator` from the [forward.iterators] ISO C++ Standard section.

begin and cbegin

```
iterator begin();

const_iterator begin() const;

const_iterator cbegin() const;
```

Returns: an iterator to the first element in the container.

end and cend

```
iterator end();

const_iterator end() const;

const_iterator cend() const;
```

Returns: an iterator to the element that follows the last element in the container.

Size and capacity

empty

```
bool empty() const;
```

Returns: `true` if the container is empty; `false`, otherwise.

The result may differ from the actual container state in case of pending concurrent insertions.

size

```
size_type size() const;
```

Returns: the number of elements in the container.

The result may differ from the actual container size in case of pending concurrent insertions.

max_size

```
size_type max_size() const;
```

Returns: the maximum number of elements that container can hold.

Concurrently safe modifiers

All member functions in this section can be performed concurrently with each other, lookup methods and while traversing the container.

Inserting values

```
std::pair<iterator, bool> insert( const value_type& value );
```

Attempts to insert the value `value` into the container.

Returns: `std::pair<iterator, bool>` where `iterator` points to the inserted element or to an existing equal element. Boolean value is `true` if insertion took place; `false`, otherwise.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, const value_type& other );
```

Attempts to insert the value `value` into the container.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

Returns: an `iterator` to the inserted element or to an existing equal element.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
std::pair<iterator, bool> insert( value_type&& value );
```

Attempts to insert the value `value` into the container using move semantics.

`value` is left in a valid, but unspecified state.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element or to an existing equal element. Boolean value is `true` if insertion took place; `false`, otherwise.

Requirements: the type `value_type` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, value_type&& other );
```

Attempts to insert the value `value` into the container using move semantics.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

`value` is left in a valid, but unspecified state.

Returns: an `iterator` to the inserted element or to an existing equal element.

Requirements: the type `value_type` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

Inserting sequences of elements

```
template <typename InputIterator>
void insert( InputIterator first, InputIterator last );
```

Attempts to insert all items from the half-open interval `[first, last)` into the container.

If the interval `[first, last)` contains multiple equal elements, it is unspecified which element should be inserted.

Requirements: the type `InputIterator` must meet the requirements of *InputIterator* from `[input.iterators]` ISO C++ Standard section.

```
void insert( std::initializer_list<value_type> init );
```

Equivalent to `insert(init.begin(), init.end())`.

Inserting nodes

```
std::pair<iterator, bool> insert( node_type&& nh );
```

If the node handle `nh` is empty, does nothing.

Otherwise, attempts to insert the node owned by `nh` into the container.

If the insertion fails, node handle `nh` keeps ownership of the node.

Otherwise, `nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element or to an existing element equal to `nh.value()`. Boolean value is `true` if insertion took place; `false`, otherwise.

```
iterator insert( const_iterator hint, node_type&& nh );
```

If the node handle `nh` is empty, does nothing.

Otherwise, attempts to insert the node owned by `nh` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

If the insertion fails, node handle `nh` keeps ownership of the node.

Otherwise, `nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: an iterator pointing to the inserted element or to an existing element equal to `nh.value()`.

Emplacing elements

```
template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args );
```

Attempts to insert an element constructed in-place from `args` into the container.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element or to an existing equal element. Boolean value is `true` if insertion took place, `false` otherwise.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ Standard section.

```
template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args );
```

Attempts to insert an element constructed in-place from `args` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

Returns: an `iterator` to the inserted element or to an existing equal element.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ Standard section.

Merging containers

```
template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_set<T, SrcHash, SrcKeyEqual, Allocator>&␣
↪source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_set<T, SrcHash, SrcKeyEqual, Allocator>&&␣
↪source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_multiset<T, SrcHash, SrcKeyEqual, Allocator>&␣
↪source );
```

(continues on next page)

(continued from previous page)

```
template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_multiset<T, SrcHash, SrcKeyEqual, Allocator>&&
↪ source );
```

Transfers those elements from `source` that do not exist in the container.

In case of merging with the container with multiple equal elements, it is unspecified which element would be transferred.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `get_allocator() != source.get_allocator()`.

Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

Clearing

```
void clear();
```

Removes all elements from the container.

Erasing elements

```
iterator unsafe_erase( const_iterator pos );

iterator unsafe_erase( iterator pos );
```

Removes the element pointed to by `pos` from the container.

Invalidates all iterators and references to the removed element.

Returns: `iterator` that follows the removed element.

Requirements: the iterator `pos` should be valid, dereferenceable, and point to the element in `*this`.

```
size_type unsafe_erase( const key_type& key );
```

Removes the element equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed element.

Returns: 1 if an element equivalent to `key` exists; `0`, otherwise.

```
template <typename K>
size_type unsafe_erase( const K& key );
```

Removes the element equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed element.

This overload only participates in overload resolution if all of the following statements are `true`:

- The qualified-id `hasher::transparent_key_equal` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: 1 if an element equivalent to `key` exists; `0`, otherwise.

Erasing sequences

```
iterator unsafe_erase( const_iterator first, const_iterator last );
```

Removes all elements from the half-open interval `[first, last)` from the container.

Returns: `iterator` that follows the last removed element.

Requirements: the range `[first, last)` must be a valid subrange in `*this`.

Extracting nodes

```
node_type unsafe_extract( iterator pos );

node_type unsafe_extract( const_iterator pos );
```

Transfers ownership of the element pointed to by `pos` from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element.

Requirements: the iterator `pos` should be valid, dereferenceable, and point to the element in `*this`.

```
node_type unsafe_extract( const key_type& key );
```

If an element equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element or an empty node handle if an element equivalent to `key` was not found.

```
template <typename K>
node_type unsafe_extract( const K& key );
```

If an element equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

This overload participates in overload resolution only if all of the following statements are `true`:

- The qualified-id `hasher::transparent_key_equal` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: the node handle that owns the extracted element or an empty node handle if an element equivalent to `key` was not found.

swap

```
void swap( concurrent_unordered_set& other ) noexcept(/*See below*/);
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`.

Otherwise, if `get_allocator() != other.get_allocator()`, the behavior is undefined.

Exceptions: `noexcept` specification:

```
noexcept(std::allocator_traits<allocator_type>::is_always_equal::value␣
↪&&
         std::is_nothrow_swappable<hasher>::value &&
         std::is_nothrow_swappable<key_equal>::value
```

Lookup

All methods in this section can be executed concurrently with each other, concurrently-safe modifiers and while traversing the container.

count

```
size_type count( const key_type& key );
```

Returns: the number of elements equivalent to `key`.

```
template <typename K>
size_type count( const K& key );
```

Returns: the number of elements that is equivalent to `key`.

This overload only participates in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

find

```
iterator find( const key_type& key );

const_iterator find( const key_type& key ) const;
```

Returns: an iterator to the element equivalent to `key`, or `end()` if no such element exists.

```
template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;
```

Returns: an iterator to the element that is equivalent to `key`, or `end()` if no such element exists.

These overloads only participates in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

contains

```
bool contains( const key_type& key ) const;
```

Returns: `true` if an element equivalent to `key` exists in the container; `false`, otherwise.

```
template <typename K>
bool contains( const K& key ) const;
```

Returns: `true` if an element equivalent to `key` exists in the container; `false`, otherwise.

This overload only participates in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

equal_range

```
std::pair<iterator, iterator> equal_range( const key_type& key );

std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;
```

Returns: if an element equivalent to `key` exists, a pair of iterators `{f, l}`, where `f` is an iterator to this element, `l` is `std::next(f)`. Otherwise, `{end(), end()}`.

```
template <typename K>
std::pair<iterator, iterator> equal_range( const K& key )

template <typename K>
std::pair<const_iterator, const_iterator> equal_range( const K& key )
```

Returns: if an element equivalent to `key` exists, a pair of iterators `{f, l}`, where `f` is an iterator to this element, `l` is `std::next(f)`. Otherwise, `{end(), end()}`.

These overloads participate in overload resolution only if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

Bucket interface

The types `concurrent_unordered_set::local_iterator` and `concurrent_unordered_set::const_local_iterator` meet the requirements of `ForwardIterator` from the [forward.iterators] ISO C++ Standard section.

Use these iterators to traverse the certain bucket.

All methods in this section can only be executed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

Bucket begin and bucket end

```
local_iterator unsafe_begin( size_type n );

const_local_iterator unsafe_begin( size_type n ) const;

const_local_iterator unsafe_cbegin( size_type n ) const;
```

Returns: an iterator to the first element in the bucket number `n`.

```
local_iterator unsafe_end( size_type n );

const_local_iterator unsafe_end( size_type n ) const;

const_local_iterator unsafe_cend( size_type n ) const;
```

Returns: an iterator to the element that follows the last element in the bucket number `n`.

The number of buckets

```
size_type unsafe_bucket_count() const;
```

Returns: the number of buckets in the container.

```
size_type unsafe_max_bucket_count() const;
```

Returns: the maximum number of buckets that container can hold.

Size of the bucket

```
size_type unsafe_bucket_size( size_type n ) const;
```

Returns: the number of elements in the bucket number `n`.

Bucket number

```
size_type unsafe_bucket( const key_type& key ) const;
```

Returns: the number of the bucket in which the element with the key `key` is stored.

Hash policy

Hash policy of `concurrent_unordered_set` manages the number of buckets in the container and the allowed maximum number of elements per bucket (load factor). If the maximum load factor is exceeded, the container can automatically increase the number of buckets.

Load factor

```
float load_factor() const;
```

Returns: the average number of elements per bucket, which is `size()/unsafe_bucket_count()`.

```
float max_load_factor() const;
```

Returns: the maximum number of elements per bucket.

```
void max_load_factor( float ml );
```

Sets the maximum number of elements per bucket to `ml`.

Manual rehashing

```
void rehash( size_type n );
```

Sets the number of buckets to `n` and rehashes the container.

```
void reserve( size_type n );
```

Sets the number of buckets to the value that is needed to store `n` elements.

Observers

get_allocator

```
allocator_type get_allocator() const;
```

Returns: a copy of the allocator associated with `*this`.

hash_function

```
hasher hash_function() const;
```

Returns: a copy of the hash function associated with `*this`.

key_eq

```
key_equal key_eq() const;
```

Returns: a copy of the key equality predicate associated with `*this`.

Parallel iteration

Member types `concurrent_unordered_set::range_type` and `concurrent_unordered_set::const_range_type` meet the *ContainerRange requirements*.

These types differ only in that the bounds for a `concurrent_unordered_set::const_range_type` are of type `concurrent_unordered_set::const_iterator`, whereas the bounds for a `concurrent_unordered_set::range_type` are of type `concurrent_unordered_set::iterator`.

range member function

```
range_type range();

const_range_type range() const;
```

Returns: a range object representing all elements in the container.

Non-member functions

These functions provide binary comparison and swap operations on `oneapi::tbb::concurrent_unordered_set` objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define `oneapi::tbb::concurrent_unordered_set` as a type alias for which the non-member functions are reachable only via argument-dependent lookup.

```
template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
void swap( concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& lhs,
           concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& rhs );

template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& lhs,
                 const concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& rhs );

template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& lhs,
                 const concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& rhs );
```

Non-member swap

```
template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
void swap( concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& lhs,
           concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& rhs )␣
↪noexcept(noexcept(lhs.swap(rhs)));
```

Equivalent to `lhs.swap(rhs)`.

Non-member binary comparisons

Two objects of `concurrent_unordered_set` are equal if the following conditions are `true`:

- They contain an equal number of elements.
- Each element from one container is also available in the other.

```
template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& lhs,
                 const concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& rhs );
```

Returns: `true` if `lhs` is equal to `rhs`, `false` otherwise.

```
template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator!=( const concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& lhs,
                 const concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& rhs );
```

Equivalent to `!(lhs == rhs)`.

Returns: `true` if `lhs` is not equal to `rhs`; `false`, otherwise.

Other

Deduction guides

If possible, `concurrent_unordered_set` constructors support class template argument deduction (since C++17). Copy and move constructors, including constructors with an explicit `allocator_type` argument, provide implicitly-generated deduction guides. In addition, the following explicit deduction guides are provided:

```
template <typename InputIterator,
          typename Hash = std::hash<iterator_value_t<InputIterator>>,
          typename KeyEqual = std::equal_to<iterator_value_t<InputIterator>>,
          typename Allocator = tbb::tbb_allocator<iterator_value_t<InputIterator>>>
concurrent_unordered_set( InputIterator, InputIterator,
                          set_size_type = {},
                          Hash = Hash(),
                          KeyEqual = KeyEqual(),
                          Allocator = Allocator() )
-> concurrent_unordered_set<iterator_value_t<InputIterator>,
                            Hash,
                            KeyEqual,
                            Allocator>;

template <typename InputIterator,
          typename Allocator>
concurrent_unordered_set( InputIterator, InputIterator,
                          set_size_type,
                          Allocator )
-> concurrent_unordered_set<iterator_value_t<InputIterator>,
```

(continues on next page)

(continued from previous page)

```
                            std::hash<iterator_value_t<InputIterator>>,
                            std::equal_to<iterator_value_t<InputIterator>>,
                            Allocator>;

template <typename T,
          typename Hash = std::hash<T>,
          typename KeyEqual = std::equal_to<T>,
          typename Allocator = tbb::tbb_allocator<T>>
concurrent_unordered_set( std::initializer_list<T>,
                          set_size_type = {},
                          Hash = Hash(),
                          KeyEqual = KeyEqual(),
                          Allocator = Allocator() )
-> concurrent_unordered_set<T,
                            Hash,
                            KeyEqual,
                            Allocator>;

template <typename T,
          typename Allocator>
concurrent_unordered_set( std::initializer_list<T>,
                          set_size_type,
                          Allocator )
-> concurrent_unordered_set<T,
                            std::hash<T>,
                            std::equal_to<T>,
                            Allocator>;

template <typename T,
          typename Allocator>
concurrent_unordered_set( std::initializer_list<T>,
                          Allocator )
-> concurrent_unordered_set<T,
                            std::hash<T>,
                            std::equal_to<T>,
                            Allocator>;

template <typename T,
          typename Hash,
          typename Allocator>
concurrent_unordered_set( std::initializer_list<T>,
                          set_size_type,
                          Hash,
                          Allocator )
-> concurrent_unordered_set<T,
                            Hash,
                            std::equal_to<T>,
                            Allocator>;
```

Where the `set_size_type` type refers to the `size_type` member type of the deduced `concurrent_unordered_set` and the type alias `iterator_value_t` is defined as follows:

```
template <typename InputIterator>
using iterator_value_t = typename std::iterator_traits<InputIterator>::value_type;
```

These deduction guides only participate in the overload resolution if the following requirements are met:

- The `InputIterator` type meets the `InputIterator` requirements described in the [input.iterators] section of the ISO C++ Standard.
- The `Allocator` type meets the `Allocator` requirements described in the [allocator.requirements] section of the ISO C++ Standard.
- The `Hash` type does not meet the `Allocator` requirements.
- The `KeyEqual` type does not meet the `Allocator` requirements.

Example

```
#include <oneapi/tbb/concurrent_unordered_set.h>
#include <vector>
#include <functional>

struct CustomHasher {...};

int main() {
    std::vector<int> v;

    // Deduces s1 as concurrent_unordered_set<int>
    oneapi::tbb::concurrent_unordered_set s1(v.begin(), v.end());

    // Deduces s2 as concurrent_unordered_set<int, CustomHasher>;
    oneapi::tbb::concurrent_unordered_set s2(v.begin(), v.end(), CustomHasher{});
}
```

concurrent_unordered_multiset

[containers.concurrent_unordered_multiset]

`oneapi::tbb::concurrent_unordered_multiset` is a class template that represents an unordered sequence of elements, It supports concurrent insertion, lookup, and traversal, but does not support concurrent erasure. In this container, multiple equivalent elements can be stored.

Class Template Synopsis

```
// Defined in header <oneapi/tbb/concurrent_unordered_set.h>

namespace oneapi {
    namespace tbb {
        template <typename T,
                  typename Hash = std::hash<Key>,
                  typename KeyEqual = std::equal_to<Key>,
                  typename Allocator = tbb_allocator<std::pair<const Key, T>>>
        class concurrent_unordered_multiset {
        public:
```

(continues on next page)

(continued from previous page)

```
            using key_type = Key;
            using value_type = Key;

            using size_type = <implementation-defined unsigned integer type>;
            using difference_type = <implementation-defined signed integer type>;

            using hasher = Hash;
            using key_equal = /*See below*/;

            using allocator_type = Allocator;

            using reference = value_type&;
            using const_reference = const value_type&;

            using pointer = typename std::allocator_traits<Allocator>::pointer;
            using const_pointer = typename std::allocator_traits<Allocator>::const_
↪pointer;

            using iterator = <implementation-defined ForwardIterator>;
            using const_iterator = <implementation-defined constant ForwardIterator>;

            using local_iterator = <implementation-defined ForwardIterator>;
            using const_local_iterator = <implementation-defined constant␣
↪ForwardIterator>;

            using node_type = <implementation-defined node handle>;

            using range_type = <implementation-defined ContainerRange>;
            using const_range_type = <implementation-defined constant ContainerRange>;

            // Construction, destruction, copying
            concurrent_unordered_multiset();

            explicit concurrent_unordered_multiset( size_type bucket_count, const hasher&
↪ hash = hasher(),
                                                    const key_equal& equal = key_equal(),
                                                    const allocator_type& alloc =␣
↪allocator_type() );

            concurrent_unordered_multiset( size_type bucket_count, const allocator_type&␣
↪alloc );

            concurrent_unordered_multiset( size_type bucket_count, const hasher& hash,
                                           const allocator_type& alloc );

            explicit concurrent_unordered_multiset( const allocator_type& alloc );

            template <typename InputIterator>
            concurrent_unordered_multiset( InputIterator first, InputIterator last,
                                           size_type bucket_count = /*implementation-
↪defined*/,
                                           const hasher& hash = hasher(),
```

(continues on next page)

(continued from previous page)

```
                                        const key_equal& equal = key_equal(),
                                        const allocator_type& alloc = allocator_
↪type() );

            template <typename InputIterator>
            concurrent_unordered_multiset( InputIterator first, InputIterator last,
                                        size_type bucket_count, const allocator_type&␣
↪alloc );

            template <typename InputIterator>
            concurrent_unordered_multiset( InputIterator first, InputIterator last,
                                        size_type bucket_count, const hasher& hash,
                                        const allocator_type& alloc );

            concurrent_unordered_multiset( std::initializer_list<value_type> init,
                                        size_type bucket_count = /*implementation-
↪defined*/,
                                        const hasher& hash = hasher(),
                                        const key_equal& equal = key_equal(),
                                        const allocator_type& alloc = allocator_
↪type() );

            concurrent_unordered_multiset( std::initializer_list<value_type> init,
                                        size_type bucket_count, const allocator_type&␣
↪alloc );

            concurrent_unordered_multiset( std::initializer_list<value_type> init,
                                        size_type bucket_count, const hasher& hash,
                                        const allocator_type& alloc );

            concurrent_unordered_multiset( const concurrent_unordered_multiset& other );
            concurrent_unordered_multiset( const concurrent_unordered_multiset& other,
                                        const allocator_type& alloc );

            concurrent_unordered_multiset( concurrent_unordered_multiset&& other );
            concurrent_unordered_multiset( concurrent_unordered_multiset&& other,
                                        const allocator_type& alloc );

            ~concurrent_unordered_multiset();

            concurrent_unordered_multiset& operator=( const concurrent_unordered_
↪multiset& other );
            concurrent_unordered_multiset& operator=( concurrent_unordered_multiset&&␣
↪other ) noexcept(/*See details*/);

            concurrent_unordered_multiset& operator=( std::initializer_list<value_type>␣
↪init );

            allocator_type get_allocator() const;

            // Iterators
            iterator begin() noexcept;
```

(continues on next page)

(continued from previous page)

```
            const_iterator begin() const noexcept;
            const_iterator cbegin() const noexcept;

            iterator end() noexcept;
            const_iterator end() const noexcept;
            const_iterator cend() const noexcept;

            // Size and capacity
            bool empty() const noexcept;
            size_type size() const noexcept;
            size_type max_size() const noexcept;

            // Concurrently safe modifiers
            std::pair<iterator, bool> insert( const value_type& value );
            iterator insert( const_iterator hint, const value_type& value );

            std::pair<iterator, bool> insert( value_type&& value );
            iterator insert( const_iterator hint, value_type&& value );

            template <typename InputIterator>
            void insert( InputIterator first, InputIterator last );

            void insert( std::initializer_list<value_type> init );

            std::pair<iterator, bool> insert( node_type&& nh );
            iterator insert( const_iterator hint, node_type&& nh );

            template <typename... Args>
            std::pair<iterator, bool> emplace( Args&&... args );

            template <typename... Args>
            iterator emplace_hint( const_iterator hint, Args&&... args );

            template <typename SrcHash, typename SrcKeyEqual>
            void merge( concurrent_unordered_set<T, SrcHash, SrcKeyEqual, Allocator>&␣
↪source );

            template <typename SrcHash, typename SrcKeyEqual>
            void merge( concurrent_unordered_set<T, SrcHash, SrcKeyEqual, Allocator>&&␣
↪source );

            template <typename SrcHash, typename SrcKeyEqual>
            void merge( concurrent_unordered_multiset<T, SrcHash, SrcKeyEqual, Allocator>
↪& source );

            template <typename SrcHash, typename SrcKeyEqual>
            void merge( concurrent_unordered_multiset<T, SrcHash, SrcKeyEqual, Allocator>
↪&& source );

            // Concurrently unsafe modifiers
            void clear() noexcept;
```

(continues on next page)

(continued from previous page)

```
            iterator unsafe_erase( const_iterator pos );
            iterator unsafe_erase( iterator pos );

            iterator unsafe_erase( const_iterator first, const_iterator last );

            size_type unsafe_erase( const key_type& key );

            template <typename K>
            size_type unsafe_erase( const K& key );

            node_type unsafe_extract( const_iterator pos );
            node_type unsafe_extract( iterator pos );

            node_type unsafe_extract( const key_type& key );

            template <typename K>
            node_type unsafe_extract( const K& key );

            void swap( concurrent_unordered_multiset& other );

            // Lookup
            size_type count( const key_type& key ) const;

            template <typename K>
            size_type count( const K& key ) const;

            iterator find( const key_type& key );
            const_iterator find( const key_type& key ) const;

            template <typename K>
            iterator find( const K& key );

            template <typename K>
            const_iterator find( const K& key ) const;

            bool contains( const key_type& key ) const;

            template <typename K>
            bool contains( const K& key ) const;

            std::pair<iterator, iterator> equal_range( const key_type& key );
            std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;

            template <typename K>
            std::pair<iterator, iterator> equal_range( const K& key );

            template <typename K>
            std::pair<const_iterator, const_iterator> equal_range( const K& key ) const;

            // Bucket interface
            local_iterator unsafe_begin( size_type n );
```

(continues on next page)

(continued from previous page)

```
            const_local_iterator unsafe_begin( size_type n ) const;
            const_local_iterator unsafe_cbegin( size_type n ) const;

            local_iterator unsafe_end( size_type n );
            const_local_iterator unsafe_end( size_type n ) const;
            const_local_iterator unsafe_cend( size_type n ) const;

            size_type unsafe_bucket_count() const;
            size_type unsafe_max_bucket_bount() const;

            size_type unsafe_bucket_size( size_type n ) const;

            size_type unsafe_bucket( const key_type& key ) const;

            // Hash policy
            float load_factor() const;

            float max_load_factor() const;
            void max_load_factor( float ml );

            void rehash( size_type count );

            void reserve( size_type count );

            // Observers
            hasher hash_function() const;
            key_equal key_eq() const;

            // Parallel iteration
            range_type range();
            const_range_type range() const;
        }; // class concurrent_unordered_multiset
    } // namespace tbb
} // namespace oneapi
```

Requirements:

- The expression `std::allocator_type<Allocator>::destroy(m, val)`, where `m` is an object of the type `Allocator` and `val` is an object of type `value_type`, must be well-formed. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Hash` must meet the `Hash` requirements from the [hash] ISO C++ Standard section.
- The type `KeyEqual` must meet the `BinaryPredicate` requirements from the [algorithms.general] ISO C++ Standard section.
- The type `Allocator` must meet the `Allocator` requirements from the [allocator.requirements] ISO C++ Standard section.

Description

`oneapi::tbb::concurrent_unordered_multiset` is an unordered sequence, which elements are organized into buckets. The value of the hash function `Hash` for `Key` object determines the number of the bucket in which the corresponding element will be placed.

If the qualified-id `Hash::transparent_key_equal` is valid and denotes a type, the member type `concurrent_unordered_multiset::key_equal` is defined as the value of this qualified-id. In this case, the program is ill-formed if any of the following conditions are met:

- The template parameter `KeyEqual` is different from `std::equal_to<Key>`.
- Qualified-id `Hash::transparent_key_equal::is_transparent` is not valid or does not denote a type.

Otherwise, the member type `concurrent_unordered_multiset::key_equal` is defined as the value of the template parameter `KeyEqual`.

Member functions

Construction, destruction, copying

Empty container constructors

```
concurrent_unordered_multiset();

explicit concurrent_unordered_multiset( const allocator_type& alloc );
```

Constructs an empty `concurrent_unordered_multiset`. The initial number of buckets is unspecified.

If provided, uses the allocator `alloc` to allocate the memory.

```
explicit concurrent_unordered_multiset( size_type bucket_count,
                                        const hasher& hash = hasher(),
                                        const key_equal& equal = key_equal(),
                                        const allocator_type& alloc =␣
↪allocator_type() );

concurrent_unordered_multiset( size_type bucket_count, const allocator_type&␣
↪alloc );

concurrent_unordered_multiset( size_type bucket_count, const hasher& hash,
                               const allocator_type& alloc );
```

Constructs an empty `concurrent_unordered_multiset` with `bucket_count` buckets.

If provided, uses the hash function `hasher`, predicate `equal` to compare `key_type` objects for equality, and the allocator `alloc` to allocate the memory.

Constructors from the sequence of elements

```
template <typename InputIterator>
concurrent_unordered_multiset( InputIterator first, InputIterator last,
                               size_type bucket_count = /*implementation-
↪defined*/,
                               const hasher& hash = hasher(),
                               const key_equal& equal = key_equal(),
                               const allocator_type& alloc = allocator_type()␣
↪);

template <typename Inputiterator>
concurrent_unordered_multiset( InputIterator first, InputIterator last,
                               size_type bucket_count, const allocator_type&␣
↪alloc );

template <typename InputIterator>
concurrent_unordered_multiset( InputIterator first, InputIterator last,
                               size_type bucket_count, const hasher& hash,
                               const allocator_type& alloc );
```

Constructs the `concurrent_unordered_multiset`, which contains the elements from the half-open interval `[first, last)`.

If provided uses the hash function `hasher`, predicate `equal` to compare `key_type` objects for equality, and the allocator `alloc` to allocate the memory.

Requirements: the type `InputIterator` must meet the requirements of `InputIterator` from the [input.iterators] ISO C++ Standard section.

```
concurrent_unordered_multiset( std::initializer_list<value_type> init,
                               size_type bucket_count = /*implementation-
↪defined*/,
                               const hasher& hash = hasher(),
                               const key_equal& equal = key_equal(),
                               const allocator_type& alloc = allocator_type()␣
↪);
```

Equivalent to `concurrent_unordered_multiset(init.begin(), init.end(), bucket_count, hash, equal, alloc)`.

```
concurrent_unordered_multiset( std::initializer_list<value_type> init,
                               size_type bucket_count, const allocator_type&␣
↪alloc );
```

Equivalent to `concurrent_unordered_multiset(init.begin(), init.end(), bucket_count, alloc)`.

```
concurrent_unordered_multiset( std::initializer_list<value_type> init,
                               size_type bucket_count, const hasher& hash,
                               const allocator_type& alloc );
```

Equivalent to `concurrent_unordered_multiset(init.begin(), init.end(), bucket_count, hash, alloc)`.

Copying constructors

```
concurrent_unordered_multiset( const concurrent_unordered_multiset& other );

concurrent_unordered_multiset( const concurrent_unordered_multiset& other,
                               const allocator_type& alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Moving constructors

```
concurrent_unordered_multiset( concurrent_unordered_multiset&& other );

concurrent_unordered_multiset( concurrent_unordered_multiset&& other,
                               const allocator_type& alloc );
```

Constructs a `concurrent_unordered_multiset` with the contents of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by calling `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Destructor

```
~concurrent_unordered_multiset();
```

Destroys the `concurrent_unordered_multiset`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

Assignment operators

```
concurrent_unordered_multiset& operator=( const concurrent_unordered_multiset&␣
↪other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_assignment::` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_unordered_multiset& operator=( concurrent_unordered_multiset&&␣
↪other ) noexcept(/*See below*/);
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment:` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

Exceptions: `noexcept` specification:

```
noexcept(std::allocator_traits<allocator_type>::is_always_equal::value␣
↪&&
         std::is_nothrow_move_assignable<hasher>::value &&
         std::is_nothrow_move_assignable<key_equal>::value)
```

```
concurrent_unordered_multiset& operator=( std::initializer_list<value_type>␣
↪init );
```

Replaces all elements in `*this` by the elements in `init`.

The behavior is undefined in case of concurrent operations with `*this`.

Returns: a reference to `*this`.

Iterators

The types `concurrent_unordered_multiset::iterator` and `concurrent_unordered_multiset::const_iterator` meet the requirements of `ForwardIterator` from the [forward.iterators] ISO C++ Standard section.

begin and cbegin

```
iterator begin();

const_iterator begin() const;

const_iterator cbegin() const;
```

Returns: an iterator to the first element in the container.

end and cend

```
iterator end();

const_iterator end() const;

const_iterator cend() const;
```

Returns: an iterator to the element that follows the last element in the container.

Size and capacity

empty

```
bool empty() const;
```

Returns: `true` if the container is empty; `false`, otherwise.

The result may differ with the actual container state in case of pending concurrent insertions.

size

```
size_type size() const;
```

Returns: the number of elements in the container.

The result may differ with the actual container size in case of pending concurrent insertions.

max_size

```
size_type max_size() const;
```

Returns: the maximum number of elements that container can hold.

Concurrently safe modifiers

All member functions in this section can be performed concurrently with each other, lookup methods and while traversing the container.

Inserting values

```
std::pair<iterator, bool> insert( const value_type& value )
```

Inserts the value `value` into the container.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element. Boolean value is always `true`.

```
iterator insert( const_iterator hint, const value_type& other )
```

Inserts the value `value` into the container.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

Returns: an `iterator` to the inserted element.

```
std::pair<iterator, bool> insert( value_type&& value )
```

Inserts the value `value` into the container using move semantics.

`value` is left in a valid, but unspecified state.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element. Boolean value is always `true`.

```
iterator insert( const_iterator hint, value_type&& other )
```

Inserts the value `value` into the container using move semantics.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

`value` is left in a valid, but unspecified state.

Returns: an `iterator` to the inserted element.

Inserting sequences of elements

```
template <typename InputIterator>
void insert( InputIterator first, InputIterator last )
```

Inserts all items from the half-open interval `[first, last)` into the container.

Requirements: the type `InputIterator` must meet the requirements of *InputIterator* from the `[input.iterators]` ISO C++ Standard section.

```
void insert( std::initializer_list<value_type> init )
```

Equivalent to `insert(init.begin(), init.end())`.

Inserting nodes

```
std::pair<iterator, bool> insert( node_type&& nh )
```

If the node handle `nh` is empty, does nothing.

Otherwise - inserts the node, owned by `nh` into the container.

`nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: `std::pair<iterator, bool>` where `iterator` points to the inserted element. Boolean value is always `true`.

```
iterator insert( const_iterator hint, node_type&& nh )
```

If the node handle `nh` is empty, does nothing.

Otherwise - inserts the node, owned by `nh` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

`nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: an iterator pointing to the inserted element.

Emplacing elements

```
template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args )
```

Inserts an element ,constructed in-place from `args` into the container.

Returns: `std::pair<iterator, bool>` where `iterator` points to the inserted element. Boolean value is always `true`.

```
template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args )
```

Inserts an element ,constructed in-place from `args` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

Returns: an `iterator` to the inserted element.

Merging containers

```
template <typename SrcHash, SrcKeyEqual>
void merge( concurrent_unordered_set<T, SrcHash, SrcKeyEqual, Allocator>&␣
↪source )

template <typename SrcHash, SrcKeyEqual>
void merge( concurrent_unordered_set<T, SrcHash, SrcKeyEqual, Allocator>&&␣
↪source )

template <typename SrcHash, SrcKeyEqual>
void merge( concurrent_unordered_multiset<T, SrcHash, SrcKeyEqual, Allocator>&␣
↪source )

template <typename SrcHash, SrcKeyEqual>
void merge( concurrent_unordered_multiset<T, SrcHash, SrcKeyEqual, Allocator>&&
↪ source )
```

Transfers all elements from `source` to `*this`.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `get_allocator() != source.get_allocator()`.

Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

Clearing

```
void clear();
```

Removes all elements from the container.

Erasing elements

```
iterator unsafe_erase( const_iterator pos );

iterator unsafe_erase( iterator pos );
```

Removes the element pointed to by `pos` from the container.

Invalidates all iterators and references to the removed element.

Returns: `iterator` that follows the removed element.

Requirements: the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
size_type unsafe_erase( const key_type& key );
```

Removes the element equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed element.

Returns: the number of removed elements.

```
template <typename K>
size_type unsafe_erase( const K& key );
```

Removes the element that is equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed element.

This overload only participates in overload resolution if all of the following conditions are met:

- The qualified-id `hasher::transparent_key_equal` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: the number of removed elements.

Erasing sequences

```
iterator unsafe_erase( const_iterator first, const_iterator last );
```

Removes all elements from the half-open interval `[first, last)` from the container.

Returns: `iterator` that follows the last removed element.

Requirements: the range `[first, last)` must be a valid subrange in `*this`.

Extracting nodes

```
node_type unsafe_extract( iterator pos );

node_type unsafe_extract( const_iterator pos );
```

Transfers ownership of the element pointed to by `pos` from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element.

Requirements: the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
node_type unsafe_extract( const key_type& key );
```

If an element equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

If there are multiple elements equivalent to `key`, it is unspecified which element should be transferred.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element or an empty node handle if an element equivalent to `key` was not found.

```
template <typename K>
node_type unsafe_extract( const K& key );
```

If an element equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

If there are multiple elements which are equivalent to `key`, it is unspecified which element should be transferred.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

This overload only participates in overload resolution if all of the following conditions are met:

- The qualified-id `hasher::transparent_key_equal` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: the node handle that owns the extracted element or an empty node handle if an element equivalent to `key` was not found.

swap

```
void swap( concurrent_unordered_multiset& other ) noexcept(/*See below*/);
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`.

Otherwise, if `get_allocator() != other.get_allocator()`, the behavior is undefined.

Exceptions: `noexcept` specification:

```
noexcept(std::allocator_traits<allocator_type>::is_always_equal::value␣
↪&&
         std::is_nothrow_swappable<hasher>::value &&
         std::is_nothrow_swappable<key_equal>::value
```

Lookup

All methods in this section can be executed concurrently with each other, concurrently-safe modifiers and while traversing the container.

count

```
size_type count( const key_type& key );
```

Returns: the number of elements equivalent to `key`.

```
template <typename K>
size_type count( const K& key );
```

Returns: the number of elements that are equivalent to `key`.

This overload only participates in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

find

```
iterator find( const key_type& key );

const_iterator find( const key_type& key ) const;
```

Returns: an iterator to the element equivalent to `key`, or `end()` if no such element exists.

If there are multiple elements equivalent to `key`, it is unspecified which element should be found.

```
template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;
```

Returns: an iterator to the element equivalent to `key`, or `end()` if no such element exists.

If there are multiple elements equivalent to `key`, it is unspecified which element should be found.

These overloads only participate in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

contains

```
bool contains( const key_type& key ) const;
```

Returns: `true` if at least one element equivalent to `key` exists in the container; `false`, otherwise.

```
template <typename K>
bool contains( const K& key ) const;
```

Returns: `true` if at least one element equal to `key` exists in the container; `false`, otherwise.

This overload only participates in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

equal_range

```
std::pair<iterator, iterator> equal_range( const key_type& key );

std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;
```

Returns: if at least one element with the key equivalent to `key` exists, a pair of iterators `{f, l}`, where `f` is an iterator to the first element equivalent to `key`, `l` is an iterator to the element that follows the last element equivalent to `key`. Otherwise, `{end(), end()}`.

```
template <typename K>
std::pair<iterator, iterator> equal_range( const K& key )

template <typename K>
std::pair<const_iterator, const_iterator> equal_range( const K& key )
```

Returns: if at least one element with the key equivalent to `key` exists, a pair of iterators `{f, l}`, where `f` is an iterator to the first element equivalent to `key`, `l` is an iterator to the element that follows the last element equivalent to `key`. Otherwise, `{end(), end()}`.

These overloads participate in overload resolution only if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

Bucket interface

The types `concurrent_unordered_multiset::local_iterator` and `concurrent_unordered_multiset::const_local_iter` meet the requirements of `ForwardIterator` from the [forward.iterators] ISO C++ Standard section.

These iterators are used to traverse the certain bucket.

All methods in this section can only be executed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

Bucket begin and bucket end

```
local_iterator unsafe_begin( size_type n );

const_local_iterator unsafe_begin( size_type n ) const;

const_local_iterator unsafe_cbegin( size_type n ) const;
```

Returns: an iterator to the first element in the bucket number `n`.

```
local_iterator unsafe_end( size_type n );

const_local_iterator unsafe_end( size_type n ) const;

const_local_iterator unsafe_cend( size_type n ) const;
```

Returns: an iterator to the element that follows the last element in the bucket number `n`.

The number of buckets

```
size_type unsafe_bucket_count() const;
```

Returns: the number of buckets in the container.

```
size_type unsafe_max_bucket_count() const;
```

Returns: the maximum number of buckets that container can hold.

Size of the bucket

```
size_type unsafe_bucket_size( size_type n ) const;
```

Returns: the number of elements in the bucket number `n`.

Bucket number

```
size_type unsafe_bucket( const key_type& key ) const;
```

Returns: the number of the bucket in which the element with the key `key` is stored.

Hash policy

Hash policy of `concurrent_unordered_multiset` manages the number of buckets in the container and the allowed maximum number of elements per bucket (load factor). If the maximum load factor is exceeded, the container can automatically increase the number of buckets.

Load factor

```
float load_factor() const;
```

Returns: the average number of elements per bucket, which is `size()/unsafe_bucket_count()`.

```
float max_load_factor() const;
```

Returns: the maximum number of elements per bucket.

```
void max_load_factor( float ml );
```

Sets the maximum number of elements per bucket to `ml`.

Manual rehashing

```
void rehash( size_type n );
```

Sets the number of buckets to `n` and rehashes the container.

```
void reserve( size_type n );
```

Sets the number of buckets to the value that is needed to store `n` elements.

Observers

get_allocator

```
allocator_type get_allocator() const;
```

Returns: a copy of the allocator associated with `*this`.

hash_function

```
hasher hash_function() const;
```

Returns: a copy of the hash function associated with `*this`.

key_eq

```
key_equal key_eq() const;
```

Returns: a copy of the key equality predicate associated with `*this`.

Parallel iteration

Member types `concurrent_unordered_multiset::range_type` and `concurrent_unordered_multiset::const_range_type` meet the *ContainerRange requirements*.

These types differ only in that the bounds for a `concurrent_unordered_multiset::const_range_type` are of type `concurrent_unordered_multiset::const_iterator`, whereas the bounds for a `concurrent_unordered_multiset::range_type` are of type `concurrent_unordered_multiset::iterator`.

range member function

```
range_type range();

const_range_type range() const;
```

Returns: a range object representing all elements in the container.

Non-member functions

These functions provide binary comparison and swap operations on `oneapi::tbb::concurrent_unordered_multiset` objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define `oneapi::tbb::concurrent_unordered_multiset` as a type alias for which the non-member functions are reachable only via argument-dependent lookup.

```
template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
void swap( concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>& lhs,
           concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>& rhs );

template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>& lhs,
                 const concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>& rhs␣
↪);
```

(continues on next page)

(continued from previous page)

```
template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>& lhs,
                 const concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>& rhs␣
↪);
```

Non-member swap

```
template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
void swap( concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>& lhs,
           concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>& rhs )␣
↪noexcept(noexcept(lhs.swap(rhs)));
```

Equivalent to `lhs.swap(rhs)`.

Non-member binary comparisons

Two objects of `concurrent_unordered_multiset` are equal if the following conditions are `true`:

- They contain an equal number of elements.
- Each group of elements with the same key in one container has the corresponding group of equivalent elements in the other container (not necessarily in the same order).

```
template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>& lhs,
                 const concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>& rhs␣
↪);
```

Returns: `true` if `lhs` is equal to `rhs`; `false`, otherwise.

```
template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator!=( const concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>& lhs,
                 const concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>& rhs␣
↪);
```

Equivalent to `!(lhs == rhs)`.

Returns: `true` if `lhs` is not equal to `rhs`, `false` otherwise.

Other

Deduction guides

If possible, `concurrent_unordered_multiset` constructors support class template argument deduction (since C++17). Copy and move constructors, including constructors with an explicit `allocator_type` argument, provide implicitly-generated deduction guides. In addition, the following explicit deduction guides are provided:

```
template <typename InputIterator,
          typename Hash = std::hash<iterator_value_t<InputIterator>>,
          typename KeyEqual = std::equal_to<iterator_value_t<InputIterator>>,
          typename Allocator = tbb::tbb_allocator<iterator_value_t<InputIterator>>>
concurrent_unordered_multiset( InputIterator, InputIterator,
                               set_size_type = {},
                               Hash = Hash(),
                               KeyEqual = KeyEqual(),
                               Allocator = Allocator() )
-> concurrent_unordered_multiset<iterator_value_t<InputIterator>,
                                 Hash,
                                 KeyEqual,
                                 Allocator>;

template <typename InputIterator,
          typename Allocator>
concurrent_unordered_multiset( InputIterator, InputIterator,
                               set_size_type,
                               Allocator )
-> concurrent_unordered_multiset<iterator_value_t<InputIterator>,
                                 std::hash<iterator_value_t<InputIterator>>,
                                 std::equal_to<iterator_value_t<InputIterator>>,
                                 Allocator>;

template <typename T,
          typename Hash = std::hash<T>,
          typename KeyEqual = std::equal_to<T>,
          typename Allocator = tbb::tbb_allocator<T>>
concurrent_unordered_multiset( std::initializer_list<T>,
                               set_size_type = {},
                               Hash = Hash(),
                               KeyEqual = KeyEqual(),
                               Allocator = Allocator() )
-> concurrent_unordered_multiset<T,
                                 Hash,
                                 KeyEqual,
                                 Allocator>;

template <typename T,
          typename Allocator>
concurrent_unordered_multiset( std::initializer_list<T>,
                               set_size_type,
                               Allocator )
-> concurrent_unordered_multiset<T,
                                 std::hash<T>,
```

(continues on next page)

(continued from previous page)

```
                                  std::equal_to<T>,
                                  Allocator>;

template <typename T,
          typename Allocator>
concurrent_unordered_multiset( std::initializer_list<T>,
                               Allocator )
-> concurrent_unordered_multiset<T,
                                  std::hash<T>,
                                  std::equal_to<T>,
                                  Allocator>;

template <typename T,
          typename Hash,
          typename Allocator>
concurrent_unordered_multiset( std::initializer_list<T>,
                               set_size_type,
                               Hash,
                               Allocator )
-> concurrent_unordered_multiset<T,
                                  Hash,
                                  std::equal_to<T>,
                                  Allocator>;
```

Where the `set_size_type` type refers to the `size_type` member type of the deduced `concurrent_unordered_multiset` and the type alias `iterator_value_t` is defined as follows:

```
template <typename InputIterator>
using iterator_value_t = typename std::iterator_traits<InputIterator>::value_type;
```

These deduction guides only participate in the overload resolution if the following requirements are met:

- The `InputIterator` type meets the `InputIterator` requirements described in the [input.iterators] section of the ISO C++ Standard.
- The `Allocator` type meets the `Allocator` requirements described in the [allocator.requirements] section of the ISO C++ Standard.
- The `Hash` type does not meet the `Allocator` requirements.
- The `KeyEqual` type does not meet the `Allocator` requirements.

Example

```
#include <oneapi/tbb/concurrent_unordered_set.h>
#include <vector>
#include <functional>

struct CustomHasher {...};

int main() {
    std::vector<int> v;

    // Deduces s1 as concurrent_unordered_multiset<int>
    oneapi::tbb::concurrent_unordered_multiset s1(v.begin(), v.end());
```

(continues on next page)

(continued from previous page)

```
    // Deduces s2 as concurrent_unordered_multiset<int, CustomHasher>;
    oneapi::tbb::concurrent_unordered_multiset s2(v.begin(), v.end(), CustomHasher{});
}
```

Ordered associative containers

concurrent_map

[containers.concurrent_map]

`oneapi::tbb::concurrent_map` is a class template that represents a sorted associative container. It stores unique elements and supports concurrent insertion, lookup, and traversal, but does not support concurrent erasure.

Class Template Synopsis

```
namespace oneapi {
    namespace tbb {

        template <typename Key,
                  typename T,
                  typename Compare = std::less<Key>,
                  typename Allocator = tbb_allocator<std::pair<const Key, T>>
        class concurrent_map {
        public:
            using key_type = Key;
            using mapped_type = T;
            using value_type = std::pair<const Key, T>;

            using size_type = <implementation-defined unsigned integer type>;
            using difference_type = <implementation-defined signed integer type>;

            using key_compare = Compare;
            using allocator_type = Allocator;

            using reference = value_type&;
            using const_reference = const value_type&;
            using pointer = std::allocator_traits<Allocator>::pointer;
            using const_pointer = std::allocator_traits<Allocator>::const_pointer;

            using iterator = <implementation-defined ForwardIterator>;
            using const_iterator = <implementation-defined constant ForwardIterator>;

            using node_type = <implementation-defined node handle>;

            using range_type = <implementation-defined range>;
            using const_range_type = <implementation-defined constant node handle>;

            class value_compare;
```

(continues on next page)

(continued from previous page)

```
            // Construction, destruction, copying
            concurrent_map();
            explicit concurrent_map( const key_compare& comp,
                                     const allocator_type& alloc = allocator_type() );

            explicit concurrent_map( const allocator_type& alloc );

            template <typename InputIterator>
            concurrent_map( InputIterator first, InputIterator last,
                            const key_compare& comp = key_compare(),
                            const allocator_type& alloc = allocator_type() );

            template <typename InputIterator>
            concurrent_map( InputIterator first, InputIterator last,
                            const allocator_type& alloc );

            concurrent_map( std::initializer_list<value_type> init,
                            const key_compare& comp = key_compare(),
                            const allocator_type& alloc = allocator_type() );

            concurrent_map( std::initializer_list<value_type> init, const allocator_type&
↪ alloc );

            concurrent_map( const concurrent_map& other );
            concurrent_map( const concurrent_map& other,
                        const allocator_type& alloc );

            concurrent_map( concurrent_map&& other );
            concurrent_map( concurrent_map&& other,
                            const allocator_type& alloc );

            ~concurrent_map();

            concurrent_map& operator=( const concurrent_map& other );
            concurrent_map& operator=( concurrent_map&& other );
            concurrent_map& operator=( std::initializer_list<value_type> init );

            allocator_type get_allocator() const;

            // Element access
            value_type& at( const key_type& key );
            const value_type& at( const key_type& key ) const;

            value_type& operator[]( const key_type& key );
            value_type& operator[]( key_type&& key );

            // Iterators
            iterator begin();
            const_iterator begin() const;
            const_iterator cbegin() const;

            iterator end();
```

(continues on next page)

(continued from previous page)

```
        const_iterator end() const;
        const_iterator cend() const;

        // Size and capacity
        bool empty() const;
        size_type size() const;
        size_type max_size() const;

        // Concurrently safe modifiers
        std::pair<iterator, bool> insert( const value_type& value );

        iterator insert( const_iterator hint, const value_type& value );

        template <typename P>
        std::pair<iterator, bool> insert( P&& value );

        template <typename P>
        iterator insert( const_iterator hint, P&& value );

        std::pair<iterator, bool> insert( value_type&& value );

        iterator insert( const_iterator hint, value_type&& value );

        template <typename InputIterator>
        void insert( InputIterator first, InputIterator last );

        void insert( std::initializer_list<value_type> init );

        std::pair<iterator, bool> insert( node_type&& nh );
        iterator insert( const_iterator hint, node_type&& nh );

        template <typename... Args>
        std::pair<iterator, bool> emplace( Args&&... args );

        template <typename... Args>
        iterator emplace_hint( const_iterator hint, Args&&... args );

        template <typename SrcCompare>
        void merge( concurrent_map<Key, T, SrcCompare, Allocator>& source );

        template <typename SrcCompare>
        void merge( concurrent_map<Key, T, SrcCompare, Allocator>&& source );

        template <typename SrcCompare>
        void merge( concurrent_multimap<Key, T, SrcCompare, Allocator>& source );

        template <typename SrcCompare>
        void merge( concurrent_multimap<Key, T, SrcCompare, Allocator>&& source );

        // Concurrently unsafe modifiers
        void clear();
```

(continues on next page)

(continued from previous page)

```
            iterator unsafe_erase( const_iterator pos );
            iterator unsafe_erase( iterator pos );

            iterator unsafe_erase( const_iterator first, const_iterator last );

            size_type unsafe_erase( const key_type& key );

            template <typename K>
            size_type unsafe_erase( const K& key );

            node_type unsafe_extract( const_iterator pos );
            node_type unsafe_extract( iterator pos );

            node_type unsafe_extract( const key_type& key );

            template <typename K>
            node_type unsafe_extract( const K& key );

            void swap( concurrent_map& other );

            // Lookup
            size_type count( const key_type& key );

            template <typename K>
            size_type count( const K& key );

            iterator find( const key_type& key );
            const_iterator find( const key_type& key ) const;

            template <typename K>
            iterator find( const K& key );

            template <typename K>
            const_iterator find( const K& key ) const;

            bool contains( const key_type& key ) const;

            template <typename K>
            bool contains( const K& key ) const;

            std::pair<iterator, iterator> equal_range( const key_type& key );
            std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;

            template <typename K>
            std::pair<iterator, iterator>  equal_range( const K& key );
            std::pair<const_iterator, const_iterator> equal_range( const K& key ) const;

            iterator lower_bound( const key_type& key );
            const_iterator lower_bound( const key_type& key ) const;

            template <typename K>
```

(continues on next page)

(continued from previous page)

```
            iterator lower_bound( const K& key );

            template <typename K>
            const_iterator lower_bound( const K& key ) const;

            iterator upper_bound( const key_type& key );
            const_iterator upper_bound( const key_type& key ) const;

            template <typename K>
            iterator upper_bound( const K& key );

            template <typename K>
            const_iterator upper_bound( const K& key ) const;

            // Observers
            key_compare key_comp() const;

            value_compare value_comp() const;

            // Parallel iteration
            range_type range();
            const_range_type range() const;
        }; // class concurrent_map

    } // namespace tbb
} // namespace oneapi
```

Requirements:

- The expression `std::allocator_traits<Allocator>::destroy(m, val)`, where `m` is an object of the type `Allocator` and `val` is an object of the type `value_type`, must be well-formed. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Compare` must meet the `Compare` requirements from the [alg.sorting] ISO C++ Standard section.
- The type `Allocator` must meet the `Allocator` requirements from the [allocator.requirements] ISO C++ Standard section.

Member classes

value_compare

`concurrent_map::value_compare` is a function object that is used to compare `concurrent_map::value_type` objects by comparing their first components.

Class Synopsis

```
namespace oneapi {
    namespace tbb {

        template <typename Key, typename T,
                  typename Compare, typename Allocator>
        class concurrent_map<Key, T, Compare, Allocator>::value_compare {
        protected:
            key_compare comp;

            value_compare( key_compare c );

        public:
            bool operator()( const value_type& lhs, const value_type& rhs )␣
↪const;
        }; // class value_compare

    } // namespace tbb
} // namespace oneapi
```

Member objects

```
key_compare comp;
```

The key comparison function object.

Member functions

```
value_compare( key_compare c );
```

Constructs a `value_compare` with the stored key comparison function object `c`.

```
bool operator()( const value_type& lhs, const value_type& rhs ) const;
```

Compares `lhs.first` and `rhs.first` by calling the stored key comparison function `comp`.

Returns: `true` if first components of `lhs` and `rhs` are equal; `false`, otherwise.

Member functions

Construction, destruction, copying

Empty container constructors

```
concurrent_map();

explicit concurrent_map( const key_compare& comp,
                         const allocator_type& alloc = allocator_type() );

explicit concurrent_map( const allocator_type& alloc );
```

Constructs an empty `concurrent_map`.

If provided, uses the comparison function object `comp` for all `key_type` comparisons and the allocator `alloc` to allocate the memory.

Constructors from the sequence of elements

```
template <typename InputIterator>
concurrent_map( InputIterator first, InputIterator last,
                const key_compare& comp = key_compare(),
                const allocator_type& alloc = allocator_type() );

template <typename InputIterator>
concurrent_map( InputIterator first, InputIterator last,
                const allocator_type& alloc = allocator_type() );
```

Constructs the `concurrent_map`, which contains the elements from the half-open interval `[first, last)`.

If the range `[first, last)` contains multiple elements with equal keys, it is unspecified which element would be inserted.

If provided, uses the comparison function object `comp` for all `key_type` comparisons and the allocator `alloc` to allocate the memory.

Requirements: the type `InputIterator` must meet the requirements of *InputIterator* from the `[input.iterators]` ISO C++ Standard section.

```
concurrent_map( std::initializer_list<value_type> init, const key_compare&␣
↪comp = key_compare(),
                const allocator_type& alloc = allocator_type() );
```

Equivalent to `concurrent_map(init.begin(), init.end(), comp, alloc)`.

```
concurrent_map( std::initializer_list<value_type> init,
                const allocator_type& alloc );
```

Equivalent to `concurrent_map(init.begin(), init.end(), alloc)`.

Copying constructors

```
concurrent_map( const concurrent_map& other );

concurrent_map( const concurrent_map& other, const allocator_type& alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Moving constructors

```
concurrent_map( concurrent_map&& other );

concurrent_map( concurrent_map&& other, const allocator_type& alloc );
```

Constructs a *concurrent_map* with the contents of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by calling `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Destructor

```
~concurrent_map();
```

Destroys the `concurrent_map`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

Assignment operators

```
concurrent_map& operator=( const concurrent_map& other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_assignment::` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_map& operator=( concurrent_map&& other );
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment:` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_map& operator=( std::initializer_list<value_type> init );
```

Replaces all elements in `*this` by the elements in `init`.

If `init` contains multiple elements with equal keys, it is unspecified which element would be inserted.

The behavior is undefined in case of concurrent operations with `*this`.

Returns: a reference to `*this`.

Element access

at

```
value_type& at( const key_type& key );

const value_type& at( const key_type& key ) const;
```

Returns: a reference to `item.second`, where `item` is the element with the key equivalent to `key`.

Throws: `std::out_of_range` exception if the element with the key equivalent to `key` is not present in the container.

operator[]

```
value_type& operator[]( const key_type& key );
```

If the element with the key equivalent to `key` is not present in the container, inserts a new element constructed in-place from `std::piecewise_construct, std::forward_as_tuple(key), std::tuple<>()`.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ section.

Returns: a reference to `item.second`, where `item` is the element with the key equivalent to `key`.

```
value_type& operator[]( key_type&& key );
```

If the element with the key equivalent to `key` is not present in the container, inserts a new element, constructed in-place from `std::piecewise_construct, std::forward_as_tuple(std::move(key)), std::tuple<>()`.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ section.

Returns: a reference to `item.second`, where `item` is the element with the key equivalent to `key`.

Iterators

The types `concurrent_map::iterator` and `concurrent_map::const_iterator` meet the requirements of `ForwardIterator` from the [forward.iterators] ISO C++ standard section.

begin and cbegin

```
iterator begin();

const_iterator begin() const;

const_iterator cbegin() const;
```

Returns: an iterator to the first element in the container.

end and cend

```
iterator end();

const_iterator end() const;

const_iterator cend() const;
```

Returns: an iterator to the element that follows the last element in the container.

Size and capacity

empty

```
bool empty() const;
```

Returns: `true` if the container is empty; `false`, otherwise.

The result may differ with the actual container state in case of pending concurrent insertions.

size

```
size_type size() const;
```

Returns: the number of elements in the container.

The result may differ with the actual container size in case of pending concurrent insertions.

max_size

```
size_type max_size() const;
```

Returns: the maximum number of elements that container can hold.

Concurrently safe modifiers

All member functions in this section can be performed concurrently with each other, lookup methods and while traversing the container.

Inserting values

```
std::pair<iterator, bool> insert( const value_type& value );
```

Attempts to insert the value `value` into the container.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element or to an existing element with equal key. Boolean value is `true` if insertion took place; `false`, otherwise.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, const value_type& other );
```

Attempts to insert the value `value` into the container.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

Returns: an `iterator` to the inserted element or to an existing element with equal key.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
template <typename P>
std::pair<iterator, bool> insert( P&& value );
```

Equivalent to `emplace(std::forward<P>(value))`.

This overload only participates in overload resolution if `std::is_constructible<value_type, P&&>::value` is `true`.

```
template <typename P>
iterator insert( const_iterator hint, P&& value );
```

Equivalent to `emplace_hint(hint, std::forward<P>(value))`.

This overload only participates in overload resolution if `std::is_constructible<value_type, P&&>::value` is `true`.

```
std::pair<iterator, bool> insert( value_type&& value );
```

Attempts to insert the value `value` into the container using move semantics.

`value` is left in a valid, but unspecified state.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element or to an existing element with equal key. Boolean value is `true` if insertion took place; `false`, otherwise.

Requirements: the type `value_type` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, value_type&& other );
```

Attempts to insert the value `value` into the container using move semantics.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

`value` is left in a valid, but unspecified state.

Returns: an `iterator` to the inserted element or to an existing element with equal key.

Requirements: the type `value_type` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

Inserting sequences of elements

```
template <typename InputIterator>
void insert( InputIterator first, InputIterator last );
```

Attempts to insert all items from the half-open interval `[first, last)` into the container.

If the interval `[first, last)` contains multiple elements with equal keys, it is unspecified which element should be inserted.

Requirements: the type `InputIterator` must meet the requirements of *InputIterator* from the `[input.iterators]` ISO C++ Standard section.

```
void insert( std::initializer_list<value_type> init );
```

Equivalent to `insert(init.begin(), init.end())`.

Inserting nodes

```
std::pair<iterator, bool> insert( node_type&& nh );
```

If the node handle `nh` is empty, does nothing.

Otherwise, attempts to insert the node owned by `nh` into the container.

If the insertion fails, node handle `nh` keeps ownership of the node.

Otherwise, `nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element or to an existing element with key equivalent to `nh.key()`. Boolean value is `true` if insertion took place; `false`, otherwise.

```
iterator insert( const_iterator hint, node_type&& nh );
```

If the node handle `nh` is empty, does nothing.

Otherwise, attempts to insert the node owned by `nh` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

If the insertion fails, node handle `nh` keeps ownership of the node.

Otherwise - `nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: an iterator pointing to the inserted element or to an existing element with key equivalent to `nh.key()`.

Emplacing elements

```
template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args );
```

Attempts to insert an element ,constructed in-place from `args` into the container.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element or to an existing element with equal key. Boolean value is `true` if insertion took place; `false`, otherwise.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ section.

```
template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args );
```

Attempts to insert an element constructed in-place from `args` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

Returns: an `iterator` to the inserted element or to an existing element with equal key.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ section.

Merging containers

```
template <typename SrcCompare>
void merge( concurrent_map<Key, T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_map<Key, T, SrcCompare, Allocator>&& source );

template <typename SrcCompare>
void merge( concurrent_multimap<Key, T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_multimap<Key, T, SrcCompare, Allocator>&& source );
```

Transfers those elements from `source` which keys do not exist in the container.

In case of merging with the container with multiple elements with equal keys, it is unspecified which element would be transferred.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `get_allocator() != source.get_allocator()`.

Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

Clearing

```
void clear();
```

Removes all elements from the container.

Erasing elements

```
iterator unsafe_erase( const_iterator pos );

iterator unsafe_erase( iterator pos );
```

Removes the element pointed to by `pos` from the container.

Invalidates all iterators and references to the removed element.

Returns: `iterator` which follows the removed element.

Requirements: the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
size_type unsafe_erase( const key_type& key );
```

Removes the element with the key equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed element.

Returns: 1 if an element with the key equivalent to `key` exists; `0`, otherwise.

```
template <typename K>
size_type unsafe_erase( const K& key );
```

Removes the element with the key that is equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed element.

This overload only participates in overload resolution if all of the following statements are `true`:

- The qualified-id `key_compare::is_transparent` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: 1 if an element with the key equivalent to `key` exists; `0`, otherwise.

Erasing sequences

```
iterator unsafe_erase( const_iterator first, const_iterator last );
```

Removes all elements from the half-open interval `[first, last)` from the container.

Returns: `iterator` that follows the last removed element.

Requirements: the range `[first, last)` must be a valid subrange in `*this`.

Extracting nodes

```
node_type unsafe_extract( iterator pos );

node_type unsafe_extract( const_iterator pos );
```

Transfers ownership of the element pointed to by `pos` from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element.

Requirements: the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
node_type unsafe_extract( const key_type& key );
```

If an element with the key equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element or an empty node handle if an element with the key equivalent to `key` was not found.

```
template <typename K>
node_type unsafe_extract( const K& key );
```

If an element with the key equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

This overload only participates in overload resolution if all of the following statements are `true`:

- The qualified-id `key_compare::is_transparent` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: the node handle that owns the extracted element or an empty node handle if an element with the key equivalent to `key` was not found.

swap

```
void swap( concurrent_map& other );
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`.

Otherwise, if `get_allocator() != other.get_allocator()`, the behavior is undefined.

Lookup

All methods in this section can be executed concurrently with each other, concurrently-safe modifiers, and while traversing the container.

count

```
size_type count( const key_type& key );
```

Returns: the number of elements with the key equivalent to `key`.

```
template <typename K>
size_type count( const K& key );
```

Returns: the number of elements with the key that is equivalent to `key`.

This overload only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

find

```
iterator find( const key_type& key );

const_iterator find( const key_type& key ) const;
```

Returns: an iterator to the element with the key equivalent to `key`, or `end()` if no such element exists.

```
template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;
```

Returns: an iterator to the element with the key equivalent to `key`, or `end()` if no such element exists.

These overloads only participate in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

contains

```
bool contains( const key_type& key ) const;
```

Returns: `true` if an element with the key equivalent to `key` exists in the container; `false`, otherwise.

```
template <typename K>
bool contains( const K& key ) const;
```

Returns: `true` if an element with the key that is equivalent to `key` exists in the container; `false`, otherwise.

This overload only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

lower_bound

```
iterator lower_bound( const key_type& key );

const_iterator lower_bound( const key_type& key ) const;
```

Returns: an iterator to the first element in the container with the key that is *not less* than `key`.

```
template <typename K>
iterator lower_bound( const K& key )

template <typename K>
const_iterator lower_bound( const K& key ) const
```

Returns: an iterator to the first element in the container with the key that is *not less* than `key`.

These overloads only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

upper_bound

```
iterator upper_bound( const key_type& key );

const_iterator upper_bound( const key_type& key ) const;
```

Returns: an iterator to the first element in the container with the key that compares *greater* than `key`.

```
template <typename K>
iterator upper_bound( const K& key );

template <typename K>
const_iterator upper_bound( const K& key ) const;
```

Returns: an iterator to the first element in the container with the key that compares `greater` than `key`.

These overloads only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

equal_range

```
std::pair<iterator, iterator> equal_range( const key_type& key );

std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;
```

Returns: if an element with the key equivalent to `key` exists, a pair of iterators `{f, l}`, where `f` is an iterator to this element, `l` is `std::next(f)`. Otherwise, `{end(), end()}`.

```
template <typename K>
std::pair<iterator, iterator> equal_range( const K& key )

template <typename K>
std::pair<const_iterator, const_iterator> equal_range( const K& key )
```

Returns: if an element with the key that is equivalent to `key` exists, a pair of iterators `{f, l}`, where `f` is an iterator to this element, `l` is `std::next(f)`. Otherwise, `{end(), end()}`.

These overloads only participate in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

Observers

get_allocator

```
allocator_type get_allocator() const;
```

Returns: a copy of the allocator associated with `*this`.

key_comp

```
key_compare key_comp() const;
```

Returns: a copy of the key comparison functor associated with `*this`.

value_comp

```
value_compare value_comp() const;
```

Returns: an object of the `value_compare` class that is used to compare `value_type` objects.

Parallel iteration

Member types `concurrent_map::range_type` and `concurrent_map::const_range_type` meet the *ContainerRange requirements*.

These types differ only in that the bounds for a `concurrent_map::const_range_type` are of type `concurrent_map::const_iterator`, whereas the bounds for a `concurrent_map::range_type` are of type `concurrent_map::iterator`.

range member function

```
range_type range();

const_range_type range() const;
```

Returns: a range object representing all elements in the container.

Non-member functions

These functions provide binary and lexicographical comparison and swap operations on `oneapi::tbb::concurrent_map` objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define `oneapi::tbb::concurrent_map` as a type alias for which the non-member functions are reachable only via argument-dependent lookup.

```
template <typename Key, typename T, typename Compare, typename Allocator>
void swap( concurrent_map<Key, T, Compare, Allocator>& lhs,
           concurrent_map<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator==( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                 const concurrent_map<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator!=( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                 const concurrent_map<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator<( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                const concurrent_map<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator>( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                const concurrent_map<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator<=( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                 const concurrent_map<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator>=( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                 const concurrent_map<Key, T, Compare, Allocator>& rhs );
```

Non-member swap

```
template <typename Key, typename T, typename Compare, typename Allocator>
void swap( concurrent_map<Key, T, Compare, Allocator>& lhs,
           concurrent_map<Key, T, Compare, Allocator>& rhs );
```

Equivalent to `lhs.swap(rhs)`.

Non-member binary comparisons

Two `oneapi::tbb::concurrent_map` objects are equal if they have the same number of elements and each element in one container is equal to the element in other container on the same position.

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator==( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                 const concurrent_map<Key, T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is equal to `rhs`; `false`, otherwise.

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator!=( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                 const concurrent_map<Key, T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is not equal to `rhs`; `false`, otherwise.

Non-member lexicographical comparisons

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator<( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                const concurrent_map<Key, T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *less* than `rhs`.

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator<=( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                 const concurrent_map<Key, T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *less or equal* than `rhs`.

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator>( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                const concurrent_map<Key, T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *greater* than `rhs`.

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator>=( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                 const concurrent_map<Key, T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *greater or equal* than `rhs`.

Other

Deduction guides

If possible, `concurrent_map` constructors support class template argument deduction (since C++17). Copy and move constructors, including constructors with an explicit `allocator_type` argument, provide implicitly-generated deduction guides. In addition, the following explicit deduction guides are provided:

```
template <typename InputIterator,
          typename Compare = std::less<iterator_key_t<InputIterator>>,
          typename Allocator = tbb_allocator<iterator_alloc_value_t<InputIterator>>>
concurrent_map( InputIterator, InputIterator,
                Compare = Compare(),
                Allocator = Allocator() )
-> concurrent_map<iterator_key_t<InputIterator>,
                  iterator_mapped_t<InputIterator>,
                  Compare,
                  Allocator>;

template <typename InputIterator,
          typename Allocator>
concurrent_map( InputIterator, InputIterator,
                Allocator )
-> concurrent_map<iterator_key_t<InputIterator>,
                  iterator_mapped_t<InputIterator>,
                  std::less<iterator_key_t<InputIterator>>,
                  Allocator>;

template <typename Key, typename T,
          typename Compare = std::less<std::remove_const_t<Key>>,
          typename Allocator = tbb_allocator<std::pair<const Key, T>>>
concurrent_map( std::initializer_list<std::pair<Key, T>>,
                Compare = Compare(),
                Allocator = Allocator() )
-> concurrent_map<std::remove_const_t<Key>,
                  T,
                  Compare,
                  Allocator>;

template <typename Key, typename T,
          typename Allocator>
concurrent_map( std::initializer_list<std::pair<Key, T>>, Allocator )
-> concurrent_map<std::remove_const_t<Key>,
                  T,
                  std::less<std::remove_const_t<Key>>,
                  Allocator>;
```

where the type aliases `iterator_key_t`, `iterator_mapped_t`, `iterator_alloc_value_t` are defined as follows:

```
template <typename InputIterator>
using iterator_key_t = std::remove_const_t<typename std::iterator_traits<InputIterator>
↪::value_type::first_type>;

template <typename InputIterator>
using iterator_mapped_t = typename std::iterator_traits<InputIterator>::value_
↪type::second_type;

template <typename InputIterator>
using iterator_alloc_value_t = std::pair<std::add_const_t<iterator_key_t<InputIterator>>,
                                         iterator_mapped_t<InputIterator>>;
```

These deduction guides only participate in the overload resolution if the following requirements are met:

- The `InputIterator` type meets the `InputIterator` requirements described in the [input.iterators] section of the ISO C++ Standard.
- The `Allocator` type meets the `Allocator` requirements described in the [allocator.requirements] section of the ISO C++ Standard.
- The `Compare` type does not meet the `Allocator` requirements.

Example

```
#include <oneapi/tbb/concurrent_map.h>
#include <vector>

int main() {
    std::vector<std::pair<int, float>> v;

    // Deduces cm1 as concurrent_map<int, float>
    oneapi::tbb::concurrent_map cm1(v.begin(), v.end());

    // Deduces cm2 as concurrent_map<int, float>
    oneapi::tbb::concurrent_map cm2({std::pair(1, 2f), std::pair(2, 3f)});
}
```

concurrent_multimap

[containers.concurrent_multimap]

`oneapi::tbb::concurrent_multimap` is a class template that represents a sorted associative container. It supports concurrent insertion, lookup, and traversal, but does not support concurrent erasure. In this container, multiple elements with equal keys can be stored.

Class Template Synopsis

```
namespace oneapi {
    namespace tbb {

        template <typename Key,
                  typename T,
                  typename Compare = std::less<Key>,
                  typename Allocator = tbb_allocator<std::pair<const Key, T>>
        class concurrent_multimap {
        public:
            using key_type = Key;
            using mapped_type = T;
            using value_type = std::pair<const Key, T>;

            using size_type = <implementation-defined unsigned integer type>;
            using difference_type = <implementation-defined signed integer type>;

            using key_compare = Compare;
            using allocator_type = Allocator;

            using reference = value_type&;
            using const_reference = const value_type&;
            using pointer = std::allocator_traits<Allocator>::pointer;
            using const_pointer = std::allocator_traits<Allocator>::const_pointer;

            using iterator = <implementation-defined ForwardIterator>;
            using const_iterator = <implementation-defined constant ForwardIterator>;

            using node_type = <implementation-defined node handle>;

            using range_type = <implementation-defined range>;
            using const_range_type = <implementation-defined constant node handle>;

            class value_compare;

            // Construction, destruction, copying
            concurrent_multimap();
            explicit concurrent_multimap( const key_compare& comp,
                                          const allocator_type& alloc = allocator_type()␣
↪);

            explicit concurrent_multimap( const allocator_type& alloc );

            template <typename InputIterator>
            concurrent_multimap( InputIterator first, InputIterator last,
                                 const key_compare& comp = key_compare(),
                                 const allocator_type& alloc = allocator_type() );

            template <typename InputIterator>
            concurrent_multimap( InputIterator first, InputIterator last,
                                 const allocator_type& alloc );
```

(continues on next page)

(continued from previous page)

```
            concurrent_multimap( std::initializer_list<value_type> init,
                                 const key_compare& comp = key_compare(),
                                 const allocator_type& alloc = allocator_type() );

            concurrent_multimap( std::initializer_list<value_type> init, const allocator_
↪type& alloc );

            concurrent_multimap( const concurrent_multimap& other );
            concurrent_multimap( const concurrent_multimap& other,
                                 const allocator_type& alloc );

            concurrent_multimap( concurrent_multimap&& other );
            concurrent_multimap( concurrent_multimap&& other,
                                 const allocator_type& alloc );

            ~concurrent_multimap();

            concurrent_multimap& operator=( const concurrent_multimap& other );
            concurrent_multimap& operator=( concurrent_multimap&& other );
            concurrent_multimap& operator=( std::initializer_list<value_type> init );

            allocator_type get_allocator() const;

            // Iterators
            iterator begin();
            const_iterator begin() const;
            const_iterator cbegin() const;

            iterator end();
            const_iterator end() const;
            const_iterator cend() const;

            // Size and capacity
            bool empty() const;
            size_type size() const;
            size_type max_size() const;

            // Concurrently safe modifiers
            std::pair<iterator, bool> insert( const value_type& value );

            iterator insert( const_iterator hint, const value_type& value );

            template <typename P>
            std::pair<iterator, bool> insert( P&& value );

            template <typename P>
            iterator insert( const_iterator hint, P&& value );

            std::pair<iterator, bool> insert( value_type&& value );

            iterator insert( const_iterator hint, value_type&& value );
```

(continues on next page)

(continued from previous page)

```
        template <typename InputIterator>
        void insert( InputIterator first, InputIterator last );

        void insert( std::initializer_list<value_type> init );

        std::pair<iterator, bool> insert( node_type&& nh );
        iterator insert( const_iterator hint, node_type&& nh );

        template <typename... Args>
        std::pair<iterator, bool> emplace( Args&&... args );

        template <typename... Args>
        iterator emplace_hint( const_iterator hint, Args&&... args );

        template <typename SrcCompare>
        void merge( concurrent_map<Key, T, SrcCompare, Allocator>& source );

        template <typename SrcCompare>
        void merge( concurrent_map<Key, T, SrcCompare, Allocator>&& source );

        template <typename SrcCompare>
        void merge( concurrent_multimap<Key, T, SrcCompare, Allocator>& source );

        template <typename SrcCompare>
        void merge( concurrent_multimap<Key, T, SrcCompare, Allocator>&& source );

        // Concurrently unsafe modifiers
        void clear();

        iterator unsafe_erase( const_iterator pos );
        iterator unsafe_erase( iterator pos );

        iterator unsafe_erase( const_iterator first, const_iterator last );

        size_type unsafe_erase( const key_type& key );

        template <typename K>
        size_type unsafe_erase( const K& key );

        node_type unsafe_extract( const_iterator pos );
        node_type unsafe_extract( iterator pos );

        node_type unsafe_extract( const key_type& key );

        template <typename K>
        node_type unsafe_extract( const K& key );

        void swap( concurrent_multimap& other );

        // Lookup
        size_type count( const key_type& key );
```

(continues on next page)

(continued from previous page)

```
            template <typename K>
            size_type count( const K& key );

            iterator find( const key_type& key );
            const_iterator find( const key_type& key ) const;

            template <typename K>
            iterator find( const K& key );

            template <typename K>
            const_iterator find( const K& key ) const;

            bool contains( const key_type& key ) const;

            template <typename K>
            bool contains( const K& key ) const;

            std::pair<iterator, iterator> equal_range( const key_type& key );
            std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;

            template <typename K>
            std::pair<iterator, iterator>  equal_range( const K& key );
            std::pair<const_iterator, const_iterator> equal_range( const K& key ) const;

            iterator lower_bound( const key_type& key );
            const_iterator lower_bound( const key_type& key ) const;

            template <typename K>
            iterator lower_bound( const K& key );

            template <typename K>
            const_iterator lower_bound( const K& key ) const;

            iterator upper_bound( const key_type& key );
            const_iterator upper_bound( const key_type& key ) const;

            template <typename K>
            iterator upper_bound( const K& key );

            template <typename K>
            const_iterator upper_bound( const K& key ) const;

            // Observers
            key_compare key_comp() const;

            value_compare value_comp() const;

            // Parallel iteration
            range_type range();
            const_range_type range() const;
        }; // class concurrent_multimap
```

(continues on next page)

(continued from previous page)

```
    } // namespace tbb
} // namespace oneapi
```

Requirements:

- The expression `std::allocator_traits<Allocator>::destroy(m, val)`, where `m` is an object of the type `Allocator` and `val` is an object of the type `value_type`, must be well-formed. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Compare` must meet the `Compare` requirements from the [alg.sorting] ISO C++ Standard section.
- The type `Allocator` must meet the `Allocator` requirements from the [allocator.requirements] ISO C++ Standard section.

Member classes

value_compare

`concurrent_multimap::value_compare` is a function object that is used to compare `concurrent_multimap::value_type` objects by comparing their first components.

Class Synopsis

```
namespace oneapi {
    namespace tbb {

        template <typename Key, typename T,
                  typename Compare, typename Allocator>
        class concurrent_multimap<Key, T, Compare, Allocator>::value_compare {
        protected:
            key_compare comp;

            value_compare( key_compare c );

        public:
            bool operator()( const value_type& lhs, const value_type& rhs )␣
↪const;
        }; // class value_compare

    } // namespace tbb
} // namespace oneapi
```

Member objects

```
key_compare comp;
```

The key comparison function object.

Member functions

```
value_compare( key_compare c );
```

Constructs a `value_compare` with the stored key comparison function object `c`.

```
bool operator()( const value_type& lhs, const value_type& rhs ) const;
```

Compares `lhs.first` and `rhs.first` by calling the stored key comparison function `comp`.

Returns: `true` if first components of `lhs` and `rhs` are equal; `false`, otherwise.

Member functions

Construction, destruction, copying

Empty container constructors

```
concurrent_multimap();

explicit concurrent_multimap( const key_compare& comp,
                              const allocator_type& alloc = allocator_type() );

explicit concurrent_multimap( const allocator_type& alloc );
```

Constructs an empty `concurrent_multimap`.

If provided, uses the comparison function object `comp` for all `key_type` comparisons and the allocator `alloc` to allocate the memory.

Constructors from the sequence of elements

```
template <typename InputIterator>
concurrent_multimap( InputIterator first, InputIterator last,
                     const key_compare& comp = key_compare(),
                     const allocator_type& alloc = allocator_type() );

template <typename InputIterator>
concurrent_multimap( InputIterator first, InputIterator last,
                const allocator_type& alloc = allocator_type() );
```

Constructs the `concurrent_multimap`, which contains all elements from the half-open interval `[first, last)`.

If provided, uses the comparison function object `comp` for all `key_type` comparisons and the allocator `alloc` to allocate the memory.

Requirements: the type `InputIterator` must meet the requirements of *InputIterator* from `[input.iterators]` ISO C++ Standard section.

```
concurrent_multimap( std::initializer_list<value_type> init, const key_compare&
↪ comp = key_compare(),
                     const allocator_type& alloc = allocator_type() );
```

Equivalent to `concurrent_multimap(init.begin(), init.end(), comp, alloc)`.

```
concurrent_multimap( std::initializer_list<value_type> init,
                     const allocator_type& alloc );
```

Equivalent to `concurrent_multimap(init.begin(), init.end(), alloc)`.

Copying constructors

```
concurrent_multimap( const concurrent_multimap& other );

concurrent_multimap( const concurrent_multimap& other, const allocator_type&␣
↪alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Moving constructors

```
concurrent_multimap( concurrent_multimap&& other );

concurrent_multimap( concurrent_multimap&& other, const allocator_type& alloc␣
↪);
```

Constructs a *concurrent_multimap* with the contents of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by calling `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Destructor

```
~concurrent_multimap();
```

Destroys the `concurrent_multimap`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

Assignment operators

```
concurrent_multimap& operator=( const concurrent_multimap& other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_assignment::` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_multimap& operator=( concurrent_multimap&& other );
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment::` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_multimap& operator=( std::initializer_list<value_type> init );
```

Replaces all elements in `*this` by the elements in `init`.

If `init` contains multiple elements with equal keys, it is unspecified which element would be inserted.

The behavior is undefined in case of concurrent operations with `*this`.

Returns: a reference to `*this`.

Iterators

The types `concurrent_multimap::iterator` and `concurrent_multimap::const_iterator` meet the requirements of `ForwardIterator` from the [forward.iterators] ISO C++ standard section.

begin and cbegin

```
iterator begin();

const_iterator begin() const;

const_iterator cbegin() const;
```

Returns: an iterator to the first element in the container.

end and cend

```
iterator end();

const_iterator end() const;

const_iterator cend() const;
```

Returns: an iterator to the element that follows the last element in the container.

Size and capacity

empty

```
bool empty() const;
```

Returns: `true` if the container is empty, `false` otherwise.

The result may differ with the actual container state in case of pending concurrent insertions.

size

```
size_type size() const;
```

Returns: the number of elements in the container.

The result may differ with the actual container size in case of pending concurrent insertions.

max_size

```
size_type max_size() const;
```

Returns: the maximum number of elements that container can hold.

Concurrently safe modifiers

All member functions in this section can be performed concurrently with each other, lookup methods and while traversing the container.

Emplacing elements

```
template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args );
```

Inserts an element constructed in-place from `args` into the container.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element. Boolean value is always `true`.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from [container.requirements] ISO C++ section.

```
template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args );
```

Inserts an element constructed in-place from `args` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

Returns: an `iterator` to the inserted element.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ section.

Inserting values

```
std::pair<iterator, bool> insert( const value_type& value );
```

Inserts the value `value` into the container.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element. Boolean value is always `true`.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, const value_type& other );
```

Inserts the value `value` into the container.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

Returns: an `iterator` to the inserted element.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
template <typename P>
std::pair<iterator, bool> insert( P&& value );
```

Equivalent to `emplace(std::forward<P>(value))`.

This overload only participates in overload resolution if `std::is_constructible<value_type, P&&>::value` is `true`.

```
template <typename P>
iterator insert( const_iterator hint, P&& value );
```

Equivalent to `emplace_hint(hint, std::forward<P>(value))`.

This overload only participates in overload resolution if `std::is_constructible<value_type, P&&>::value` is `true`.

```
std::pair<iterator, bool> insert( value_type&& value );
```

Inserts the value `value` into the container using move semantics.

`value` is left in a valid, but unspecified state.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element. Boolean value is always `true`.

Requirements: the type `value_type` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, value_type&& other );
```

Inserts the value `value` into the container using move semantics.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

`value` is left in a valid, but unspecified state.

Returns: an `iterator` to the inserted element.

Requirements: the type `value_type` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

Inserting sequences of elements

```
template <typename InputIterator>
void insert( InputIterator first, InputIterator last );
```

Inserts all items from the half-open interval `[first, last)` into the container.

Requirements: the type `InputIterator` must meet the requirements of *InputIterator* from `[input.iterators]` the ISO C++ Standard section.

```
void insert( std::initializer_list<value_type> init );
```

Equivalent to `insert(init.begin(), init.end())`.

Inserting nodes

```
std::pair<iterator, bool> insert( node_type&& nh );
```

If the node handle `nh` is empty, does nothing.

Otherwise, inserts the node owned by `nh` into the container.

`nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element. Boolean value is always `true`.

```
iterator insert( const_iterator hint, node_type&& nh );
```

If the node handle `nh` is empty, does nothing.

Otherwise, inserts the node owned by `nh` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

`nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: an iterator pointing to the inserted element.

Merging containers

```
template <typename SrcCompare>
void merge( concurrent_map<Key, T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_map<Key, T, SrcCompare, Allocator>&& source );

template <typename SrcCompare>
void merge( concurrent_multimap<Key, T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_multimap<Key, T, SrcCompare, Allocator>&& source );
```

Transfers all elements from `source` to `*this`.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `get_allocator() != source.get_allocator()`.

Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

Clearing

```
void clear();
```

Removes all elements from the container.

Erasing elements

```
iterator unsafe_erase( const_iterator pos );

iterator unsafe_erase( iterator pos );
```

Removes the element pointed to by `pos` from the container.

Invalidates all iterators and references to the removed element.

Returns: `iterator` which follows the removed element.

Requirements: the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
size_type unsafe_erase( const key_type& key );
```

Removes all element with the key equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed elements.

Returns: the number of removed elements.

```
template <typename K>
size_type unsafe_erase( const K& key );
```

Removes all elements with the key that is equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed elements.

This overload only participates in overload resolution if all of the following statements are `true`:

- The qualified-id `key_compare::is_transparent` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: the number of removed elements.

Erasing sequences

```
iterator unsafe_erase( const_iterator first, const_iterator last );
```

Removes all elements from the half-open interval `[first, last)` from the container.

Returns: `iterator` that follows the last removed element.

Requirements: the range `[first, last)` must be a valid subrange in `*this`.

Extracting nodes

```
node_type unsafe_extract( iterator pos );

node_type unsafe_extract( const_iterator pos );
```

Transfers ownership of the element pointed to by `pos` from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element.

Requirements: the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
node_type unsafe_extract( const key_type& key );
```

If at least one element with the key equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

If there are multiple elements with the key equivalent to `key`, it is unspecified which element should be transferred.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element or an empty node handle if an element with the key equivalent to `key` was not found.

```
template <typename K>
node_type unsafe_extract( const K& key );
```

If at least one element with the key that is equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

If there are multiple elements with the key that is equivalent to `key`, it is unspecified which element should be transferred.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

This overload only participates in overload resolution if all of the following statements are `true`:

- The qualified-id `key_compare::is_transparent` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: the node handle that owns the extracted element or an empty node handle if an element with the key that is equivalent to `key` was not found.

swap

```
void swap( concurrent_multimap& other );
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`.

Otherwise, if `get_allocator() != other.get_allocator()`, the behavior is undefined.

Lookup

All methods in this section can be executed concurrently with each other, concurrently-safe modifiers and while traversing the container.

count

```
size_type count( const key_type& key );
```

Returns: the number of elements with the key equivalent to `key`.

```
template <typename K>
size_type count( const K& key );
```

Returns: the number of elements with the key equivalent to `key`.

This overload only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

find

```
iterator find( const key_type& key );

const_iterator find( const key_type& key ) const;
```

Returns: an iterator to the element with the key equivalent to `key`, or `end()` if no such element exists.

If there are multiple elements with the key equivalent to `key`, it is unspecified which element should be found.

```
template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;
```

Returns: an iterator to the element with the key that is equivalent to `key`, or `end()` if no such element exists.

If there are multiple elements with the key that is equivalent to `key`, it is unspecified which element should be found.

These overloads only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

contains

```
bool contains( const key_type& key ) const;
```

Returns: `true` if an element with the key equivalent to `key` exists in the container; `false`, otherwise.

```
template <typename K>
bool contains( const K& key ) const;
```

Returns: `true` if an element with the key equivalent to `key` exists in the container; `false`, otherwise.

This overload only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

lower_bound

```
iterator lower_bound( const key_type& key );

const_iterator lower_bound( const key_type& key ) const;
```

Returns: an iterator to the first element in the container with the key that is *not less* than `key`.

```
template <typename K>
iterator lower_bound( const K& key )

template <typename K>
const_iterator lower_bound( const K& key ) const
```

Returns: an iterator to the first element in the container with the key that is *not less* than `key`.

These overloads only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

upper_bound

```
iterator upper_bound( const key_type& key );

const_iterator upper_bound( const key_type& key ) const;
```

Returns: an iterator to the first element in the container with the key that compares *greater* than `key`.

```
template <typename K>
iterator upper_bound( const K& key );

template <typename K>
const_iterator upper_bound( const K& key ) const;
```

Returns: an iterator to the first element in the container with the key that compares `greater` than `key`.

These overloads only participate in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

equal_range

```
std::pair<iterator, iterator> equal_range( const key_type& key );

std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;
```

Returns: if at least one element with the key equivalent to `key` exists, a pair of iterators `{f, l}`, where `f` is an iterator to the first element with the key equivalent to `key`, `l` is an iterator to the element that follows the last element with the key equivalent to `key`. Otherwise - `{end(), end()}`.

```
template <typename K>
std::pair<iterator, iterator> equal_range( const K& key )

template <typename K>
std::pair<const_iterator, const_iterator> equal_range( const K& key )
```

Returns: if at least one element with the key equivalent to `key` exists, a pair of iterators `{f, l}`, where `f` is an iterator to the first element with the key that is equivalent to `key`, `l` is an iterator to the element that follows the last element with the key that is equivalent to `key`. Otherwise, `{end(), end()}`.

These overloads only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

Observers

get_allocator

```
allocator_type get_allocator() const;
```

Returns: a copy of the allocator associated with `*this`.

key_comp

```
key_compare key_comp() const;
```

Returns: a copy of the key comparison functor associated with `*this`.

value_comp

```
value_compare value_comp() const;
```

Returns: an object of the `value_compare` class that is used to compare `value_type` objects.

Parallel iteration

Member types `concurrent_multimap::range_type` and `concurrent_multimap::const_range_type` meet the *ContainerRange requirements*.

These types differ only in that the bounds for a `concurrent_multimap::const_range_type` are of type `concurrent_multimap::const_iterator`, whereas the bounds for a `concurrent_multimap::range_type` are of type `concurrent_multimap::iterator`.

range member function

```
range_type range();

const_range_type range() const;
```

Returns: a range object representing all elements in the container.

Non-member functions

These functions provide binary and lexicographical comparison and swap operations on `oneapi::tbb::concurrent_multimap` objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define `oneapi::tbb::concurrent_multimap` as a type alias for which the non-member functions are reachable only via argument-dependent lookup.

```
template <typename Key, typename T, typename Compare, typename Allocator>
void swap( concurrent_multimap<Key, T, Compare, Allocator>& lhs,
           concurrent_multimap<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator==( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                 const concurrent_multimap<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator!=( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                 const concurrent_multimap<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator<( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                const concurrent_multimap<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator>( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                const concurrent_multimap<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator<=( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                 const concurrent_multimap<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator>=( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                 const concurrent_mutlimap<Key, T, Compare, Allocator>& rhs );
```

Non-member swap

```
template <typename Key, typename T, typename Compare, typename Allocator>
void swap( concurrent_multimap<Key, T, Compare, Allocator>& lhs,
           concurrent_multimap<Key, T, Compare, Allocator>& rhs );
```

Equivalent to `lhs.swap(rhs)`.

Non-member binary comparisons

Two `oneapi::tbb::concurrent_multimap` objects are equal if they have the same number of elements and each element in one container is equal to the element in other container on the same position.

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator==( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                 const concurrent_multimap<Key, T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is equal to `rhs`; `false`, otherwise.

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator!=( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                 const concurrent_multimap<Key, T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is not equal to `rhs`; `false`, otherwise.

Non-member lexicographical comparisons

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator<( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                const concurrent_multimap<Key, T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *less* than `rhs`.

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator<=( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                 const concurrent_multimap<Key, T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *less or equal* than `rhs`.

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator>( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                const concurrent_multimap<Key, T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *greater* than `rhs`.

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator>=( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                 const concurrent_multimap<Key, T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *greater or equal* than `rhs`.

Other

Deduction guides

If possible, `concurrent_multimap` constructors support class template argument deduction (since C++17). Copy and move constructors, including constructors with an explicit `allocator_type` argument, provide implicitly-generated deduction guides. In addition, the following explicit deduction guides are provided:

```
template <typename InputIterator,
          typename Compare = std::less<iterator_key_t<InputIterator>>,
          typename Allocator = tbb_allocator<iterator_alloc_value_t<InputIterator>>>
concurrent_multimap( InputIterator, InputIterator,
                     Compare = Compare(),
                     Allocator = Allocator() )
```

(continues on next page)

(continued from previous page)

```
-> concurrent_multimap<iterator_key_t<InputIterator>,
                       iterator_mapped_t<InputIterator>,
                       Compare,
                       Allocator>;

template <typename InputIterator,
          typename Allocator>
concurrent_multimap( InputIterator, InputIterator,
                     Allocator )
-> concurrent_multimap<iterator_key_t<InputIterator>,
                       iterator_mapped_t<InputIterator>,
                       std::less<iterator_key_t<InputIterator>>,
                       Allocator>;

template <typename Key, typename T,
          typename Compare = std::less<std::remove_const_t<Key>>,
          typename Allocator = tbb_allocator<std::pair<const Key, T>>>
concurrent_multimap( std::initializer_list<std::pair<Key, T>>,
                     Compare = Compare(),
                     Allocator = Allocator() )
-> concurrent_multimap<std::remove_const_t<Key>,
                       T,
                       Compare,
                       Allocator>;

template <typename Key, typename T,
          typename Allocator>
concurrent_multimap( std::initializer_list<std::pair<Key, T>>, Allocator )
-> concurrent_multimap<std::remove_const_t<Key>,
                       T,
                       std::less<std::remove_const_t<Key>>,
                       Allocator>;
```

where the type aliases `iterator_key_t`, `iterator_mapped_t`, `iterator_alloc_value_t` are defined as follows:

```
template <typename InputIterator>
using iterator_key_t = std::remove_const_t<typename std::iterator_traits<InputIterator>
↪::value_type::first_type>;

template <typename InputIterator>
using iterator_mapped_t = typename std::iterator_traits<InputIterator>::value_
↪type::second_type;

template <typename InputIterator>
using iterator_alloc_value_t = std::pair<std::add_const_t<iterator_key_t<InputIterator>>,
                                         iterator_mapped_t<InputIterator>>;
```

These deduction guides only participate in the overload resolution if the following requirements are met:

- The `InputIterator` type meets the `InputIterator` requirements described in the [input.iterators] section of the ISO C++ Standard.
- The `Allocator` type meets the `Allocator` requirements described in the [allocator.requirements] section of the ISO C++ Standard.

- The `Compare` type does not meet the `Allocator` requirements.

Example

```
#include <oneapi/tbb/concurrent_map.h>
#include <vector>

int main() {
    std::vector<std::pair<int, float>> v;

    // Deduces cm1 as concurrent_multimap<int, float>
    oneapi::tbb::concurrent_multimap cm1(v.begin(), v.end());

    // Deduces cm2 as concurrent_multimap<int, float>
    oneapi::tbb::concurrent_multimap cm2({std::pair(1, 2f), std::pair(2, 3f)});
}
```

concurrent_set

[containers.concurrent_set]

`oneapi::tbb::concurrent_set` is a class template that represents a sorted sequence of unique elements. It supports concurrent insertion, lookup and traversal, but does not support concurrent erasure.

Class Template Synopsis

```
// Defined in header <oneapi/tbb/concurrent_set.h>

namespace oneapi {
    namespace tbb {

        template <typename T,
                  typename Compare = std::less<T>,
                  typename Allocator = tbb_allocator<T>>
        class concurrent_set {
        public:
            using key_type = T;
            using value_type = T;

            using size_type = <implementation-defined unsigned integer type>;
            using difference_type = <implementation-defined signed integer type>;

            using key_compare = Compare;
            using value_compare = Compare;

            using allocator_type = Allocator;

            using reference = value_type&;
            using const_reference = const value_type&;
            using pointer = std::allocator_traits<Allocator>::pointer;
            using const_pointer = std::allocator_traits<Allocator>::const_pointer;
```

(continues on next page)

(continued from previous page)

```
            using iterator = <implementation-defined ForwardIterator>;
            using const_iterator = <implementation-defined constant ForwardIterator>;

            using node_type = <implementation-defined node handle>;

            using range_type = <implementation-defined range>;
            using const_range_type = <implementation-defined constant node handle>;

            // Construction, destruction, copying
            concurrent_set();
            explicit concurrent_set( const key_compare& comp,
                                     const allocator_type& alloc = allocator_type() );

            explicit concurrent_set( const allocator_type& alloc );

            template <typename InputIterator>
            concurrent_set( InputIterator first, InputIterator last,
                            const key_compare& comp = key_compare(),
                            const allocator_type& alloc = allocator_type() );

            template <typename InputIterator>
            concurrent_set( InputIterator first, InputIterator last,
                            const allocator_type& alloc );

            concurrent_set( std::initializer_list<value_type> init,
                            const key_compare& comp = key_compare(),
                            const allocator_type& alloc = allocator_type() );

            concurrent_set( std::initializer_list<value_type> init, const allocator_type&
↪ alloc );

            concurrent_set( const concurrent_set& other );
            concurrent_set( const concurrent_set& other,
                            const allocator_type& alloc );

            concurrent_set( concurrent_set&& other );
            concurrent_set( concurrent_set&& other,
                            const allocator_type& alloc );

            ~concurrent_set();

            concurrent_set& operator=( const concurrent_set& other );
            concurrent_set& operator=( concurrent_set&& other );
            concurrent_set& operator=( std::initializer_list<value_type> init );

            allocator_type get_allocator() const;

            // Iterators
            iterator begin();
            const_iterator begin() const;
            const_iterator cbegin() const;
```

(continues on next page)

(continued from previous page)

```
        iterator end();
        const_iterator end() const;
        const_iterator cend() const;

        // Size and capacity
        bool empty() const;
        size_type size() const;
        size_type max_size() const;

        // Concurrently safe modifiers
        std::pair<iterator, bool> insert( const value_type& value );

        iterator insert( const_iterator hint, const value_type& value );

        std::pair<iterator, bool> insert( value_type&& value );

        iterator insert( const_iterator hint, value_type&& value );

        template <typename InputIterator>
        void insert( InputIterator first, InputIterator last );

        void insert( std::initializer_list<value_type> init );

        std::pair<iterator, bool> insert( node_type&& nh );
        iterator insert( const_iterator hint, node_type&& nh );

        template <typename... Args>
        std::pair<iterator, bool> emplace( Args&&... args );

        template <typename... Args>
        iterator emplace_hint( const_iterator hint, Args&&... args );

        template <typename SrcCompare>
        void merge( concurrent_set<T, SrcCompare, Allocator>& source );

        template <typename SrcCompare>
        void merge( concurrent_set<T, SrcCompare, Allocator>&& source );

        template <typename SrcCompare>
        void merge( concurrent_multiset<T, SrcCompare, Allocator>& source );

        template <typename SrcCompare>
        void merge( concurrent_multiset<T, SrcCompare, Allocator>&& source );

        // Concurrently unsafe modifiers
        void clear();

        iterator unsafe_erase( const_iterator pos );
        iterator unsafe_erase( iterator pos );

        iterator unsafe_erase( const_iterator first, const_iterator last );
```

(continues on next page)

(continued from previous page)

```
            size_type unsafe_erase( const key_type& key );

            template <typename K>
            size_type unsafe_erase( const K& key );

            node_type unsafe_extract( const_iterator pos );
            node_type unsafe_extract( iterator pos );

            node_type unsafe_extract( const key_type& key );

            template <typename K>
            node_type unsafe_extract( const K& key );

            void swap( concurrent_set& other );

            // Lookup
            size_type count( const key_type& key );

            template <typename K>
            size_type count( const K& key );

            iterator find( const key_type& key );
            const_iterator find( const key_type& key ) const;

            template <typename K>
            iterator find( const K& key );

            template <typename K>
            const_iterator find( const K& key ) const;

            bool contains( const key_type& key ) const;

            template <typename K>
            bool contains( const K& key ) const;

            std::pair<iterator, iterator> equal_range( const key_type& key );
            std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;

            template <typename K>
            std::pair<iterator, iterator>  equal_range( const K& key );
            std::pair<const_iterator, const_iterator> equal_range( const K& key ) const;

            iterator lower_bound( const key_type& key );
            const_iterator lower_bound( const key_type& key ) const;

            template <typename K>
            iterator lower_bound( const K& key );

            template <typename K>
            const_iterator lower_bound( const K& key ) const;
```

(continues on next page)

(continued from previous page)

```
            iterator upper_bound( const key_type& key );
            const_iterator upper_bound( const key_type& key ) const;

            template <typename K>
            iterator upper_bound( const K& key );

            template <typename K>
            const_iterator upper_bound( const K& key ) const;

            // Observers
            key_compare key_comp() const;

            value_compare value_comp() const;

            // Parallel iteration
            range_type range();
            const_range_type range() const;
        }; // class concurrent_set

    } // namespace tbb
} // namespace oneapi
```

Requirements:

- The expression `std::allocator_traits<Allocator>::destroy(m, val)`, where `m` is an object of the type `Allocator` and `val` is an object of the type `value_type`, must be well-formed. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Compare` must meet the `Compare` requirements from the [alg.sorting] ISO C++ Standard section.
- The type `Allocator` must meet the `Allocator` requirements from the [allocator.requirements] ISO C++ Standard section.

Member functions

Construction, destruction, copying

Empty container constructors

```
concurrent_set();

explicit concurrent_set( const key_compare& comp,
                         const allocator_type& alloc = allocator_type() );

explicit concurrent_set( const allocator_type& alloc );
```

Constructs an empty `concurrent_set`.

If provided, uses the comparison function object `comp` for all `key_type` comparisons and the allocator `alloc` to allocate the memory.

Constructors from the sequence of elements

```
template <typename InputIterator>
concurrent_set( InputIterator first, InputIterator last,
                const key_compare& comp = key_compare(),
                const allocator_type& alloc = allocator_type() );

template <typename InputIterator>
concurrent_set( InputIterator first, InputIterator last,
                const allocator_type& alloc = allocator_type() );
```

Constructs the `concurrent_set` that contains the elements from the half-open interval `[first, last)`.

If the range `[first, last)` contains multiple equal elements, it is unspecified which element would be inserted.

If provided, uses the comparison function object `comp` for all `key_type` comparisons and the allocator `alloc` to allocate the memory.

Requirements: the type `InputIterator` must meet the requirements of *InputIterator* from the `[input.iterators]` ISO C++ Standard section.

```
concurrent_set( std::initializer_list<value_type> init, const key_compare&␣
↪comp = key_compare(),
                const allocator_type& alloc = allocator_type() );
```

Equivalent to `concurrent_set(init.begin(), init.end(), comp, alloc)`.

```
concurrent_set( std::initializer_list<value_type> init,
                const allocator_type& alloc );
```

Equivalent to `concurrent_set(init.begin(), init.end(), alloc)`.

Copying constructors

```
concurrent_set( const concurrent_set& other );

concurrent_set( const concurrent_set& other, const allocator_type& alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Moving constructors

```
concurrent_set( concurrent_set&& other );

concurrent_set( concurrent_set&& other, const allocator_type& alloc );
```

Constructs a *concurrent_set* with the contents of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by calling `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Destructor

```
~concurrent_set();
```

Destroys the `concurrent_set`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

Assignment operators

```
concurrent_set& operator=( const concurrent_set& other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_assignment::` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_set& operator=( concurrent_set&& other );
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment:` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_set& operator=( std::initializer_list<value_type> init );
```

Replaces all elements in `*this` by the elements in `init`.

If `init` contains multiple elements with equal keys, it is unspecified which element would be inserted.

The behavior is undefined in case of concurrent operations with `*this`.

Returns: a reference to `*this`.

Iterators

The types `concurrent_set::iterator` and `concurrent_set::const_iterator` meet the requirements of `ForwardIterator` from the [forward.iterators] ISO C++ standard section.

begin and cbegin

```
iterator begin();

const_iterator begin() const;

const_iterator cbegin() const;
```

Returns: an iterator to the first element in the container.

end and cend

```
iterator end();

const_iterator end() const;

const_iterator cend() const;
```

Returns: an iterator to the element that follows the last element in the container.

Size and capacity

empty

```
bool empty() const;
```

Returns: `true` if the container is empty; `false`, otherwise.

The result may differ from the actual container state in case of pending concurrent insertions.

size

```
size_type size() const;
```

Returns: the number of elements in the container.

The result may differ from the actual container size in case of pending concurrent insertions.

max_size

```
size_type max_size() const;
```

Returns: the maximum number of elements that container can hold.

Concurrently safe modifiers

All member functions in this section can be performed concurrently with each other, lookup methods and while traversing the container.

Inserting values

```
std::pair<iterator, bool> insert( const value_type& value );
```

Attempts to insert the value `value` into the container.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element or to an existing element with equal key. Boolean value is `true` if insertion took place; `false`, otherwise.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, const value_type& other );
```

Attempts to insert the value `value` into the container.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

Returns: an `iterator` to the inserted element or to an existing element with equal key.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
std::pair<iterator, bool> insert( value_type&& value );
```

Attempts to insert the value `value` into the container using move semantics.

`value` is left in a valid, but unspecified state.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element or to an existing element with equal key. Boolean value is `true` if insertion took place; `false`, otherwise.

Requirements: the type `value_type` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, value_type&& other );
```

Attempts to insert the value `value` into the container using move semantics.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

`value` is left in a valid, but unspecified state.

Returns: an `iterator` to the inserted element or to an existing element with equal key.

Requirements: the type `value_type` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

Inserting sequences of elements

```
template <typename InputIterator>
void insert( InputIterator first, InputIterator last );
```

Attempts to insert all items from the half-open interval `[first, last)` into the container.

If the interval `[first, last)` contains multiple equal elements, it is unspecified which element should be inserted.

Requirements: the type `InputIterator` must meet the requirements of *InputIterator* from the `[input.iterators]` ISO C++ Standard section.

```
void insert( std::initializer_list<value_type> init );
```

Equivalent to `insert(init.begin(), init.end())`.

Inserting nodes

```
std::pair<iterator, bool> insert( node_type&& nh );
```

If the node handle `nh` is empty, does nothing.

Otherwise, attempts to insert the node owned by `nh` into the container.

If the insertion fails, node handle `nh` keeps ownership of the node.

Otherwise, `nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element or to an existing element equal to `nh.value()`. Boolean value is `true` if insertion took place; `false`, otherwise.

```
iterator insert( const_iterator hint, node_type&& nh );
```

If the node handle `nh` is empty, does nothing.

Otherwise, attempts to insert the node owned by `nh` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

If the insertion fails, node handle `nh` remains ownership of the node.

Otherwise, `nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: an iterator pointing to the inserted element or to an existing element equal to `nh.value()`.

Emplacing elements

```
template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args );
```

Attempts to insert an element constructed in-place from `args` into the container.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element or to an existing element with equal key. Boolean value is `true` if insertion took place; `false`, otherwise.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ section.

```
template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args );
```

Attempts to insert an element constructed in-place from `args` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

Returns: an `iterator` to the inserted element or to an existing element with equal key.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ section.

Merging containers

```
template <typename SrcCompare>
void merge( concurrent_set<T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_set<T, SrcCompare, Allocator>&& source );

template <typename SrcCompare>
void merge( concurrent_multiset<T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_multiset<T, SrcCompare, Allocator>&& source );
```

Transfers those elements from `source` which keys do not exist in the container.

In case of merging with the container with multiple equal elements, it is unspecified which element would be transferred.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `get_allocator() != source.get_allocator()`.

Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

Clearing

```
void clear();
```

Removes all elements from the container.

Erasing elements

```
iterator unsafe_erase( const_iterator pos );

iterator unsafe_erase( iterator pos );
```

Removes the element pointed to by `pos` from the container.

Invalidates all iterators and references to the removed element.

Returns: `iterator` that follows the removed element.

Requirements: the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
size_type unsafe_erase( const key_type& key );
```

Removes the element equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed element.

Returns: 1 if an element equivalent to `key` exists; `0`, otherwise.

```
template <typename K>
size_type unsafe_erase( const K& key );
```

Removes the element that is equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed element.

This overload only participates in overload resolution if all of the following statements are `true`:

- The qualified-id `key_compare::is_transparent` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.

- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: 1 if an element equivalent to `key` exists; `0`, otherwise.

Erasing sequences

```
iterator unsafe_erase( const_iterator first, const_iterator last );
```

Removes all elements from the half-open interval `[first, last)` from the container.

Returns: `iterator` that follows the last removed element.

Requirements: the range `[first, last)` must be a valid subrange in `*this`.

Extracting nodes

```
node_type unsafe_extract( iterator pos );

node_type unsafe_extract( const_iterator pos );
```

Transfers ownership of the element pointed to by `pos` from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element.

Requirements: the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
node_type unsafe_extract( const key_type& key );
```

If an element equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element or an empty node handle if an element equivalent to `key` was not found.

```
template <typename K>
node_type unsafe_extract( const K& key );
```

If an element equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

This overload only participates in overload resolution if all of the following statements are `true`:

- The qualified-id `key_compare::is_transparent` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: the node handle that owns the extracted element or an empty node handle if an element equivalent to `key` was not found.

swap

```
void swap( concurrent_set& other );
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`.

Otherwise, if `get_allocator() != other.get_allocator()`, the behavior is undefined.

Lookup

All methods in this section can be executed concurrently with each other, concurrently-safe modifiers and while traversing the container.

count

```
size_type count( const key_type& key );
```

Returns: the number of elements equivalent to `key`.

```
template <typename K>
size_type count( const K& key );
```

Returns: the number of elements that are equivalent to `key`.

This overload only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

find

```
iterator find( const key_type& key );

const_iterator find( const key_type& key ) const;
```

Returns: an iterator to the element equivalent to `key`, or `end()` if no such element exists.

```
template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;
```

Returns: an iterator to the element that is equivalent to `key`, or `end()` if no such element exists.

These overloads only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

contains

```
bool contains( const key_type& key ) const;
```

Returns: `true` if an element equivalent to `key` exists in the container; `false`, otherwise.

```
template <typename K>
bool contains( const K& key ) const;
```

Returns: `true` if an element equivalent to `key` exists in the container; `false`, otherwise.

This overload only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

lower_bound

```
iterator lower_bound( const key_type& key );

const_iterator lower_bound( const key_type& key ) const;
```

Returns: an iterator to the first element in the container that is *not less* than `key`.

```
template <typename K>
iterator lower_bound( const K& key )

template <typename K>
const_iterator lower_bound( const K& key ) const
```

Returns: an iterator to the first element in the container that is *not less* than `key`.

These overloads only participate in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

upper_bound

```
iterator upper_bound( const key_type& key );

const_iterator upper_bound( const key_type& key ) const;
```

Returns: an iterator to the first element in the container that compares *greater* than `key`.

```
template <typename K>
iterator upper_bound( const K& key );

template <typename K>
const_iterator upper_bound( const K& key ) const;
```

Returns: an iterator to the first element in the container that compares `greater` than `key`.

These overloads only participate in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

equal_range

```
std::pair<iterator, iterator> equal_range( const key_type& key );

std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;
```

Returns: if an element equivalent to `key` exists, a pair of iterators `{f, l}`, where `f` is an iterator to this element, `l` is `std::next(f)`. Otherwise, `{end(), end()}`.

```
template <typename K>
std::pair<iterator, iterator> equal_range( const K& key )

template <typename K>
std::pair<const_iterator, const_iterator> equal_range( const K& key )
```

Returns: if an element equivalent to `key` exists, a pair of iterators `{f, l}`, where `f` is an iterator to this element, `l` is `std::next(f)`. Otherwise, `{end(), end()}`.

These overloads only participate in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

Observers

get_allocator

```
allocator_type get_allocator() const;
```

Returns: a copy of the allocator associated with `*this`.

key_comp

```
key_compare key_comp() const;
```

Returns: a copy of the key comparison functor associated with `*this`.

value_comp

```
value_compare value_comp() const;
```

Returns: an object of the `value_compare` class that is used to compare `value_type` objects.

Parallel iteration

Member types `concurrent_set::range_type` and `concurrent_set::const_range_type` meet the *ContainerRange requirements*.

These types differ only in that the bounds for a `concurrent_set::const_range_type` are of type `concurrent_set::const_iterator`, whereas the bounds for a `concurrent_set::range_type` are of type `concurrent_set::iterator`.

range member function

```
range_type range();

const_range_type range() const;
```

Returns: a range object representing all elements in the container.

Non-member functions

These functions provide binary and lexicographical comparison and swap operations on `oneapi::tbb::concurrent_set` objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define `oneapi::tbb::concurrent_set` as a type alias for which the non-member functions are reachable only via argument-dependent lookup.

```
template <typename T, typename Compare, typename Allocator>
void swap( concurrent_set<T, Compare, Allocator>& lhs,
           concurrent_set<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator==( const concurrent_set<T, Compare, Allocator>& lhs,
                 const concurrent_set<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator!=( const concurrent_set<T, Compare, Allocator>& lhs,
```

(continues on next page)

(continued from previous page)

```
                const concurrent_set<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator<( const concurrent_set<T, Compare, Allocator>& lhs,
                const concurrent_set<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator>( const concurrent_set<T, Compare, Allocator>& lhs,
                const concurrent_set<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator<=( const concurrent_set<T, Compare, Allocator>& lhs,
                 const concurrent_set<T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator>=( const concurrent_set<T, Compare, Allocator>& lhs,
                 const concurrent_set<T, Compare, Allocator>& rhs );
```

Non-member swap

```
template <typename T, typename Compare, typename Allocator>
void swap( concurrent_set<T, Compare, Allocator>& lhs,
           concurrent_set<T, Compare, Allocator>& rhs );
```

Equivalent to `lhs.swap(rhs)`.

Non-member binary comparisons

Two `oneapi::tbb::concurrent_set` objects are equal if they have the same number of elements and each element in one container is equal to the element in other container on the same position.

```
template <typename T, typename Compare, typename Allocator>
bool operator==( const concurrent_set<T, Compare, Allocator>& lhs,
                 const concurrent_set<T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is equal to `rhs`; `false`, otherwise.

```
template <typename T, typename Compare, typename Allocator>
bool operator!=( const concurrent_set<T, Compare, Allocator>& lhs,
                 const concurrent_set<T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is not equal to `rhs`; `false`, otherwise.

Non-member lexicographical comparisons

```
template <typename T, typename Compare, typename Allocator>
bool operator<( const concurrent_set<T, Compare, Allocator>& lhs,
                const concurrent_set<T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *less* than `rhs`.

```
template <typename T, typename Compare, typename Allocator>
bool operator<=( const concurrent_set<T, Compare, Allocator>& lhs,
                 const concurrent_set<T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *less or equal* than `rhs`.

```
template <typename T, typename Compare, typename Allocator>
bool operator>( const concurrent_set<T, Compare, Allocator>& lhs,
                const concurrent_set<T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *greater* than `rhs`.

```
template <typename T, typename Compare, typename Allocator>
bool operator>=( const concurrent_set<T, Compare, Allocator>& lhs,
                 const concurrent_set<T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *greater or equal* than `rhs`.

Other

Deduction guides

If possible, `concurrent_set` constructors support class template argument deduction (since C++17). Copy and move constructors, including constructors with an explicit `allocator_type` argument, provide implicitly-generated deduction guides. In addition, the following explicit deduction guides are provided:

```
template <typename InputIterator,
          typename Compare = std::less<iterator_value_t<InputIterator>>,
          typename Allocator = tbb::tbb_allocator<iterator_value_t<InputIterator>>>
concurrent_set( InputIterator, InputIterator,
                Compare = Compare(),
                Allocator = Allocator() )
-> concurrent_set<iterator_value_t<InputIterator>,
                  Compare,
                  Allocator>;

template <typename InputIterator,
          typename Allocator>
concurrent_set( InputIterator, InputIterator,
```

(continues on next page)

(continued from previous page)

```
                Allocator )
-> concurrent_set<iterator_value_t<InputIterator>,
                  std::less<iterator_value_t<InputIterator>>,
                  Allocator>;

template <typename Key,
          typename Compare = std::less<Key>,
          typename Allocator = tbb::tbb_allocator<Key>>
concurrent_set( std::initializer_list<Key>,
                Compare = Compare(),
                Allocator = Allocator() )
-> concurrent_set<Key,
                  Compare,
                  Allocator>;

template <typename Key,
          typename Allocator>
concurrent_set( std::initializer_list<Key>,
                Allocator )
-> concurrent_set<Key,
                  std::less<Key>,
                  Allocator>;
```

Where the type alias `iterator_value_t` is defined as follows:

```
template <typename InputIterator>
using iterator_value_t = typename std::iterator_traits<InputIterator>::value_type;
```

These deduction guides only participate in the overload resolution if the following requirements are met:

- The `InputIterator` type meets the `InputIterator` requirements described in the [input.iterators] section of the ISO C++ Standard.
- The `Allocator` type meets the `Allocator` requirements described in the [allocator.requirements] section of the ISO C++ Standard.
- The `Compare` type does not meet the `Allocator` requirements.

Example

```
#include <oneapi/tbb/concurrent_set.h>
#include <vector>

int main() {
    std::vector<int> v;

    // Deduces cs1 as concurrent_set<int>
    oneapi::tbb::concurrent_set cs1(v.begin(), v.end());

    // Deduces cs2 as concurrent_set<int>
    oneapi::tbb::concurrent_set cs2({1, 2, 3});
}
```

concurrent_multiset

[containers.concurrent_multiset]

`oneapi::tbb::concurrent_multiset` is a class template that represents a sorted sequence of elements. It supports concurrent insertion, lookup, and traversal, but does not support concurrent erasure. In this container, multiple equivalent elements can be stored.

Class Template Synopsis

```
// Defined in header <oneapi/tbb/concurrent_set.h>

namespace oneapi {
    namespace tbb {

        template <typename T,
                  typename Compare = std::less<T>,
                  typename Allocator = tbb_allocator<T>>
        class concurrent_multiset {
        public:
            using key_type = T;
            using value_type = T;

            using size_type = <implementation-defined unsigned integer type>;
            using difference_type = <implementation-defined signed integer type>;

            using key_compare = Compare;
            using value_compare = Compare;

            using allocator_type = Allocator;

            using reference = value_type&;
            using const_reference = const value_type&;
            using pointer = std::allocator_traits<Allocator>::pointer;
            using const_pointer = std::allocator_traits<Allocator>::const_pointer;

            using iterator = <implementation-defined ForwardIterator>;
            using const_iterator = <implementation-defined constant ForwardIterator>;

            using node_type = <implementation-defined node handle>;

            using range_type = <implementation-defined range>;
            using const_range_type = <implementation-defined constant node handle>;

            // Construction, destruction, copying
            concurrent_multiset();
            explicit concurrent_multiset( const key_compare& comp,
                                          const allocator_type& alloc = allocator_type()␣
↪);

            explicit concurrent_multiset( const allocator_type& alloc );
```

(continues on next page)

(continued from previous page)

```
            template <typename InputIterator>
            concurrent_multiset( InputIterator first, InputIterator last,
                                 const key_compare& comp = key_compare(),
                                 const allocator_type& alloc = allocator_type() );

            template <typename InputIterator>
            concurrent_multiset( InputIterator first, InputIterator last,
                                 const allocator_type& alloc );

            concurrent_multiset( std::initializer_list<value_type> init,
                                 const key_compare& comp = key_compare(),
                                 const allocator_type& alloc = allocator_type() );

            concurrent_multiset( std::initializer_list<value_type> init, const allocator_
↪type& alloc );

            concurrent_multiset( const concurrent_multiset& other );
            concurrent_multiset( const concurrent_multiset& other,
                                 const allocator_type& alloc );

            concurrent_multiset( concurrent_multiset&& other );
            concurrent_multiset( concurrent_multiset&& other,
                                 const allocator_type& alloc );

            ~concurrent_multiset();

            concurrent_multiset& operator=( const concurrent_multiset& other );
            concurrent_multiset& operator=( concurrent_multiset&& other );
            concurrent_multiset& operator=( std::initializer_list<value_type> init );

            allocator_type get_allocator() const;

            // Iterators
            iterator begin();
            const_iterator begin() const;
            const_iterator cbegin() const;

            iterator end();
            const_iterator end() const;
            const_iterator cend() const;

            // Size and capacity
            bool empty() const;
            size_type size() const;
            size_type max_size() const;

            // Concurrently safe modifiers
            std::pair<iterator, bool> insert( const value_type& value );

            iterator insert( const_iterator hint, const value_type& value );

            std::pair<iterator, bool> insert( value_type&& value );
```

(continues on next page)

(continued from previous page)

```
        iterator insert( const_iterator hint, value_type&& value );

        template <typename InputIterator>
        void insert( InputIterator first, InputIterator last );

        void insert( std::initializer_list<value_type> init );

        std::pair<iterator, bool> insert( node_type&& nh );
        iterator insert( const_iterator hint, node_type&& nh );

        template <typename... Args>
        std::pair<iterator, bool> emplace( Args&&... args );

        template <typename... Args>
        iterator emplace_hint( const_iterator hint, Args&&... args );

        template <typename SrcCompare>
        void merge( concurrent_set<T, SrcCompare, Allocator>& source );

        template <typename SrcCompare>
        void merge( concurrent_set<T, SrcCompare, Allocator>&& source );

        template <typename SrcCompare>
        void merge( concurrent_multiset<T, SrcCompare, Allocator>& source );

        template <typename SrcCompare>
        void merge( concurrent_multiset<T, SrcCompare, Allocator>&& source );

        // Concurrently unsafe modifiers
        void clear();

        iterator unsafe_erase( const_iterator pos );
        iterator unsafe_erase( iterator pos );

        iterator unsafe_erase( const_iterator first, const_iterator last );

        size_type unsafe_erase( const key_type& key );

        template <typename K>
        size_type unsafe_erase( const K& key );

        node_type unsafe_extract( const_iterator pos );
        node_type unsafe_extract( iterator pos );

        node_type unsafe_extract( const key_type& key );

        template <typename K>
        node_type unsafe_extract( const K& key );

        void swap( concurrent_multiset& other );
```

(continues on next page)

(continued from previous page)

```
            // Lookup
            size_type count( const key_type& key );

            template <typename K>
            size_type count( const K& key );

            iterator find( const key_type& key );
            const_iterator find( const key_type& key ) const;

            template <typename K>
            iterator find( const K& key );

            template <typename K>
            const_iterator find( const K& key ) const;

            bool contains( const key_type& key ) const;

            template <typename K>
            bool contains( const K& key ) const;

            std::pair<iterator, iterator> equal_range( const key_type& key );
            std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;

            template <typename K>
            std::pair<iterator, iterator>  equal_range( const K& key );
            std::pair<const_iterator, const_iterator> equal_range( const K& key ) const;

            iterator lower_bound( const key_type& key );
            const_iterator lower_bound( const key_type& key ) const;

            template <typename K>
            iterator lower_bound( const K& key );

            template <typename K>
            const_iterator lower_bound( const K& key ) const;

            iterator upper_bound( const key_type& key );
            const_iterator upper_bound( const key_type& key ) const;

            template <typename K>
            iterator upper_bound( const K& key );

            template <typename K>
            const_iterator upper_bound( const K& key ) const;

            // Observers
            key_compare key_comp() const;

            value_compare value_comp() const;

            // Parallel iteration
```

(continues on next page)

(continued from previous page)

```
            range_type range();
            const_range_type range() const;
        }; // class concurrent_multiset

    } // namespace tbb
} // namespace oneapi
```

Requirements:

- The expression `std::allocator_traits<Allocator>::destroy(m, val)`, where `m` is an object of the type `Allocator` and `val` is an object of the type `value_type`, should be well-formed. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Compare` must meet the `Compare` requirements from the [alg.sorting] ISO C++ Standard section.
- The type `Allocator` must meet the `Allocator` requirements from the [allocator.requirements] ISO C++ Standard section.

Member functions

Construction, destruction, copying

Empty container constructors

```
concurrent_multiset();

explicit concurrent_multiset( const key_compare& comp,
                              const allocator_type& alloc = allocator_type() );

explicit concurrent_multiset( const allocator_type& alloc );
```

Constructs an empty `concurrent_multiset`.

If provided, uses the comparison function object `comp` for all `key_type` comparisons and the allocator `alloc` to allocate the memory.

Constructors from the sequence of elements

```
template <typename InputIterator>
concurrent_multiset( InputIterator first, InputIterator last,
                     const key_compare& comp = key_compare(),
                     const allocator_type& alloc = allocator_type() );

template <typename InputIterator>
concurrent_multiset( InputIterator first, InputIterator last,
                     const allocator_type& alloc = allocator_type() );
```

Constructs the `concurrent_multiset`, which contains all elements from the half-open interval `[first, last)`.

If provided, uses the comparison function object `comp` for all `key_type` comparisons and the allocator `alloc` to allocate the memory.

Requirements: the type `InputIterator` must meet the requirements of *InputIterator* from the `[input.iterators]` ISO C++ Standard section.

```
concurrent_multiset( std::initializer_list<value_type> init, const key_compare&
↪ comp = key_compare(),
                     const allocator_type& alloc = allocator_type() );
```

Equivalent to `concurrent_multiset(init.begin(), init.end(), comp, alloc)`.

```
concurrent_multiset( std::initializer_list<value_type> init,
                     const allocator_type& alloc );
```

Equivalent to `concurrent_multiset(init.begin(), init.end(), alloc)`.

Copying constructors

```
concurrent_multiset( const concurrent_multiset& other );

concurrent_multiset( const concurrent_multiset& other, const allocator_type&␣
↪alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Moving constructors

```
concurrent_multiset( concurrent_multiset&& other );

concurrent_multiset( concurrent_multiset&& other, const allocator_type& alloc␣
↪);
```

Constructs a *concurrent_multiset* with the contents of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by calling `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Destructor

```
~concurrent_multiset();
```

Destroys the `concurrent_multiset`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

Assignment operators

```
concurrent_multiset& operator=( const concurrent_multiset& other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_assignment::` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_multiset& operator=( concurrent_multiset&& other );
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment:` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_multiset& operator=( std::initializer_list<value_type> init );
```

Replaces all elements in `*this` by the elements in `init`.

If `init` contains multiple elements with equal keys, it is unspecified which element would be inserted.

The behavior is undefined in case of concurrent operations with `*this`.

Returns: a reference to `*this`.

Iterators

The types `concurrent_multiset::iterator` and `concurrent_multiset::const_iterator` meet the requirements of `ForwardIterator` from the [forward.iterators] ISO C++ standard section.

begin and cbegin

```
iterator begin();

const_iterator begin() const;

const_iterator cbegin() const;
```

Returns: an iterator to the first element in the container.

end and cend

```
iterator end();

const_iterator end() const;

const_iterator cend() const;
```

Returns: an iterator to the element that follows the last element in the container.

Size and capacity

empty

```
bool empty() const;
```

Returns: `true` if the container is empty; `false`, otherwise.

The result may differ from the actual container state in case of pending concurrent insertions.

size

```
size_type size() const;
```

Returns: the number of elements in the container.

The result may differ from the actual container size in case of pending concurrent insertions.

max_size

```
size_type max_size() const;
```

Returns: the maximum number of elements that container can hold.

Concurrently safe modifiers

All member functions in this section can be performed concurrently with each other, lookup methods and while traversing the container.

Inserting values

```
std::pair<iterator, bool> insert( const value_type& value );
```

Inserts the value `value` into the container.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element. Boolean value is always `true`.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, const value_type& other );
```

Inserts the value `value` into the container.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

Returns: an `iterator` to the inserted element.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
std::pair<iterator, bool> insert( value_type&& value );
```

Inserts the value `value` into the container using move semantics.

`value` is left in a valid, but unspecified state.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element. Boolean value is always `true`.

Requirements: the type `value_type` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, value_type&& other );
```

Inserts the value `value` into the container using move semantics.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

`value` is left in a valid, but unspecified state.

Returns: an `iterator` to the inserted element.

Requirements: the type `value_type` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

Inserting sequences of elements

```
template <typename InputIterator>
void insert( InputIterator first, InputIterator last );
```

Inserts all items from the half-open interval `[first, last)` into the container.

Requirements: the type `InputIterator` must meet the requirements of *InputIterator* from the `[input.iterators]` ISO C++ Standard section.

```
void insert( std::initializer_list<value_type> init );
```

Equivalent to `insert(init.begin(), init.end())`.

Inserting nodes

```
std::pair<iterator, bool> insert( node_type&& nh );
```

If the node handle `nh` is empty, does nothing.

Otherwise, inserts the node owned by `nh` into the container.

`nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element. Boolean value is always `true`.

```
iterator insert( const_iterator hint, node_type&& nh );
```

If the node handle `nh` is empty, does nothing.

Otherwise, inserts the node owned by `nh` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

`nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: an iterator pointing to the inserted element.

Emplacing elements

```
template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args );
```

Inserts an element, constructed in-place from `args` into the container.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element. Boolean value is always `true`.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ section.

```
template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args );
```

Inserts an element constructed in-place from `args` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

Returns: an `iterator` to the inserted element.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ section.

Merging containers

```
template <typename SrcCompare>
void merge( concurrent_set<T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_set<T, SrcCompare, Allocator>&& source );

template <typename SrcCompare>
void merge( concurrent_multiset<T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_multiset<T, SrcCompare, Allocator>&& source );
```

Transfers all elements from `source` to `*this`.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `get_allocator() != source.get_allocator()`.

Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

Clearing

```
void clear();
```

Removes all elements from the container.

Erasing elements

```
iterator unsafe_erase( const_iterator pos );

iterator unsafe_erase( iterator pos );
```

Removes the element pointed to by `pos` from the container.

Invalidates all iterators and references to the removed element.

Returns: `iterator` that follows the removed element.

Requirements: the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
size_type unsafe_erase( const key_type& key );
```

Removes all elements equivalent to `key` if they exist in the container.

Invalidates all iterators and references to the removed element.

Returns: the number of removed elements.

```
template <typename K>
size_type unsafe_erase( const K& key );
```

Removes all elements that are equivalent to `key` if they exist in the container.

Invalidates all iterators and references to the removed element.

This overload only participates in overload resolution if all of the following statements are `true`:

- The qualified-id `key_compare::is_transparent` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: the number of removed elements.

Erasing sequences

```
iterator unsafe_erase( const_iterator first, const_iterator last );
```

Removes all elements from the half-open interval `[first, last)` from the container.

Returns: `iterator` that follows the last removed element.

Requirements: the range `[first, last)` must be a valid subrange in `*this`.

Extracting nodes

```
node_type unsafe_extract( iterator pos );

node_type unsafe_extract( const_iterator pos );
```

Transfers ownership of the element pointed to by `pos` from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element.

Requirements: the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
node_type unsafe_extract( const key_type& key );
```

If an element equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

If there are multiple elements equivalent to `key`, it is unspecified which element should be transferred.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element or an empty node handle if an element equivalent to `key` was not found.

```
template <typename K>
node_type unsafe_extract( const K& key );
```

If an element that is equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

If there are multiple elements equivalent to `key`, it is unspecified which element should be transferred.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

This overload only participates in overload resolution if all of the following statements are `true`:

- The qualified-id `key_compare::is_transparent` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: the node handle that owns the extracted element or an empty node handle if an element equivalent to `key` was not found.

swap

```
void swap( concurrent_multiset& other );
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`.

Otherwise, if `get_allocator() != other.get_allocator()`, the behavior is undefined.

Lookup

All methods in this section can be executed concurrently with each other, concurrently-safe modifiers and while traversing the container.

count

```
size_type count( const key_type& key );
```

Returns: the number of elements equivalent to `key`.

```
template <typename K>
size_type count( const K& key );
```

Returns: the number of elements that are equivalent to `key`.

This overload only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

find

```
iterator find( const key_type& key );

const_iterator find( const key_type& key ) const;
```

Returns: an iterator to the element equivalent to `key` or `end()` if no such element exists.

If there are multiple elements equivalent to `key`, it is unspecified which element should be found.

```
template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;
```

Returns: an iterator to the element that is equivalent to `key`, or `end()` if no such element exists.

If there are multiple elements that are equivalent to `key`, it is unspecified which element should be found.

These overloads only participate in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

contains

```
bool contains( const key_type& key ) const;
```

Returns: `true` if at least one element equivalent to `key` exists in the container; `false`, otherwise.

```
template <typename K>
bool contains( const K& key ) const;
```

Returns: `true` if at least one element that is equivalent to `key` exists in the container; `false`, otherwise.

This overload participates in overload resolution only if qualified-id `key_compare::is_transparent` is valid and denotes a type.

lower_bound

```
iterator lower_bound( const key_type& key );

const_iterator lower_bound( const key_type& key ) const;
```

Returns: an iterator to the first element in the container that is *not less* than `key`.

```
template <typename K>
iterator lower_bound( const K& key )

template <typename K>
const_iterator lower_bound( const K& key ) const
```

Returns: an iterator to the first element in the container that is *not less* than `key`.

These overloads only participate in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

upper_bound

```
iterator upper_bound( const key_type& key );

const_iterator upper_bound( const key_type& key ) const;
```

Returns: an iterator to the first element in the container that compares *greater* than `key`.

```
template <typename K>
iterator upper_bound( const K& key );

template <typename K>
const_iterator upper_bound( const K& key ) const;
```

Returns: an iterator to the first element in the container that compares `greater` than `key`.

These overloads participate in overload resolution only if qualified-id `key_compare::is_transparent` is valid and denotes a type.

equal_range

```
std::pair<iterator, iterator> equal_range( const key_type& key );

std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;
```

Returns: if at least one element equivalent to `key` exists, a pair of iterators `{f, l}`, where `f` is an iterator to the first element equivalent to `key`, `l` is an iterator to the element that follows the last element equivalent to `key`. Otherwise, `{end(), end()}`.

```
template <typename K>
std::pair<iterator, iterator> equal_range( const K& key )

template <typename K>
std::pair<const_iterator, const_iterator> equal_range( const K& key )
```

Returns: if at least one element that is equivalent to `key` exists, a pair of iterators `{f, l}`, where `f` is an iterator to the first element that is equivalent to `key`, `l` is an iterator to the element that follows the last element that is equivalent to `key`. Otherwise, `{end(), end()}`.

These overloads participate in overload resolution only if qualified-id `key_compare::is_transparent` is valid and denotes a type.

Observers

get_allocator

```
allocator_type get_allocator() const;
```

Returns: a copy of the allocator associated with `*this`.

key_comp

```
key_compare key_comp() const;
```

Returns: a copy of the key comparison functor associated with `*this`.

value_comp

```
value_compare value_comp() const;
```

Returns: an object of the `value_compare` class that is used to compare `value_type` objects.

Parallel iteration

Member types `concurrent_multiset::range_type` and `concurrent_multiset::const_range_type` meet the *ContainerRange requirements*.

These types differ only in that the bounds for a `concurrent_multiset::const_range_type` are of type `concurrent_multiset::const_iterator`, whereas the bounds for a `concurrent_multiset::range_type` are of type `concurrent_multiset::iterator`.

range member function

```
range_type range();

const_range_type range() const;
```

Returns: a range object representing all elements in the container.

Non-member functions

These functions provides binary and lexicographical comparison and swap operations on `oneapi::tbb::concurrent_multiset` objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define `oneapi::tbb::concurrent_multiset` as a type alias for which the non-member functions are reachable only via argument dependent lookup.

```
template <typename T, typename Compare, typename Allocator>
void swap( concurrent_multiset<T, Compare, Allocator>& lhs,
           concurrent_multiset<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator==( const concurrent_multiset<T, Compare, Allocator>& lhs,
                 const concurrent_multiset<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator!=( const concurrent_multiset<T, Compare, Allocator>& lhs,
                 const concurrent_multiset<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator<( const concurrent_multiset<T, Compare, Allocator>& lhs,
                const concurrent_multiset<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator>( const concurrent_multiset<T, Compare, Allocator>& lhs,
                const concurrent_multiset<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator<=( const concurrent_multiset<T, Compare, Allocator>& lhs,
                 const concurrent_multiset<T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator>=( const concurrent_multiset<T, Compare, Allocator>& lhs,
                 const concurrent_multiset<T, Compare, Allocator>& rhs );
```

Non-member swap

```
template <typename T, typename Compare, typename Allocator>
void swap( concurrent_multiset<T, Compare, Allocator>& lhs,
           concurrent_multiset<T, Compare, Allocator>& rhs );
```

Equivalent to `lhs.swap(rhs)`.

Non-member binary comparisons

Two `oneapi::tbb::concurrent_multiset` objects are equal if they have the same number of elements and each element in one container is equal to the element in other container on the same position.

```
template <typename T, typename Compare, typename Allocator>
bool operator==( const concurrent_multiset<T, Compare, Allocator>& lhs,
                 const concurrent_multiset<T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is equal to `rhs`; `false`, otherwise.

```
template <typename T, typename Compare, typename Allocator>
bool operator!=( const concurrent_multiset<T, Compare, Allocator>& lhs,
                 const concurrent_multiset<T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is not equal to `rhs`; `false`, otherwise.

Non-member lexicographical comparisons

```
template <typename T, typename Compare, typename Allocator>
bool operator<( const concurrent_multiset<T, Compare, Allocator>& lhs,
                const concurrent_multiset<T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *less* than `rhs`.

```
template <typename T, typename Compare, typename Allocator>
bool operator<=( const concurrent_multiset<T, Compare, Allocator>& lhs,
                 const concurrent_multiset<T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *less or equal* than `rhs`.

```
template <typename T, typename Compare, typename Allocator>
bool operator>( const concurrent_multiset<T, Compare, Allocator>& lhs,
                const concurrent_multiset<T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *greater* than `rhs`.

```
template <typename T, typename Compare, typename Allocator>
bool operator>=( const concurrent_multiset<T, Compare, Allocator>& lhs,
                 const concurrent_multiset<T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *greater or equal* than `rhs`.

Other

Deduction guides

If possible, `concurrent_multiset` constructors support class template argument deduction (since C++17). Copy and move constructors, including constructors with an explicit `allocator_type` argument, provide implicitly-generated deduction guides. In addition, the following explicit deduction guides are provided:

```
template <typename InputIterator,
          typename Compare = std::less<iterator_value_t<InputIterator>>,
          typename Allocator = tbb::tbb_allocator<iterator_value_t<InputIterator>>>
concurrent_multiset( InputIterator, InputIterator,
                     Compare = Compare(),
                     Allocator = Allocator() )
```

(continues on next page)

(continued from previous page)

```
-> concurrent_multiset<iterator_value_t<InputIterator>,
                       Compare,
                       Allocator>;

template <typename InputIterator,
          typename Allocator>
concurrent_multiset( InputIterator, InputIterator,
                     Allocator )
-> concurrent_multiset<iterator_value_t<InputIterator>,
                       std::less<iterator_value_t<InputIterator>>,
                       Allocator>;

template <typename Key,
          typename Compare = std::less<Key>,
          typename Allocator = tbb::tbb_allocator<Key>>
concurrent_multiset( std::initializer_list<Key>,
                     Compare = Compare(),
                     Allocator = Allocator() )
-> concurrent_multiset<Key,
                       Compare,
                       Allocator>;

template <typename Key,
          typename Allocator>
concurrent_multiset( std::initializer_list<Key>,
                     Allocator )
-> concurrent_multiset<Key,
                       std::less<Key>,
                       Allocator>;
```

Where the type alias `iterator_value_t` is defined as follows:

```
template <typename InputIterator>
using iterator_value_t = typename std::iterator_traits<InputIterator>::value_type;
```

These deduction guides only participate in the overload resolution if the following requirements are met:

- The `InputIterator` type meets the `InputIterator` requirements described in the [input.iterators] section of the ISO C++ Standard.
- The `Allocator` type meets the `Allocator` requirements described in the [allocator.requirements] section of the ISO C++ Standard.
- The `Compare` type does not meet the `Allocator` requirements.

Example

```
#include <oneapi/tbb/concurrent_set.h>
#include <vector>

int main() {
    std::vector<int> v;

    // Deduces cs1 as concurrent_multiset<int>
```

(continues on next page)

(continued from previous page)

```
    oneapi::tbb::concurrent_multiset cs1(v.begin(), v.end());

    // Deduces cs2 as concurrent_multiset<int>
    oneapi::tbb::concurrent_multiset cs2({1, 2, 3});
}
```

Auxiliary classes

tbb_hash_compare

[containers.tbb_hash_compare]

`oneapi::tbb::tbb_hash_compare` is a class template for hash support. Use it with the `oneapi::tbb::concurrent_hash_map` associative container to calculate hash codes and compare keys for equality.

`tbb_hash_compare` meets the *HashCompare requirements*.

Class Template Synopsis

```
// Defined in header <oneapi/tbb/concurrent_hash_map.h>

namespace oneapi {
    namespace tbb {

        template <typename Key>
        class tbb_hash_compare {
            static std::size_t hash( const Key& k );
            static bool equal( const Key& k1, const Key& k2 );
        }; // class tbb_hash_compare

    } // namespace tbb
} // namespace oneapi
```

Member functions

```
static std::size_t hash( const Key& k );
```

Returns: a hash code for a key `k`.

```
static bool equal( const Key& k1, const Key& k2 );
```

Equivalent to `k1 == k2`.

Returns: `true` if the keys are equal; `false`, otherwise.

Node handles

[containers.node_handles]

Concurrent associative containers (`concurrent_map`, `concurrent_multimap`, `concurrent_set`, `concurrent_multiset`, `concurrent_unordered_map`, `concurrent_unordered_multimap`, `concurrent_unordered_set`, and `concurrent_unordered_multiset`) store elements in individually allocated, connected nodes. These containers support data transfer between containers with compatible node types by changing the connections without copying or moving the actual data.

Class synopsis

```
class node-handle { // Exposition-only name
public:
    using key_type = <container-specific>;    // Only for maps
    using mapped_type = <container-specific>; // Only for maps
    using value_type = <container-specific>;  // Only for sets
    using allocator_type = <container-specific>;

    node-handle();
    node-handle( node-handle&& other );

    ~node-handle();

    node-handle& operator=( node-handle&& other );

    void swap( node-handle& nh );

    bool empty() const;
    explicit operator bool() const;

    key_type& key() const;       // Only for maps
    mapped_type& mapped() const; // Only for maps
    value_type& value() const;   // Only for sets

    allocator_type get_allocator() const;
};
```

A `node handle` is a container-specific move-only nested type (exposed as *container::node_type*) that represents a node outside of any container instance. It allows reading and modifying the data stored in the node, and inserting the node into a compatible container instance. The following containers have compatible node types and may exchange nodes:

- `concurrent_map` and `concurrent_multimap` with the same `key_type`, `mapped_type` and `allocator_type`.
- `concurrent_set` and `concurrent_multiset` with the same `value_type` and `allocator_type`.
- `concurrent_unordered_map` and `concurrent_unordered_multimap` with the same `key_type`, `mapped_type` and `allocator_type`.
- `concurrent_unordered_set` and `concurrent_unordered_multiset` with the same `value_type` and `allocator_type`.

Default or moved-from node handles are *empty* and do not represent a valid node. A non-empty node handle is typically created when a node is extracted out of a container, for example, with the `unsafe_extract` method. It stores the node along with a copy of the container's allocator. Upon assignment or destruction a non-empty node handle destroys the stored data and deallocates the node.

Member functions

Constructors

```
node_handle();
```

Constructs an empty node handle.

```
node_handle( node_handle&& other );
```

Constructs a node handle that takes ownership of the node from `other`.

`other` is left in an empty state.

Assignment

```
node_handle& operator=( node_handle&& other );
```

Transfers ownership of the node from `other` to `*this`. If `*this` was not empty before transferring, destroys and deallocates the stored node.

Move assigns the stored allocator if `std::allocator_traits<allocator_type>::propagate_on_container_move_assig` is `true`.

`other` is left in an empty state.

Destructor

```
~node_handle();
```

Destroys the node handle. If it is not empty, destroys and deallocates the owned node.

Swap

```
void swap( node_handle& other )
```

Exchanges the nodes owned by `*this` and `other`.

Swaps the stored allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`.

State

```
bool empty() const;
```

Returns: `true` if the node handle is empty, `false` otherwise.

```
explicit operator bool() const;
```

Equivalent to `!empty()`.

Access to the stored element

```
key_type& key() const;
```

Available only for map node handles.

Returns: a reference to the key of the element stored in the owned node.

The behavior is undefined if the node handle is empty.

```
mapped_type& mapped() const;
```

Available only for map node handles.

Returns: a reference to the value of the element stored in the owned node.

The behavior is undefined if the node handle is empty.

```
value_type& value() const;
```

Available only for set node handles.

Returns: a reference to the element stored in the owned node.

The behavior is undefined if the node handle is empty.

get_allocator

```
allocator_type get_allocator() const;
```

Returns: a copy of the allocator stored in the node handle.

The behavior is undefined if the node handle is empty.

8.2.6 Thread Local Storage

[thread_local_storage]

oneAPI Threading Building Blocks provides class templates for thread local storage (TLS). Each provides a thread-local element per thread and lazily creates elements on demand.

combinable

[tls.combinable]

A class template for holding thread-local values during a parallel computation that will be merged into a final value.

A `combinable` provides each thread with its own instance of type `T`.

```
// Defined in header <oneapi/tbb/combinable.h>

namespace oneapi {
namespace tbb {
    template <typename T>
    class combinable {
    public:
        combinable();

        combinable(const combinable& other);
        combinable(combinable&& other);

        template <typename FInit>
        explicit combinable(FInit finit);

        ~combinable();

        combinable& operator=( const combinable& other);
        combinable& operator=( combinable&& other);

        void clear();

        T& local();
        T& local(bool & exists);

        template<typename BinaryFunc> T combine(BinaryFunc f);
        template<typename UnaryFunc> void combine_each(UnaryFunc f);
    };
} // namespace tbb
} // namespace oneapi
```

Member functions

combinable()

Constructs `combinable` such that thread-local instances of `T` will be default-constructed.

template<typename **FInit**>
explicit **combinable**(*FInit* finit)

Constructs `combinable` such that thread-local elements will be created by copying the result of *finit()*.

> **Caution:** The expression *finit()* must be safe to evaluate concurrently by multiple threads. It is evaluated each time a new thread-local element is created.

combinable(const *combinable* &other)

Constructs a copy of *other*, so that it has copies of each element in *other* with the same thread mapping.

combinable(*combinable* &&other)

Constructs `combinable` by moving the content of *other* intact. *other* is left in an unspecified state but can be safely destroyed.

~combinable()

Destroys all elements in `*this`.

combinable &**operator=**(const combinable &other)

Sets `*this` to be a copy of *other*. Returns a reference to `*this`.

combinable &**operator=**(combinable &&other)

Moves the content of *other* to `*this` intact. *other* is left in an unspecified state but can be safely destroyed. Returns a reference to `*this`.

void **clear**()

Removes all elements from `*this`.

T &**local**()

If an element does not exist for the current thread, creates it.

Returns: Reference to thread-local element.

T &**local**(bool &exists)

Similar to `local()`, except that *exists* is set to true if an element was already present for the current thread; false, otherwise.

Returns: Reference to thread-local element.

template<typename **BinaryFunc**>
T **combine**(*BinaryFunc* f)

Requires: A `BinaryFunc` must meet the *Function Objects* requirements described in the [function.objects] section of the ISO C++ standard. Specifically, the type should be an associative binary functor with the signature `T BinaryFunc(T,T)` or `T BinaryFunc(const T&,const T&)`. A `T` type must be the same as a corresponding template parameter for the `combinable` object.

Effects: Computes a reduction over all elements using binary functor *f*. All evaluations of *f* are done sequentially in the calling thread. If there are no elements, creates the result using the same rules as for creating a new element.

Returns: Result of the reduction.

template<typename **UnaryFunc**>

void **combine_each**(*UnaryFunc* f)

Requires: An `UnaryFunc` must meet the *Function Objects* requirements described in the [function.objects] section of the ISO C++ standard. Specifically, the type should be an unary functor with the one of the signatures: `void UnaryFunc(T)`, `void UnaryFunc(T&)`, or `void UnaryFunc(const T&)` A `T` type must be the same as a corresponding template parameter for the `enumerable_thread_specific` object.

Effects: Evaluates *f(x)* for each thread-local element *x* in `*this`. All evaluations are done sequentially in the calling thread.

Note: Methods of `class combinable` are not thread-safe, except for `local`.

---

enumerable_thread_specific

[tls.enumerable_thread_specific]

A class template for thread local storage (TLS).

```
// Defined in header <oneapi/tbb/enumerable_thread_specific.h>

namespace oneapi {
namespace tbb {

    enum ets_key_usage_type {
        ets_key_per_instance,
        ets_no_key,
        ets_suspend_aware
    };

    template <typename T,
        typename Allocator=cache_aligned_allocator<T>,
        ets_key_usage_type ETS_key_type=ets_no_key >
    class enumerable_thread_specific {
    public:
        // Basic types
        using value_type = T;
        using reference = T&;
        using const_reference = const T&;
        using pointer = T*;
        using size_type = /* implementation-defined */;
        using difference_type = /* implementation-defined */;
        using allocator_type = Allocator;

        // Iterator types
        using iterator = /* implementation-defined */;
        using const_iterator = /* implementation-defined */;

        // Parallel range types
        using range_type = /* implementation-defined */;
        using const_range_type = /* implementation-defined */;

        // Construction
        enumerable_thread_specific();
```

(continues on next page)

(continued from previous page)

```
        template <typename Finit>
        explicit enumerable_thread_specific( Finit finit );
        explicit enumerable_thread_specific( const T& exemplar );
        explicit enumerable_thread_specific( T&& exemplar );
        template <typename... Args>
        enumerable_thread_specific( Args&&... args );

        // Destruction
        ~enumerable_thread_specific();

        // Copy constructors
        enumerable_thread_specific( const enumerable_thread_specific& other);
        template<typename Alloc, ets_key_usage_type Cachetype>
        enumerable_thread_specific( const enumerable_thread_specific<T, Alloc, Cachetype>
↪& other);
        // Copy assignments
        enumerable_thread_specific& operator=( const enumerable_thread_specific& other );
        template<typename Alloc, ets_key_usage_type Cachetype>
        enumerable_thread_specific& operator=( const enumerable_thread_specific<T, Alloc,
↪ Cachetype>& other );

        // Move constructors
        enumerable_thread_specific( enumerable_thread_specific&& other);
        template<typename Alloc, ets_key_usage_type Cachetype>
        enumerable_thread_specific( enumerable_thread_specific<T, Alloc, Cachetype>&&␣
↪other);
        // Move assignments
        enumerable_thread_specific& operator=( enumerable_thread_specific&& other );
        template<typename Alloc, ets_key_usage_type Cachetype>
        enumerable_thread_specific& operator=( enumerable_thread_specific<T, Alloc,␣
↪Cachetype>&& other );

        // Other whole container operations
        void clear();

        // Concurrent operations
        reference local();
        reference local( bool& exists );
        size_type size() const;
        bool empty() const;

        // Combining
        template<typename BinaryFunc> T combine( BinaryFunc f );
        template<typename UnaryFunc> void combine_each( UnaryFunc f );

        // Parallel iteration
        range_type range( size_t grainsize=1 );
        const_range_type range( size_t grainsize=1 ) const;

        // Iterators
        iterator begin();
        iterator end();
```

(continues on next page)

(continued from previous page)

```
        const_iterator begin() const;
        const_iterator end() const;
    };

} // namespace tbb
} // namespace oneapi
```

A class template `enumerable_thread_specific` provides TLS for elements of type `T`. A class template `enumerable_thread_specific` acts as a container by providing iterators and ranges across all of the thread-local elements.

The thread-local elements are created lazily. A freshly constructed `enumerable_thread_specific` has no elements. When a thread requests access to an `enumerable_thread_specific`, it creates an element corresponding to that thread. The number of elements is equal to the number of distinct threads that have accessed the `enumerable_thread_specific` and not necessarily the number of threads in use by the application. Clearing an `enumerable_thread_specific` removes all its elements.

Use the `ETS_key_usage_type` parameter type to select an underlying implementation.

Caution: `enumerable_thread_specific` uses the OS-specific value returned by `std::this_thread::get_id()` to identify threads. This value is not guaranteed to be unique except for the life of the thread. A newly created thread may get an OS-specific ID equal to that of an already destroyed thread. The number of elements of the `enumerable_thread_specific` may therefore be less than the number of actual distinct threads that have called `local()`, and the element returned by the first reference by a thread to the `enumerable_thread_specific` may not be newly-constructed.

Member functions

Construction, destruction, copying

Empty container constructors

```
enumerable_thread_specific();
```

Constructs an `enumerable_thread_specific` where each thread-local element will be default-constructed.

```
template<typename Finit> explicit enumerable_thread_specific( Finit finit );
```

Constructs an `enumerable_thread_specific` such that any thread-local element will be created by copying the result of `finit()`.

Note: The expression `finit()` must be safe to evaluate concurrently by multiple threads. It is evaluated each time a thread-local element is created.

```
explicit enumerable_thread_specific( const T& exemplar );
```

Constructs an `enumerable_thread_specific` where each thread-local element will be copy-constructed from `exemplar`.

```
explicit enumerable_thread_specific( T&& exemplar );
```

Constructs an `enumerable_thread_specific` object, move constructor of `T` can be used to store `exemplar` internally; however, thread-local elements are always copy-constructed.

```
template <typename... Args> enumerable_thread_specific( Args&&... args );
```

Constructs `enumerable_thread_specific` such that any thread-local element will be constructed by invoking `T(args...)`.

Note: This constructor does not participate in overload resolution if the type of the first argument in `args...` is `T`, or `enumerable_thread_specific<T>`, or `foo()` is a valid expression for a value `foo` of that type.

Copying constructors

```
enumerable_thread_specific ( const enumerable_thread_specific& other );

template<typename Alloc, ets_key_usage_type Cachetype> enumerable_thread_specific( const␣
↪enumerable_thread_specific <T, Alloc, Cachetype>& other );
```

Constructs an `enumerable_thread_specific` as a copy of `other`. The values are copy-constructed from the values in `other` and have same thread correspondence.

Moving constructors

```
enumerable_thread_specific ( enumerable_thread_specific&& other )
```

Constructs an `enumerable_thread_specific` by moving the content of `other` intact. `other` is left in an unspecified state, but can be safely destroyed.

```
template<typename Alloc, ets_key_usage_type Cachetype> enumerable_thread_specific(␣
↪enumerable_thread_specific <T, Alloc, Cachetype>&& other )
```

Constructs an `enumerable_thread_specific` using per-element move construction from the values in `other`, and keeping their thread correspondence. `other` is left in an unspecified state, but can be safely destroyed.

Destructor

```
~enumerable_thread_specific()
```

Destroys all elements in `*this`. Destroys any native TLS keys that were created for this instance.

Assignment operators

```
enumerable_thread_specific& operator=( const enumerable_thread_specific& other );
```

Copies the content of `other` to `*this`. Returns a reference to `this*`.

```
template<typename Alloc, ets_key_usage_type Cachetype>
enumerable_thread_specific& operator=( const enumerable_thread_specific<T, Alloc,␣
↪Cachetype>& other );
```

Copies the content of `other` to `*this`. Returns a reference to `this*`.

Note: The allocator and key usage specialization is unchanged by this call.

```
enumerable_thread_specific& operator=( enumerable_thread_specific&& other );
```

Moves the content of `other` to `*this` intact. An `other` is left in an unspecified state, but can be safely destroyed. Returns a reference to `this*`.

```
template<typename Alloc, ets_key_usage_type Cachetype>
enumerable_thread_specific& operator=( enumerable_thread_specific<T, Alloc, Cachetype>&&␣
↪other );
```

Moves the content of `other` to `*this` using per-element move construction and keeping thread correspondence. An `other` is left in an unspecified state, but can be safely destroyed. Returns a reference to `this*`.

Note: The allocator and key usage specialization is unchanged by this call.

Concurrently safe modifiers

All member functions in this section can be performed concurrently with each other.

reference **`local`**`()`

> If there is no current element corresponding to the current thread, this method constructs a new element. A new element is copy-constructed if an exemplar was provided to the constructor for `*this`; otherwise, a new element is default-constructed.
>
> **Returns**: A reference to the element of `*this` that corresponds to the current thread.

reference **`local`**`(bool &exists)`

> Similar to `local()`, except that `exists` is set to true if an element was already present for the current thread; false, otherwise.
>
> **Returns**: Reference to the thread-local element.

Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these methods with other (either concurrently safe) methods.

clear

```
void clear();
```

Destroys all elements in `*this`.

Size and capacity

size_type **size**() const

Returns the number of elements in `*this`. The value is equal to the number of distinct threads that have called `local()` after `*this` was constructed or most recently cleared.

bool **empty**() const

Returns `true` if the container is empty; `false`, otherwise.

Iteration

Class template `enumerable_thread_specific` supports random access iterators, which enable iteration over the set of all elements in the container.

iterator **begin**()

Returns `iterator` pointing to the beginning of the set of elements.

iterator **end**()

Returns `iterator` pointing to the end of the set of elements.

const_iterator **begin**() const

Returns `const_iterator` pointing to the beginning of the set of elements.

const_iterator **end**() const

Returns `const_iterator` pointing to the end of the set of elements.

Class template `enumerable_thread_specific` supports `const_range_type` and `range_type` types, which model the *ContainerRange requirement*. The types differ only in that the bounds for a `const_range_type` are of type `const_iterator`, whereas the bounds for a `range_type` are of type `iterator`.

const_range_type **range**(size_t grainsize = 1) const

Returns: A `const_range_type` representing all elements in `*this`. The parameter `grainsize` is in units of elements.

range_type **range**(size_t grainsize = 1)

Returns: A `range_type` representing all elements in `*this`. The parameter `grainsize` is in units of elements.

Combining

The member functions in this section iterate across the entire container sequentially in the calling thread.

template<typename **BinaryFunc**>
T **combine**(*BinaryFunc* f)

> **Requires**: A `BinaryFunc` must meet the *Function Objects* requirements described in the [function.objects] section of the ISO C++ standard. Specifically, the type should be an associative binary functor with the signature `T BinaryFunc(T,T)` or `T BinaryFunc(const T&,const T&)`. A `T` type must be the same as a corresponding template parameter for `enumerable_thread_specific` object.
>
> **Effects**: Computes reduction over all elements using binary functor `f`. If there are no elements, creates the result using the same rules as for creating a thread-local element.
>
> **Returns**: Result of the reduction.

template<typename **UnaryFunc**>
void **combine_each**(*UnaryFunc* f)

> **Requires**: An `UnaryFunc` must meet the *Function Objects* requirements described in the [function.objects] section of the ISO C++ standard. Specifically, the type should be an unary functor with one of signatures: `void UnaryFunc(T)`, `void UnaryFunc(T&)`, or `void UnaryFunc(const T&)` A `T` type must be the same as a corresponding template parameter for the `enumerable_thread_specific` object.
>
> **Effects**: Evaluates `f(x)` for each instance `x` of `T` in `*this`.

Non-member types and constants

enum `ets_key_usage_type::`**ets_key_per_instance**

Enumeration parameter type used to select an implementation that consumes 1 native TLS key per `enumerable_thread_specific` instance. The number of native TLS keys may be limited and can be fairly small.

enum `ets_key_usage_type::`**ets_no_key**

Enumeration parameter type used to select an implementation that consumes no native TLS keys. If no `ETS_key_usage_type` parameter type is provided, `ets_no_key` is used by default.

enum `ets_key_usage_type::`**ets_suspend_aware**

The `oneapi::tbb::task::suspend` function can change the value of the `enumerable_thread_specific` object. To avoid this problem, use the `ets_suspend_aware` enumeration parameter type. The `local()` value can be the same for different threads, but no two distinct threads can access the same value simultaneously.

This section also describes class template `flatten2d`, which assists a common idiom where an `enumerable_thread_specific` represents a container partitioner across threads.

flattened2d

[tls.flattened2d]

The class template `flattened2d` is an adaptor that provides a flattened view of a container of containers.

```
// Defined in header <oneapi/tbb/enumerable_thread_specific.h>

namespace oneapi {
namespace tbb {

    template<typename Container>
    class flattened2d {
    public:
        // Basic types
        using size_type = /* implementation-defined */;
        using difference_type = /* implementation-defined */;
        using allocator_type = /* implementation-defined */;
        using value_type = /* implementation-defined */;
        using reference = /* implementation-defined */;
        using const_reference = /* implementation-defined */;
        using pointer = /* implementation-defined */;
        using const_pointer = /* implementation-defined */;

        using iterator = /* implementation-defined */;
        using const_iterator = /* implementation-defined */;

        explicit flattened2d( const Container& c );

        flattened2d( const Container& c,
                     typename Container::const_iterator first,
                     typename Container::const_iterator last );

        iterator begin();
        iterator end();
        const_iterator begin() const;
        const_iterator end() const;

        size_type size() const;
    };

    template <typename Container>
    flattened2d<Container> flatten2d(const Container &c);

    template <typename Container>
    flattened2d<Container> flatten2d(
        const Container &c,
        const typename Container::const_iterator first,
        const typename Container::const_iterator last);

} // namespace tbb
} // namespace oneapi
```

Requirements:

- A `Container` type must meet the container requirements from the [container.requirements.general] ISO C++ section.

Iterating from `begin()` to `end()` visits all of the elements in the inner containers. The class template supports forward iterators only.

The utility function `flatten2d` creates a `flattened2d` object from a specified container.

Member functions

explicit **`flattened2d`**(const Container &c)

Constructs a `flattened2d` representing the sequence of elements in the inner containers contained by outer container c.

Safety: these operations must not be invoked concurrently on the same `flattened2d`.

`flattened2d`(const Container &c, typename Container::const_iterator first, typename Container::const_iterator last)

Constructs a `flattened2d` representing the sequence of elements in the inner containers in the half-open interval `[first, last)` of a container `c`.

Safety: these operations must not be invoked concurrently on the same `flattened2d`.

size_type **`size`**() const

Returns the sum of the sizes of the inner containers that are viewable in the `flattened2d`.

Safety: These operations may be invoked concurrently on the same `flattened2d`.

iterator **`begin`**()

Returns `iterator` pointing to the beginning of the set of local copies.

iterator **`end`**()

Returns `iterator` pointing to the end of the set of local copies.

const_iterator **`begin`**() const

Returns `const_iterator` pointing to the beginning of the set of local copies.

const_iterator **`end`**() const

Returns `const_iterator` pointing to the end of the set of local copies.

Non-member functions

template<typename **`Container`**>
flattened2d<*Container*> **`flatten2d`**(const *Container* &c, const typename *Container*::const_iterator b, const typename *Container*::const_iterator e)

Constructs and returns a `flattened2d` object that provides iterators that traverse the elements in the containers within the half-open range `[b, e)` of a container `c`.

template<typename **`Container`**>
`flattened2d`(const *Container* &c)

Constructs and returns a `flattened2d` that provides iterators that traverse the elements in all of the containers within a container `c`.

8.3 oneTBB Auxiliary Interfaces

8.3.1 Memory Allocation

[memory_allocation]

This section describes classes and functions related to memory allocation.

Allocators

The oneAPI Threading Building Blocks (oneTBB) library implements several classes that meet the allocator requirements from the [allocator.requirements] ISO C++ Standard section.

tbb_allocator

[memory_allocation.tbb_allocator]

A `tbb_allocator` is a class template that models the allocator requirements from the [allocator.requirements] ISO C++ section.

The `tbb_allocator` allocates and frees memory via the oneTBB malloc library if it is available, otherwise, it reverts to using `std::malloc` and `std::free`.

```
// Defined in header <oneapi/tbb/tbb_allocator.h>

namespace oneapi {
namespace tbb {
    template<typename T> class tbb_allocator {
    public:
        using value_type = T;
        using size_type = std::size_t;
        using propagate_on_container_move_assignment = std::true_type;
        using is_always_equal = std::true_type;

        enum malloc_type {
            scalable,
            standard
        };

        tbb_allocator() = default;
        template<typename U>
        tbb_allocator(const tbb_allocator<U>&) noexcept;

        T* allocate(size_type);
        void deallocate(T*, size_type);

        static malloc_type allocator_type();
    };
} // namespace tbb
} // namespace oneapi
```

Member Functions

T *`allocate`(*size_type* n)

Allocates `n * sizeof(T)` bytes. Returns a pointer to the allocated memory.

void `deallocate`(T *p, *size_type* n)

Deallocates memory pointed to by `p`. The behavior is undefined if the pointer `p` is not the result of the `allocate(n)` method. The behavior is undefined if the memory has been already deallocated.

static malloc_type `allocator_type`()

Returns the enumeration type `malloc_type::scalable` if the oneTBB malloc library is available, and `malloc_type::standard`, otherwise.

Non-member Functions

These functions provide comparison operations between two `tbb_allocator` instances.

```
template<typename T, typename U>
bool operator==(const tbb_allocator<T>&, const tbb_allocator<U>&) noexcept;

template<typename T, typename U>
bool operator!=(const tbb_allocator<T>&, const tbb_allocator<U>&) noexcept;
```

The namespace where these functions are defined is unspecified, as long as they may be used in respective binary operation expressions on `tbb_allocator` objects. For example, an implementation may define the classes and functions in the same unspecified internal namespace and define `oneapi::tbb::tbb_allocator` as a type alias for which the non-member functions are reachable only via argument-dependent lookup.

template<typename **T**, typename **U**>
bool **operator==**(const tbb_allocator<*T*>&, const tbb_allocator<*U*>&) noexcept

Returns **true**.

template<typename **T**, typename **U**>
bool **operator!=**(const tbb_allocator<*T*>&, const tbb_allocator<*U*>&) noexcept

Returns **false**.

scalable_allocator

[memory_allocation.scalable_allocator]

A `scalable_allocator` is a class template that models the allocator requirements from the [allocator.requirements] ISO C++ section.

The `scalable_allocator` allocates and frees memory in a way that scales with the number of processors. Memory allocated by a `scalable_allocator` should be freed by a `scalable_allocator`, not by a `std::allocator`.

```
// Defined in header <oneapi/tbb/scalable_allocator.h>

namespace oneapi {
namespace tbb {
    template<typename T> class scalable_allocator {
    public:
        using value_type = T;
```

(continues on next page)

(continued from previous page)

```
        using size_type = std::size_t;
        using propagate_on_container_move_assignment = std::true_type;
        using is_always_equal = std::true_type;

        scalable_allocator() = default;
        template<typename U>
        scalable_allocator(const scalable_allocator<U>&) noexcept;

        T* allocate(size_type);
        void deallocate(T*, size_type);
    };
} // namespace tbb
} // namespace oneapi
```

Caution: The `scalable_allocator` requires the memory allocator library. If the library is missing, calls to the scalable allocator fail. In contrast to `scalable_allocator`, if the memory allocator library is not available, `tbb_allocator` falls back on `std::malloc` and `std::free`.

Member Functions

value_type ***allocate**(*size_type* n)

Allocates `n * sizeof(T)` bytes of memory. Returns a pointer to the allocated memory.

void **deallocate**(value_type *p, *size_type* n)

Deallocates memory pointed to by `p`. The behavior is undefined if the pointer `p` is not the result of the `allocate(n)` method. The behavior is undefined if the memory has been already deallocated.

Non-member Functions

These functions provide comparison operations between two `scalable_allocator` instances.

```
namespace oneapi {
namespace tbb {
    template<typename T, typename U>
    bool operator==(const scalable_allocator<T>&,
                    const scalable_allocator<U>&) noexcept;

    template<typename T, typename U>
    bool operator!=(const scalable_allocator<T>&,
                    const scalable_allocator<U>&) noexcept;
} // namespace tbb
} // namespace oneapi
```

The namespace where these functions are defined is unspecified, as long as they may be used in respective binary operation expressions on `scalable_allocator` objects. For example, an implementation may define the classes and functions in the same unspecified internal namespace, and define `oneapi::tbb::scalable_allocator` as a type alias for which the non-member functions are reachable only via argument-dependent lookup.

template<typename **T**, typename **U**>

bool **operator==**(const scalable_allocator<*T*>&, const scalable_allocator<*U*>&) noexcept

Returns **true**.

template<typename **T**, typename **U**>
bool **operator!=**(const scalable_allocator<*T*>&, const scalable_allocator<*U*>&) noexcept

Returns **false**.

cache_aligned_allocator

[memory_allocation.cache_aligned_allocator]

A `cache_aligned_allocator` is a class template that models the allocator requirements from the [allocator.requirements] ISO C++ section.

The `cache_aligned_allocator` allocates memory on cache line boundaries, in order to avoid false sharing and potentially improve performance. False sharing is a situation when logically distinct items occupy the same cache line, which can hurt performance if multiple threads attempt to access the different items simultaneously. Even though the items are logically separate, the processor hardware may have to transfer the cache line between the processors as if they were sharing a location. The net result can be much more memory traffic than if the logically distinct items were on different cache lines.

However, this class is sometimes an inappropriate replacement for default allocator, because the benefit of allocating on a cache line comes at the price that `cache_aligned_allocator` implicitly adds pad memory. Therefore allocating many small objects with `cache_aligned_allocator` may increase memory usage.

```
// Defined in header <oneapi/tbb/cache_aligned_allocator.h>

namespace oneapi {
namespace tbb {
    template<typename T> class cache_aligned_allocator {
    public:
        using value_type = T;
        using size_type = std::size_t;
        using propagate_on_container_move_assignment = std::true_type;
        using is_always_equal = std::true_type;

        cache_aligned_allocator() = default;
        template<typename U>
        cache_aligned_allocator(const cache_aligned_allocator<U>&) noexcept;

        T* allocate(size_type);
        void deallocate(T*, size_type);
        size_type max_size() const noexcept;
    };
} // namespace tbb
} // namespace oneapi
```

Member Functions

T *__allocate__(*size_type* n)

Returns a pointer to the allocated `n * sizeof(T)` bytes of memory, aligned on a cache-line boundary. The allocation may include extra hidden padding.

void __deallocate__(T *p, *size_type* n)

Deallocates memory pointed to by `p`. Deallocation also deallocates any extra hidden padding. The behavior is undefined if the pointer `p` is not the result of the `allocate(n)` method. The behavior is undefined if the memory has been already deallocated.

size_type __max_size__() const noexcept

Returns the largest value `n` for which the call `allocate(n)` might succeed with cache alignment constraints.

Non-member Functions

These functions provide comparison operations between two `cache_aligned_allocator` instances.

```
template<typename T, typename U>
bool operator==(const cache_aligned_allocator<T>&,
                const cache_aligned_allocator<U>&) noexcept;

template<typename T, typename U>
bool operator!=(const cache_aligned_allocator<T>&,
                const cache_aligned_allocator<U>&) noexcept;
```

The namespace where these functions are defined is unspecified, as long as they may be used in respective binary operation expressions on `cache_aligned_allocator` objects. For example, an implementation may define the classes and functions in the same unspecified internal namespace, and define `oneapi::tbb::cache_aligned_allocator` as a type alias for which the non-member functions are reachable only via argument-dependent lookup.

template<typename **T**, typename **U**>
bool **operator==**(const cache_aligned_allocator<*T*>&, const cache_aligned_allocator<*U*>&) noexcept

Returns **true**.

template<typename **T**, typename **U**>
bool **operator!=**(const cache_aligned_allocator<*T*>&, const cache_aligned_allocator<*U*>&) noexcept

Returns **false**.

Memory Resources

Starting from C++17, the standard library provides a `std::pmr::polymorphic_allocator` class that allocates memory from a supplied memory resource (see the [mem.poly.allocator.class] ISO/IEC 14882:2017 section). Class `std::pmr::memory_resource` is an abstract interface for user-side implementation of different allocation strategies. For details, see the [mem.res.class] ISO/IEC 14882:2017 standard section.

oneTBB provides a set of `std::pmr::memory_resource` implementations.

cache_aligned_resource

[memory_allocation.cache_aligned_resource]

A `cache_aligned_resource` is a general-purpose memory resource class, which acts as a wrapper to another memory resource to ensure that all allocations are aligned on cache line boundaries to avoid false sharing.

See the *cache_aligned_allocator template class* section for more information about false sharing avoidance.

```
// Defined in header <oneapi/tbb/cache_aligned_allocator.h>

namespace oneapi {
namespace tbb {
    class cache_aligned_resource {
    public:
        cache_aligned_resource();
        explicit cache_aligned_resource( std::pmr::memory_resource* );

        std::pmr::memory_resource* upstream_resource() const;

    private:
        void* do_allocate(size_t n, size_t alignment) override;
        void do_deallocate(void* p, size_t n, size_t alignment) override;
        bool do_is_equal(const std::pmr::memory_resource& other) const noexcept override;
    };
} // namespace tbb
} // namespace oneapi
```

Member Functions

cache_aligned_resource()

Constructs a `cache_aligned_resource` over `std::pmr::get_default_resource()`.

explicit **cache_aligned_resource**(std::pmr::memory_resource *r)

Constructs a `cache_aligned_resource` over the memory resource `r`.

std::pmr::memory_resource ***upstream_resource**() const

Returns the pointer to the underlying memory resource.

void ***do_allocate**(size_t n, size_t alignment) override

Allocates `n` bytes of memory on a cache-line boundary, with alignment not less than requested. The allocation may include extra memory for padding. Returns pointer to the allocated memory.

void **do_deallocate**(void *p, size_t n, size_t alignment) override

Deallocates memory pointed to by `p` and any extra padding. Pointer `p` must be obtained with `do_allocate(n, alignment)`. The memory must not be deallocated beforehand.

bool **do_is_equal**(const std::pmr::memory_resource &other) const noexcept override

Compares upstream memory resources of `*this` and `other`. If `other` is not a `cache_aligned_resource`, returns false.

scalable_memory_resource

[memory_allocation.scalable_memory_resource]

A `oneapi::tbb::scalable_memory_resource()` is a function that returns a memory resource for scalable memory allocation.

The `scalable_memory_resource()` function returns the pointer to the memory resource managed by the oneTBB scalable memory allocator. In particular, its `allocate` method uses `scalable_aligned_malloc()`, and `deallocate` uses `scalable_free()`. The memory resources returned by this function compare equal.

`std::pmr::polymorphic_allocator` instantiated with `oneapi::tbb::scalable_memory_resource()` behaves like `oneapi::tbb::scalable_allocator`.

```
// Defined in header <oneapi/tbb/scalable_allocator.h>

std::pmr::memory_resource* scalable_memory_resource();
```

Library Functions

C Interface to Scalable Allocator

[memory_allocation.scalable_alloc_c_interface]

Low-level interface for scalable memory allocation.

```
// Defined in header <oneapi/tbb/scalable_allocator.h>

extern "C" {
    // Scalable analogs of C memory allocator
    void* scalable_malloc( size_t size );
    void  scalable_free( void* ptr );
    void* scalable_calloc( size_t nobj, size_t size );
    void* scalable_realloc( void* ptr, size_t size );

    // Analog of _msize/malloc_size/malloc_usable_size.
    size_t scalable_msize( void* ptr );

    // Scalable analog of posix_memalign
    int scalable_posix_memalign( void** memptr, size_t alignment, size_t size );

    // Aligned allocation
    void* scalable_aligned_malloc( size_t size, size_t alignment);
    void scalable_aligned_free( void* ptr );
    void* scalable_aligned_realloc( void* ptr, size_t size, size_t alignment );

    // Return values for scalable_allocation_* functions
    typedef enum {
        TBBMALLOC_OK,
        TBBMALLOC_INVALID_PARAM,
        TBBMALLOC_UNSUPPORTED,
        TBBMALLOC_NO_MEMORY,
        TBBMALLOC_NO_EFFECT
    } ScalableAllocationResult;
```

(continues on next page)

(continued from previous page)

```
    typedef enum {
        // To turn on/off the use of huge memory pages
        TBBMALLOC_USE_HUGE_PAGES,
        // To set a threshold for the allocator memory usage.
        // Exceeding it will forcefully clean internal memory buffers
        TBBMALLOC_SET_SOFT_HEAP_LIMIT,
        // Lower bound for the size (Bytes), that is interpreted as huge
        // and not released during regular cleanup operations
        TBBMALLOC_SET_HUGE_SIZE_THRESHOLD
    } AllocationModeParam;

    // Set allocator-specific allocation modes.
    int scalable_allocation_mode(int param, intptr_t value);

    typedef enum {
        // Clean internal allocator buffers for all threads.
        TBBMALLOC_CLEAN_ALL_BUFFERS,
        // Clean internal allocator buffer for current thread only.
        TBBMALLOC_CLEAN_THREAD_BUFFERS
    } ScalableAllocationCmd;

    // Call allocator-specific commands.
    int scalable_allocation_command(int cmd, void *param);
}
```

These functions provide a C-level interface to the scalable allocator. With the exception of `scalable_allocation_mode` and `scalable_allocation_command`, each routine `scalable_x` behaves analogously to the library function `x`. The routines form the two families shown in the table below, "C Interface to Scalable Allocator". Storage allocated by a `scalable_x` function in one family must be freed or resized by the `scalable_x` function in the same family, not by a C standard library function. Likewise, storage allocated by a C standard library function should not be freed or resized by a `scalable_x` function.

Table 5: C Interface to Scalable Allocator

Allocation Routine	Deallocation Routine	Analogous Library
scalable_malloc	scalable_free	C standard library
scalable_calloc		
scalable_realloc		
scalable_posix_memalign		POSIX*
scalable_aligned_malloc	scalable_aligned_free	Microsoft* C run-time library
scalable_aligned_realloc		

The following functions do not allocate or free memory but allow obtaining useful information or influencing behavior of the memory allocator.

size_t **scalable_msize**(void *ptr)

> **Returns:** The usable size of the memory block pointed to by *ptr* if it was allocated by the scalable allocator. Returns zero if *ptr* does not point to such a block.

int **scalable_allocation_mode**(int mode, intptr_t value)

> Use this function to adjust behavior of the scalable memory allocator.

Returns: TBBMALLOC_OK if the operation succeeded, TBBMALLOC_INVALID_PARAM if `mode` is not one of the described below, or if `value` is not valid for the given mode. Other return values are possible, as described below.

---

scalable_allocation_mode Parameters: Parameter, Description

TBBMALLOC_USE_HUGE_PAGES

`scalable_allocation_mode(TBBMALLOC_USE_HUGE_PAGES, 1)` tells the allocator to use huge pages if enabled by the operating system. `scalable_allocation_mode(TBBMALLOC_USE_HUGE_PAGES, 0)` disables it. Setting `TBB_MALLOC_USE_HUGE_PAGES` environment variable to 1 has the same effect as `scalable_allocation_mode(TBBMALLOC_USE_HUGE_PAGES, 1)`. The mode set with `scalable_allocation_mode()` takes priority over the environment variable.

May return: `TBBMALLOC_NO_EFFECT` if huge pages are not supported on the platform.

For now, this allocation mode is only supported for Linux* OS. It works with both explicitly configured and transparent huge pages. For information about enabling and configuring huge pages, refer to OS documentation or ask your system administrator.

TBBMALLOC_SET_SOFT_HEAP_LIMIT

`scalable_allocation_mode(TBBMALLOC_SET_SOFT_HEAP_LIMIT, size)` sets a threshold of `size` bytes on the amount of memory the allocator takes from OS. Exceeding the threshold urges the allocator to release memory from its internal buffers; however it does not prevent from requesting more memory if needed.

TBBMALLOC_SET_HUGE_SIZE_THRESHOLD

`scalable_allocation_mode(TBBMALLOC_SET_HUGE_SIZE_THRESHOLD, size)` sets a lower bound threshold (with no upper limit) of `size` bytes. Any object bigger than this threshold becomes huge and does not participate in internal periodic cleanup logic. However, it does not affect the logic of the `TBBMALLOC_SET_SOFT_HEAP_LIMIT` mode as well as the `TBBMALLOC_CLEAN_ALL_BUFFERS` operation.

Setting `TBB_MALLOC_SET_HUGE_SIZE_THRESHOLD` environment variable to the size value has the same effect, but is limited to the `LONG_MAX` value. The mode set with `scalable_allocation_mode` takes priority over the environment variable.

---

int **scalable_allocation_command**(int cmd, void *reserved)

This function may be used to command the scalable memory allocator to perform an action specified by the first parameter. The second parameter is reserved and must be set to 0.

Returns: `TBBMALLOC_OK` if the operation succeeded, `TBBMALLOC_INVALID_PARAM` if `cmd` is not one of the described below, or if `reserved` is not equal to 0.

---

scalable_allocation_command Parameters: Parameter, Description

TBBMALLOC_CLEAN_ALL_BUFFERS

`scalable_allocation_command(TBBMALLOC_CLEAN_ALL_BUFFERS, 0)` cleans internal memory buffers of the allocator, and possibly reduces memory footprint. It may result in increased time for subsequent memory allocation requests. The command is not designed for frequent use, and careful evaluation of the performance impact is recommended.

May return: `TBBMALLOC_NO_EFFECT` if no buffers were released.

Note: It is not guaranteed that the call will release all unused memory.

TBBMALLOC_CLEAN_THREAD_BUFFERS

`scalable_allocation_command(TBBMALLOC_CLEAN_THREAD_BUFFERS, 0)` cleans internal memory buffers, but only for the calling thread.

May return: `TBBMALLOC_NO_EFFECT` if no buffers were released.

8.3.2 Mutual Exclusion

[mutex]

The library provides a set of mutual exclusion primitives to simplify writing race-free code. A mutex object facilitates protection against data races and provides safe synchronization of data between threads.

Mutex Classes

mutex

[mutex.mutex]

A `mutex` is a class that models *Mutex requirement* using an adaptive approach, it guarantees that the thread that cannot acquire the lock spins before blocking. The `mutex` class satisfies all of the mutex requirements described in the [thread.mutex.requirements] section of the ISO C++ standard. The `mutex` class is not fair or recursive.

```
// Defined in header <oneapi/tbb/mutex.h>

namespace oneapi {
    namespace tbb {
        class mutex {
        public:
            mutex() noexcept;
            ~mutex();

            mutex(const mutex&) = delete;
            mutex& operator=(const mutex&) = delete;

            class scoped_lock;

            void lock();
            bool try_lock();
            void unlock();

            static constexpr bool is_rw_mutex = false;
            static constexpr bool is_recursive_mutex = false;
            static constexpr bool is_fair_mutex = false;
        };
    }
}
```

Member classes

class **scoped_lock**

The corresponding `scoped_lock` class. See *Mutex requirement*.

Member functions

mutex()

Constructs a `mutex` with the unlocked state.

~mutex()

Destroys an unlocked `mutex`.

---

void **lock**()

Acquires a lock. It uses an adaptive logic for waiting, thus it is blocked after a certain time of busy waiting.

---

bool **try_lock**()

Tries to acquire a lock (non-blocking). Returns **true** if succeeded; **false** otherwise.

---

void **unlock**()

Releases the lock held by a current thread.

rw_mutex

[mutex.rw_mutex]

A `rw_mutex` is a class that models *ReaderWriterMutex requirement* using an adaptive approach, it guarantees that the thread that cannot acquire the lock spins before blocking. The `rw_mutex` class satisfies all of the shared mutex requirements described in the [thread.sharedmutex.requirements] section of the ISO C++ standard. The `rw_mutex` class is an unfair reader-writer lock with a writer preference.

```
// Defined in header <oneapi/tbb/rw_mutex.h>

namespace oneapi {
    namespace tbb {
        class rw_mutex {
        public:
            rw_mutex() noexcept;
            ~rw_mutex();

            rw_mutex(const rw_mutex&) = delete;
            rw_mutex& operator=(const rw_mutex&) = delete;

            class scoped_lock;
```

(continues on next page)

(continued from previous page)

```
            // exclusive ownership
            void lock();
            bool try_lock();
            void unlock();

            // shared ownership
            void lock_shared();
            bool try_lock_shared();
            void unlock_shared();

            static constexpr bool is_rw_mutex = true;
            static constexpr bool is_recursive_mutex = false;
            static constexpr bool is_fair_mutex = false;
        };
    }
}
```

Member classes

class **scoped_lock**

The corresponding scoped-lock class. See *ReaderWriterMutex requirement*.

Member functions

rw_mutex()

Constructs an unlocked `rw_mutex`.

~rw_mutex()

Destroys an unlocked `rw_mutex`.

---

void **lock**()

Acquires a lock. It uses an adaptive logic for waiting, thus it is blocked after a certain time of busy waiting.

---

bool **try_lock**()

Tries to acquire a lock (non-blocking) on write. Returns **true** if succeeded; **false** otherwise.

---

void **unlock**()

Releases the write lock held by the current thread.

---

void **lock_shared**()

Acquires a lock on read. It uses an adaptive logic for waiting, thus it is blocked after a certain time of busy waiting.

---

bool **try_lock_shared**()

Tries to acquire the lock (non-blocking) on read. Returns **true** if succeeded; **false** otherwise.

---

void **unlock_shared**()

Releases the read lock held by the current thread.

spin_mutex

[mutex.spin_mutex]

A `spin_mutex` is a class that models the *Mutex requirement* using a spin lock. The `spin_mutex` class satisfies all requirements of mutex type from the [thread.mutex.requirements] ISO C++ section. The `spin_mutex` class is not fair or recursive.

```
// Defined in header <oneapi/tbb/spin_mutex.h>

namespace oneapi {
namespace tbb {
    class spin_mutex {
    public:
        spin_mutex() noexcept;
        ~spin_mutex();

        spin_mutex(const spin_mutex&) = delete;
        spin_mutex& operator=(const spin_mutex&) = delete;

        class scoped_lock;

        void lock();
        bool try_lock();
        void unlock();

        static constexpr bool is_rw_mutex = false;
        static constexpr bool is_recursive_mutex = false;
        static constexpr bool is_fair_mutex = false;
    };
} // namespace tbb
} // namespace oneapi
```

Member classes

class **scoped_lock**

Corresponding `scoped_lock` class. See the *Mutex requirement*.

Member functions

spin_mutex()

Constructs `spin_mutex` with unlocked state.

~spin_mutex()

Destroys an unlocked `spin_mutex`.

void **lock**()

Acquires a lock. Spins if the lock is taken.

bool **try_lock**()

Attempts to acquire a lock (non-blocking). Returns **true** if lock is acquired; **false**, otherwise.

void **unlock**()

Releases a lock held by a current thread.

spin_rw_mutex

[mutex.spin_rw_mutex]

A `spin_rw_mutex` is a class that models the *ReaderWriterMutex requirement* and satisfies all requirements of shared mutex type from the [thread.sharedmutex.requirements] ISO C++ section.

The `spin_rw_mutex` class is unfair spinning reader-writer lock with backoff and writer-preference.

```
// Defined in header <oneapi/tbb/spin_rw_mutex.h>

namespace oneapi {
namespace tbb {
    class spin_rw_mutex {
    public:
        spin_rw_mutex() noexcept;
        ~spin_rw_mutex();

        spin_rw_mutex(const spin_rw_mutex&) = delete;
        spin_rw_mutex& operator=(const spin_rw_mutex&) = delete;

        class scoped_lock;

        // exclusive ownership
        void lock();
        bool try_lock();
        void unlock();

        // shared ownership
        void lock_shared();
        bool try_lock_shared();
```

(continues on next page)

(continued from previous page)

```
        void unlock_shared();

        static constexpr bool is_rw_mutex = true;
        static constexpr bool is_recursive_mutex = false;
        static constexpr bool is_fair_mutex = false;
    };
} // namespace tbb
} // namespace oneapi
```

Member classes

class **scoped_lock**

Corresponding scoped-lock class. See the *ReaderWriterMutex requirement*.

Member functions

spin_rw_mutex()

Constructs unlocked `spin_rw_mutex`.

~spin_rw_mutex()

Destroys unlocked `spin_rw_mutex`.

void **lock**()

Acquires a lock. Spins if the lock is taken.

bool **try_lock**()

Attempts to acquire a lock (non-blocking) on write. Returns true if the lock is acquired on write; false otherwise.

void **unlock**()

Releases a write lock, held by the current thread.

void **lock_shared**()

Acquires a lock on read. Spins if the lock is taken on write already.

bool **try_lock_shared**()

Attempts to acquire the lock (non-blocking) on read. Returns true if the lock is acquired on read; false, otherwise.

void **unlock_shared**()

Releases a read lock held by the current thread.

speculative_spin_mutex

[mutex.speculative_spin_mutex]

A `speculative_spin_mutex` is a class that models the *Mutex requirement* using a spin lock, and for processors that support hardware transactional memory (such as Intel® Transactional Synchronization Extensions (Intel® TSX)) may be implemented in a way that allows non-contending changes to the protected data to proceed in parallel.

The `speculative_spin_mutex` is not fair and not recursive. The `speculative_spin_mutex` is like a `spin_mutex`, but it may provide better throughput than non-speculative mutexes when the following conditions are met:

- Running on a processor that supports hardware transactional memory.

- Multiple threads can concurrently execute the critical section(s) protected by the mutex, mostly without conflicting.

Otherwise, it performs like a `spin_mutex`, possibly with worse throughput.

```
// Defined in header <oneapi/tbb/spin_mutex.h>

namespace oneapi {
namespace tbb {
    class speculative_spin_mutex {
    public:
        speculative_spin_mutex() noexcept;
        ~speculative_spin_mutex();

        speculative_spin_mutex(const speculative_spin_mutex&) = delete;
        speculative_spin_mutex& operator=(const speculative_spin_mutex&) = delete;

        class scoped_lock;

        static constexpr bool is_rw_mutex = false;
        static constexpr bool is_recursive_mutex = false;
        static constexpr bool is_fair_mutex = false;
    };
} // namespace tbb
} // namespace oneapi
```

Member classes

class **scoped_lock**

Corresponding `scoped_lock` class. See the *Mutex requirement*.

Member functions

speculative_spin_mutex()

Constructs `speculative_spin_mutex` with unlocked state.

~speculative_spin_mutex()

Destroys an unlocked `speculative_spin_mutex`.

speculative_spin_rw_mutex

[mutex.speculative_spin_rw_mutex]

A `speculative_spin_rw_mutex` is a class that models the *ReaderWriterMutex requirement*, and for processors that support hardware transactional memory (such as Intel® Transactional Synchronization Extensions (Intel® TSX)) may be implemented in a way that allows non-contending changes to the protected data to proceed in parallel.

The `speculative_spin_rw_mutex` class is not fair and not recursive. The `speculative_spin_rw_mutex` class is like a `spin_rw_mutex`, but it may provide better throughput than non-speculative mutexes when the following conditions are met:

- Running on a processor that supports hardware transactional memory.

- Multiple threads can concurrently execute the critical section(s) protected by the mutex, mostly without conflicting.

Otherwise, it performs like a `spin_rw_mutex`, possibly with worse throughput.

For processors that support hardware transactional memory, `speculative_spin_rw_mutex` may be implemented in a way that

- speculative readers and writers do not block each other
- a non-speculative reader blocks writers but allows speculative readers
- a non-speculative writer blocks all readers and writers

```
// Defined in header <oneapi/tbb/spin_rw_mutex.h>

namespace oneapi {
namespace tbb {
    class speculative_spin_rw_mutex {
    public:
        speculative_spin_rw_mutex() noexcept;
        ~speculative_spin_rw_mutex();

        speculative_spin_rw_mutex(const speculative_spin_rw_mutex&) = delete;
        speculative_spin_rw_mutex& operator=(const speculative_spin_rw_mutex&) = delete;

        class scoped_lock;

        static constexpr bool is_rw_mutex = true;
        static constexpr bool is_recursive_mutex = false;
        static constexpr bool is_fair_mutex = false;
    };
} // namespace tbb
} // namespace oneapi
```

Member classes

class **scoped_lock**

Corresponding `scoped_lock` class. See the *ReaderWriterMutex requirement*.

Member functions

speculative_spin_rw_mutex()

Constructs `speculative_spin_rw_mutex` with unlocked state.

~speculative_spin_rw_mutex()

Destroys an unlocked `speculative_spin_rw_mutex`.

queuing_mutex

[mutex.queuing_mutex]

A `queuing_mutex` is a class that models the *Mutex requirement*. The `queuing_mutex` is not recursive. The `queuing_mutex` is fair, threads acquire a lock on a mutex in the order that they request it.

```
// Defined in header <oneapi/tbb/queuing_mutex.h>

namespace oneapi {
namespace tbb {
    class queuing_mutex {
    public:
        queuing_mutex() noexcept;
        ~queuing_mutex();

        queuing_mutex(const queuing_mutex&) = delete;
        queuing_mutex& operator=(const queuing_mutex&) = delete;

        class scoped_lock;

        static constexpr bool is_rw_mutex = false;
        static constexpr bool is_recursive_mutex = false;
        static constexpr bool is_fair_mutex = true;
    };
} //  namespace tbb
} //  namespace oneapi
```

Member classes

class **scoped_lock**

Corresponding `scoped_lock` class. See the *Mutex requirement*.

Member functions

queuing_mutex()

Constructs unlocked `queuing_mutex`.

~queuing_mutex()

Destroys unlocked `queuing_mutex`.

queuing_rw_mutex

[mutex.queuing_rw_mutex]

A `queuing_rw_mutex` is a class that models the *ReaderWriterMutex requirement* concept. The `queuing_rw_mutex` is not recursive. The `queuing_rw_mutex` is fair, threads acquire a lock on a mutex in the order that they request it.

```
// Defined in header <oneapi/tbb/queuing_rw_mutex.h>

namespace oneapi {
```

(continues on next page)

(continued from previous page)

```
namespace tbb {
    class queuing_rw_mutex {
    public:
        queuing_rw_mutex() noexcept;
        ~queuing_rw_mutex();

        queuing_rw_mutex(const queuing_rw_mutex&) = delete;
        queuing_rw_mutex& operator=(const queuing_rw_mutex&) = delete;

        class scoped_lock;

        static constexpr bool is_rw_mutex = true;
        static constexpr bool is_recursive_mutex = false;
        static constexpr bool is_fair_mutex = true;
    };
} // namespace tbb
} // namespace oneapi
```

Member classes

class **scoped_lock**

Corresponding `scoped_lock` class. See the *ReaderWriterMutex requirement*.

Member functions

queuing_rw_mutex()

Constructs unlocked `queuing_rw_mutex`.

~queuing_rw_mutex()

Destroys unlocked `queuing_rw_mutex`.

null_mutex

[mutex.null_mutex]

A `null_mutex` is a class that models the *Mutex requirement* concept syntactically, but does nothing. It is useful for instantiating a template that expects a Mutex, but no mutual exclusion is actually needed for that instance.

```
// Defined in header <oneapi/tbb/null_mutex.h>

namespace oneapi {
namespace tbb {
    class null_mutex {
    public:
        constexpr null_mutex() noexcept;
        ~null_mutex();

        null_mutex(const null_mutex&) = delete;
        null_mutex& operator=(const null_mutex&) = delete;
```

(continues on next page)

(continued from previous page)

```
        class scoped_lock;

        void lock();
        bool try_lock();
        void unlock();

        static constexpr bool is_rw_mutex = false;
        static constexpr bool is_recursive_mutex = true;
        static constexpr bool is_fair_mutex = true;
    };
} // namespace tbb
} // namespace oneapi
```

Member classes

class **scoped_lock**

Corresponding `scoped_lock` class. See the *Mutex requirement*.

Member functions

null_mutex()

Constructs unlocked mutex.

~null_mutex()

Destroys unlocked mutex.

void **lock**()

Acquires lock.

bool **try_lock**()

Tries acquiring lock (non-blocking).

void **unlock**()

Releases the lock.

null_rw_mutex

[mutex.null_rw_mutex]

A `null_rw_mutex` is a class that models the *ReaderWriterMutex requirement* syntactically, but does nothing. The `null_rw_mutex` class also satisfies all syntactic requirements of shared mutex type from the [thread.sharedmutex.requirements] ISO C++ section, but does nothing. It is useful for instantiating a template that expects a ReaderWriterMutex, but no mutual exclusion is actually needed for that instance.

```
// Defined in header <oneapi/tbb/null_rw_mutex.h>

namespace oneapi {
namespace tbb {
    class null_rw_mutex {
```

(continues on next page)

(continued from previous page)

```
    public:
        constexpr null_rw_mutex() noexcept;
        ~null_rw_mutex();

        null_rw_mutex(const null_rw_mutex&) = delete;
        null_rw_mutex& operator=(const null_rw_mutex&) = delete;

        class scoped_lock;

        void lock();
        bool try_lock();
        void unlock();

        void lock_shared();
        bool try_lock_shared();
        void unlock_shared();

        static constexpr bool is_rw_mutex = true;
        static constexpr bool is_recursive_mutex = true;
        static constexpr bool is_fair_mutex = true;
    };
} // namespace tbb
} // namespace oneapi
```

Member classes

class **scoped_lock**

Corresponding `scoped_lock` class. See the *ReaderWriterMutex requirement*.

Member functions

null_rw_mutex()

Constructs unlocked mutex.

~null_rw_mutex()

Destroys unlocked mutex.

void **lock**()

Acquires a lock.

bool **try_lock**()

Attempts to acquire a lock (non-blocking) on write. Returns **true**.

void **unlock**()

Releases a write lock held by the current thread.

void **lock_shared**()

Acquires a lock on read.

bool **try_lock_shared**()

Attempts to acquire the lock (non-blocking) on read. Returns **true**.

void **unlock_shared**()

Releases a read lock held by the current thread.

8.3.3 Timing

[timing]

Parallel programming is about speeding up *wall clock* time, which is the real time that it takes a program or function to run. The library provides API to simplify timing within an application.

Syntax

```
// Declared in tick_count.h

class tick_count;

class tick_count::interval_t;
```

Classes

tick_count class

[timing.tick_count]

A `tick_count` is an absolute wall clock timestamp. Two `tick_count` objects can be subtracted to compute wall clock duration `tick_count::interval_t`, which can be converted to seconds.

```
namespace oneapi {
namespace tbb {

    class tick_count {
    public:
        class interval_t;
        tick_count();
        tick_count( const tick_count& );
        ~tick_count();
        tick_count& operator=( const tick_count& );
        static tick_count now();
        static double resolution();
    };

} // namespace tbb
} // namespace oneapi
```

tick_count()
Constructs `tick_count` with an unspecified wall clock timestamp.

tick_count(const tick_count&)
Constructs `tick_count` with the timestamp of the given `tick_count`.

~tick_count()
Destructor.

tick_count& operator=(const tick_count&)
Assigns the timestamp of one `tick_count` to another.

static tick_count now()
Returns a `tick_count` object that represents the current wall clock timestamp.

static double resolution()
Returns the resolution of the clock used by `tick_count`, in seconds.

tick_count::interval_t class

[timing.tick_count.interval_t]

A `tick_count::interval_t` represents wall clock duration.

```
namespace oneapi {
namespace tbb {

    class tick_count::interval_t {
    public:
        interval_t();
        explicit interval_t( double );
        ~interval_t();
        interval_t& operator=( const interval_t& );
        interval_t& operator+=( const interval_t& );
        interval_t& operator-=( const interval_t& );
        double seconds() const;
    };

} // namespace tbb
} // namespace oneapi
```

interval_t()
Constructs `interval_t` representing zero time duration.

explicit interval_t(double)
Constructs `interval_t` representing the specified number of seconds.

~interval_t()
Destructor.

interval_t& operator=(const interval_t&)
Assigns the wall clock duration of one `interval_t` to another.

interval_t& operator+=(const interval_t&)
Increases the duration to the given `interval_t`, and returns `*this`.

interval_t& operator-=(const interval_t&)
Decreases the duration to the given `interval_t`, and returns `*this`.

double seconds() const
Returns the duration measured in seconds.

Non-member functions

[timing.tick_count.nonmember]

These functions provide arithmetic binary operations with wall clock timestamps and durations.

```
oneapi::tbb::tick_count::interval_t operator-( const oneapi::tbb::tick_count&, const␣
↪oneapi::tbb::tick_count& );
oneapi::tbb::tick_count::interval_t operator+( const oneapi::tbb::tick_count::interval_t&
↪, const oneapi::tbb::tick_count::interval_t& );
oneapi::tbb::tick_count::interval_t operator-( const oneapi::tbb::tick_count::interval_t&
↪, const oneapi::tbb::tick_count::interval_t& );
```

The namespace where these functions are defined is unspecified as long as they may be used in respective binary operation expressions on `tick_count` and `tick_count::interval_t` objects. For example, an implementation may define the classes and functions in the same unspecified internal namespace, and define `oneapi::tbb::tick_count` as a type alias for which the non-member functions are reachable only via argument-dependent lookup.

oneapi::tbb::tick_count::interval_t operator-(const oneapi::tbb::tick_count&, const oneapi::tbb::tick_count&)
: Returns `interval_t` representing the duration between two given wall clock timestamps.

oneapi::tbb::tick_count::interval_t operator+(const oneapi::tbb::tick_count::interval_t&, const oneapi::tbb::tick_count::interval_t&)
: Returns `interval_t` representing the sum of two given intervals.

oneapi::tbb::tick_count::interval_t operator-(const oneapi::tbb::tick_count::interval_t&, const oneapi::tbb::tick_count::interval_t&)
: Returns `interval_t` representing the difference of two given intervals.

8.3.4 info Namespace

[info_namespace]

Interfaces to query information about execution environment.

```
// Declared in header <oneapi/tbb/info.h>

namespace oneapi {
namespace tbb {
    using numa_node_id = /*implementation-defined*/;
    using core_type_id = /*implementation-defined*/;

    namespace info {
        std::vector<numa_node_id> numa_nodes();
        std::vector<core_type_id> core_types();

        int default_concurrency(task_arena::constraints c);
        int default_concurrency(numa_node_id id = oneapi::tbb::task_arena::automatic);
    }
} // namespace tbb
} // namespace oneapi
```

Types

`numa_node_id` - Represents NUMA node identifier.

Functions

std::vector<numa_node_id> **numa_nodes**()

Returns the vector of integral indexes that indicate available NUMA nodes.

Note: If error occurs during system topology parsing, returns vector containing single element that equals to `task_arena::automatic`.

---

std::vector<core_type_id> **core_types**()

Returns the vector of integral indexes that indicate available core types. The indexes are sorted from the least performant to the most performant core type.

Note: If error occurs during system topology parsing, returns vector containing single element that equals to `task_arena::automatic`.

---

int **default_concurrency**(*task_arena*::constraints c)

Returns concurrency level for the given constraints.

int **default_concurrency**(numa_node_id id = oneapi::tbb::task_arena::automatic)

Returns concurrency level of the given NUMA node. If argument is not specified, returns default concurrency level for current library configuration.

8.4 oneTBB Deprecated Interfaces

8.4.1 task_arena::attach

[deprecated.task_arena_attach_tag]

> **Caution:** Deprecated in oneTBB Specification 1.1.

A set of methods for constructing a `task_arena` with `attach`.

```
// Defined in header <oneapi/tbb/task_arena.h>

namespace oneapi {
    namespace tbb {

        class task_arena {
        public:
            // ...
            struct attach {};

            explicit task_arena(task_arena::attach);
```

(continues on next page)

(continued from previous page)

```
            void initialize(task_arena::attach);
            // ...
        };

    } // namespace tbb
} // namespace oneapi
```

Member types and constants

struct **attach**

A tag for constructing a `task_arena` with `attach`.

Member functions

explicit **task_arena**(*task_arena*::attach)

Creates an instance of `task_arena` that is connected to the internal task arena representation currently used by the calling thread. If no such arena exists yet, creates a `task_arena` with default parameters.

Note: Unlike other `task_arena` constructors, this one automatically initializes the new `task_arena` when connecting to an already existing arena.

---

void **initialize**(*task_arena*::attach)

If an internal task arena representation currently used by the calling thread, the method ignores arena parameters and connects `task_arena` to that internal task arena representation. The method has no effect when called for an already initialized `task_arena`.

See also:

- *attach*

CHAPTER

NINE

ONEMKL

The oneAPI Math Kernel Library (oneMKL) defines a set of fundamental mathematical routines for use in high-performance computing and other applications. As part of oneAPI, oneMKL is designed to allow execution on a wide variety of computational devices: CPUs, GPUs, FPGAs, and other accelerators. The functionality is subdivided into several domains: dense linear algebra, sparse linear algebra, discrete Fourier transforms, random number generators and vector math.

The general assumptions, design features and requirements for the oneMKL library and host-to-device computational routines will be described in *oneMKL Architecture*. The individual domains and their APIs are described in *oneMKL Domains*. Other design considerations that are not necessarily part of the oneMKL specification but that are worth mentioning will be discussed in *oneMKL Appendix*.

9.1 oneMKL Architecture

The oneMKL element of oneAPI has several general assumptions, requirements and recommendations for all domains contained therein. These will be addressed in this architecture section. In particular, DPC++ allows for a great control over the execution of kernels on the various devices. We discuss the supported execution models of oneMKL APIs in *Execution Model*. A discussion of how data is stored and passed in and out of the APIs is addressed in *Memory Model*. The general structure and design of oneMKL APIs including namespaces and common data types are expressed in *API Design*. The exceptions and error handling are described in *Exceptions and Error Handling*. Finally all the other necessary aspects related to oneMKL architecture can be found in *Other Features* including versioning and discussion of pre and post conditions. Other nonessential, but useful aspects of the oneMKL architecture and design may also be found in the *oneMKL Appendix*.

9.1.1 Execution Model

This section describes the execution environment common to all oneMKL functionality. The execution environment includes how data is provided to computational routines in *Use of Queues*, support for several devices in *Device Usage*, synchronous and asynchronous execution models in *Asynchronous Execution* and *Host Thread Safety*.

Use of Queues

The `sycl::queue` defined in the oneAPI DPC++ specification is used to specify the device and features enabled on that device on which a task will be enqueued. There are two forms of computational routines in oneMKL: class based *Member Functions* and standalone *Non-Member Functions*. As these may interact with the `sycl::queue` in different ways, we provide a section for each one to describe assumptions.

Non-Member Functions

Each oneMKL non-member computational routine takes a `sycl::queue` reference as its first parameter:

```
mkl::domain::routine(sycl::queue &q, ...);
```

All computation performed by the routine shall be done on the hardware device(s) associated with this queue, with possible aid from the host, unless otherwise specified. In the case of an ordered queue, all computation shall also be ordered with respect to other kernels as if enqueued on that queue.

A particular oneMKL implementation may not support the execution of a given oneMKL routine on the specified device(s). In this case, the implementation may either perform the computation on the host or throw an exception. See *Exceptions and Error Handling* for the possible exceptions.

Member Functions

oneMKL class-based APIs, such as those in the RNG and DFT domains, require a `sycl::queue` as an argument to the constructor or another setup routine. The execution requirements for computational routines from the previous section also apply to computational class methods.

Device Usage

oneMKL itself does not currently provide any interfaces for controlling device usage: for instance, controlling the number of cores used on the CPU, or the number of execution units on a GPU. However, such functionality may be available by partitioning a `sycl::device` instance into subdevices, when supported by the device.

When given a queue associated with such a subdevice, a oneMKL implementation shall only perform computation on that subdevice.

Asynchronous Execution

The oneMKL API is designed to allow asynchronous execution of computational routines, to facilitate concurrent usage of multiple devices in the system. Each computational routine enqueues work to be performed on the selected device, and may (but is not required to) return before execution completes.

Hence, it is the calling application's responsibility to ensure that any inputs are valid until computation is complete, and likewise to wait for computation completion before reading any outputs. This can be done automatically when using DPC++ buffers, or manually when using Unified Shared Memory (USM) pointers, as described in the sections below.

Unless otherwise specified, asynchronous execution is *allowed*, but not *guaranteed*, by any oneMKL computational routine, and may vary between implementations and/or versions. oneMKL implementations must clearly document whether execution is guaranteed to be asynchronous for each supported routine. Regardless, calling applications shall not launch any oneMKL computational routine with a dependency on a future oneMKL API call, even if this computational routine executes asynchronously (i.e. a oneMKL implementation may assume no antidependencies are present). This guarantee allows oneMKL implementations to reserve resources for execution without risking deadlock.

Synchronization When Using Buffers

`sycl::buffer` objects automatically manage synchronization between kernel launches linked by a data dependency (either read-after-write, write-after-write, or write-after-read).

oneMKL routines are not required to perform any additional synchronization of `sycl::buffer` arguments.

Synchronization When Using USM APIs

When USM pointers are used as input to, or output from, a oneMKL routine, it becomes the calling application's responsibility to manage possible asynchronicity.

To help the calling application, all oneMKL routines with at least one USM pointer argument also take an optional reference to a list of *input events*, of type `std::vector<sycl::event>`, and have a return value of type `sycl::event` representing computation completion:

```
sycl::event mkl::domain::routine(..., std::vector<sycl::event> &in_events = {});
```

The routine shall ensure that all input events (if the list is present and non-empty) have occurred before any USM pointers are accessed. Likewise, the routine's output event shall not be complete until the routine has finished accessing all USM pointer arguments.

For class methods, "argument" includes any USM pointers previously provided to the object via the class constructor or other class methods.

Host Thread Safety

All oneMKL member and non-member functions shall be *host thread safe*. That is, they may be safely called simultaneously from concurrent host threads. However, oneMKL objects in class-based APIs may not be shared between concurrent host threads unless otherwise specified.

9.1.2 Memory Model

The oneMKL memory model shall follow directly from the oneAPI memory model. Mainly, oneMKL shall support two modes of encapsulating data for consumption on the device: the buffer memory abstraction model and the pointer-based memory model using Unified Shared Memory (USM). These two paradigms shall also support both synchronous and asynchronous execution models as described in *Asynchronous Execution*.

The Buffer Memory Model

The SYCL 1.2.1 specification defines the buffer container templated on the provided data type which encapsulates the data in a SYCL application across both host and devices. It provides the concept of accessors as the mechanism to access the buffer data with different modes to read and or write into that data. These accessors allow SYCL to create and manage the data dependencies in the SYCL graph that order the kernel executions. With the buffer model, all data movement is handled by the SYCL runtime supporting both synchronous and asynchronous execution.

oneMKL provides APIs where buffers (in particular 1D buffers, `sycl::buffer<T,1>`) contain the memory for all non scalar input and output data arguments. See *Synchronization When Using Buffers* for details on how oneMKL routines manage any data dependencies with buffer arguments. Any higher dimensional buffer must be converted to a 1D buffer prior to use in oneMKL APIs, e.g., via `buffer::reinterpret`.

Unified Shared Memory Model

While the buffer model is powerful and elegantly expresses data dependencies, it can be a burden for programmers to replace all pointers and arrays by buffers in their C++ applications. DPC++ also provides pointer-based addressing for device-accessible data, using the Unified Shared Memory (USM) model. Correspondingly, oneMKL provides USM APIs in which non-scalar input and output data arguments are passed by USM pointer.

USM devices and system configurations vary in their ability to share data between devices and between a device and the host. oneMKL implementations may only assume that user-provided USM pointers are accessible by the device associated with the user-provided queue. In particular, an implementation must not assume that USM pointers can be accessed by any other device, or by the host, without querying the DPC++ runtime. An implementation must accept any device-accessible USM pointer regardless of how it was created (*sycl::malloc_device*, *sycl::malloc_shared*, etc.).

Unlike buffers, USM pointers cannot automatically manage data dependencies between kernels. Users may use in-order queues to ensure ordered execution, or explicitly manage dependencies with `sycl::event` objects. To support the second use case, oneMKL USM APIs accept input events (prerequisites before computation can begin) and return an output event (indicating computation is complete). See *Synchronization When Using USM APIs* for details.

9.1.3 API Design

This section discusses the general features of oneMKL API design. In particular, it covers the use of namespaces and data types from C++, from DPC++ and new ones introduced for oneMKL APIs.

oneMKL namespaces

The oneMKL library uses C++ namespaces to organize routines by mathematical domain. All oneMKL objects and routines shall be contained within the `oneapi::mkl` base namespace. The individual oneMKL domains use a secondary namespace layer as follows:

namespace	oneMKL domain or content
`oneapi::mk`	oneMKL base namespace, contains general oneMKL data types, objects, exceptions and routines
`oneapi::mk`	Dense linear algebra routines from BLAS and BLAS like extensions. The oneapi::mkl::blas namespace should contain two namespaces column_major and row_major to support both matrix layouts. See *BLAS Routines*
`oneapi::mk`	Dense linear algebra routines from LAPACK and LAPACK like extensions. See *LAPACK Routines*
`oneapi::mk`	Sparse linear algebra routines from Sparse BLAS and Sparse Solvers. See *Sparse Linear Algebra*
`oneapi::mk`	Discrete Fourier Transforms. See *Discrete Fourier Transform Functions*
`oneapi::mk`	Random number generator routines. See *Random Number Generators*
`oneapi::mk`	Vector mathematics routines, e.g. trigonometric, exponential functions acting on elements of a vector. See *Vector Math*

Note: Inside each oneMKL domain, there are many routines, classes, enums and objects defined which constitute the breadth and scope of that oneMKL domain. It is permitted for a library implementation of the oneMKL specification to implement either all, one or more than one of the domains in oneMKL. However, within an implementation of a specific domain, all relevant routines, classes, enums and objects (including those relevant enums and objects which live outside a particular domain in the general `oneapi::mkl` namespace must be both declared and defined in the library so that an application that uses that domain could build and link against that library implementation successfully.

It is however acceptable to throw the runtime exception *oneapi::mkl::unimplemented* inside of the routines or class member functions in that domain that have not been fully implemented. For instance, a library may choose to implement the oneMKL BLAS functionality and in particular may choose to implement only the *gemm* api for their library, in which case they must also include all the other blas namespaced routines and throw the *oneapi::mkl::unimplemented* exception inside all the others.

In such a case, the implemented routines in such a library should be communicated clearly and easily understood by users of that library.

Standard C++ datatype usage

oneMKL uses C++ STL data types for scalars where applicable:

- Integer scalars are C++ fixed-size integer types (`std::intN_t`, `std::uintN_t`).
- Complex numbers are represented by C++ `std::complex` types.

In general, scalar integer arguments to oneMKL routines are 64-bit integers (`std::int64_t` or `std::uint64_t`). Integer vectors and matrices may have varying bit widths, defined on a per-routine basis.

DPC++ datatype usage

oneMKL uses the following DPC++ data types:

- SYCL queue `sycl::queue` for scheduling kernels on a SYCL device. See *Use of Queues* for more details.
- SYCL buffer `sycl::buffer` for buffer-based memory access. See *The Buffer Memory Model* for more details.
- Unified Shared Memory (USM) for pointer-based memory access. See *Unified Shared Memory Model* for more details.
- SYCL event `sycl::event` for output event synchronization in oneMKL routines with USM pointers. See *Synchronization When Using USM APIs* for more details.
- Vector of SYCL events `std::vector<sycl::event>` for input events synchronization in oneMKL routines with USM pointers. See *Synchronization When Using USM APIs* for more details.

Note: The class `sycl::vector_class` has been removed from SYCL 2020 and the standard class `std::vector` should be used instead for vector of SYCL events in oneMKL routines with USM pointers

oneMKL defined datatypes

oneMKL dense and sparse linear algebra routines use scoped enum types as type-safe replacements for the traditional character arguments used in C/Fortran implementations of BLAS and LAPACK. These types all belong to the `oneapi::mkl` namespace.

Each enumeration value comes with two names: A single-character name (the traditional BLAS/LAPACK character) and a longer, more descriptive name. The two names are exactly equivalent and may be used interchangeably.

transpose

The `transpose` type specifies whether an input matrix should be transposed and/or conjugated. It can take the following values:

Short Name	Long Name	Description
`transpose`	`transpose::nor`	Do not transpose or conjugate the matrix.
`transpose`	`transpose::tra`	Transpose the matrix (without complex conjugation).
`transpose`	`transpose::cor`	Perform Hermitian transpose (transpose and conjugate). Is the same as `transpose::trans` for real matrices.

uplo

The `uplo` type specifies whether the lower or upper triangle of a triangular, symmetric, or Hermitian matrix should be accessed. It can take the following values:

Short Name	Long Name	Description
`uplo::U`	`uplo::upper`	Access the upper triangle of the matrix.
`uplo::L`	`uplo::lower`	Access the lower triangle of the matrix.

In both cases, elements that are not in the selected triangle are not accessed or updated.

diag

The `diag` type specifies the values on the diagonal of a triangular matrix. It can take the following values:

Short Name	Long Name	Description
`diag::N`	`diag::nonu`	The matrix is not unit triangular. The diagonal entries are stored with the matrix data.
`diag::U`	`diag::uni`	The matrix is unit triangular (the diagonal entries are all 1's). The diagonal entries in the matrix data are not accessed.

side

The `side` type specifies the order of matrix multiplication when one matrix has a special form (triangular, symmetric, or Hermitian):

Short Name	Long Name	Description
`side::L`	`side::left`	The special form matrix is on the left in the multiplication.
`side::R`	`side::right`	The special form matrix is on the right in the multiplication.

offset

The `offset` type specifies whether the offset to apply to an output matrix is a fix offset, column offset or row offset. It can take the following values

Short Name	Long Name	Description
`offset`	`offset::`	The offset to apply to the output matrix is fix, all the inputs in the `C_offset` matrix has the same value given by the first element in the `co` array.
`offset`	`offset::`	The offset to apply to the output matrix is a column offset, that is to say all the columns in the `C_offset` matrix are the same and given by the elements in the `co` array.
`offset`	`offset::`	The offset to apply to the output matrix is a row offset, that is to say all the rows in the `C_offset` matrix are the same and given by the elements in the `co` array.

index_base

The `index_base` type specifies how values in index arrays are interpreted. For instance, a sparse matrix stores nonzero values and the indices that they correspond to. The indices are traditionally provided in one of two forms: C/C++-style using zero-based indices, or Fortran-style using one-based indices. The `index_base` type can take the following values:

Name	Description
`index_base::z`	Index arrays for an input matrix are provided using zero-based (C/C++ style) index values. That is, indices start at 0.
`index_base::o`	Index arrays for an input matrix are provided using one-based (Fortran style) index values. That is, indices start at 1.

layout

The `layout` type specifies how a dense matrix `A` with leading dimension `lda` is stored as one dimensional array in memory. The layouts are traditionally provided in one of two forms: C/C++-style using `row_major` layout, or Fortran-style using `column_major` layout. The `layout` type can take the following values:

Short Name	Long Name	Description
`layou`	`layout::`	For row major layout, the elements of each row of a dense matrix `A` are contiguous in memory while the elements of each column are at distance `lda` from the element in the same column and the previous row.
`layou`	`layout::`	For column major layout, the elements of each column a dense matrix `A` are contiguous in memory while the elements of each row are at distance `lda` from the element in the same row and the previous column.

Note: *oneMKL Appendix* may contain other API design decisions or recommendations that may be of use to the general developer of oneMKL, but which may not necessarily be part of the oneMKL specification.

9.1.4 Exceptions and Error Handling

oneMKL error handling relies on the mechanism of C++ exceptions. Should error occur, it will be propagated at the point of a function call where it is caught using standard C++ error handling mechanism.

Exception classification

Exception classification in oneMKL is aligned with C++ Standard Library classification. oneMKL introduces class that defines the base class in the hierarchy of oneMKL exception classes. All oneMKL routines throw exceptions inherited from this base class. In the hierarchy of oneMKL exceptions, `oneapi::mkl::exception` is the base class inherited from `std::exception` class. All other oneMKL exception classes are derived from this base class.

This specification does not require implementations to perform error-checking. However, if an implementation does provide error-checking, it shall use the following exception classes. Additional implementation-specific exception classes can be used for exceptional conditions not fitting any of these classes.

Common exceptions

Exception class	Description
`oneapi::mkl::exception`	Reports general unspecified problem
`oneapi::mkl::unsupported_device`	Reports a problem when the routine is not supported on a specific device
`oneapi::mkl::host_bad_alloc`	Reports a problem that occurred during memory allocation on the host
`oneapi::mkl::device_bad_alloc`	Reports a problem that occurred during memory allocation on a specific device
`oneapi::mkl::unimplemented`	Reports a problem when a specific routine has not been implemented for the specified parameters
`oneapi::mkl::invalid_argument`	Reports problem when arguments to the routine were rejected
`oneapi::mkl::uninitialized`	Reports problem when a handle (descriptor) has not been initialized
`oneapi::mkl::computation_error`	Reports any computation errors that have occurred inside a oneMKL routine
`oneapi::mkl::batch_error`	Reports errors that have occurred inside a batch oneMKL routine

LAPACK specific exceptions

Exception class	Description
`oneapi::mkl::lapack::exception`	Base class for all LAPACK exceptions providing access to info code familiar to users of conventional LAPACK API. All LAPACK related exceptions can be handled with catch block for this class.
`oneapi::mkl::lapack::invalid_argume`	Reports errors when arguments provided to the LAPACK subroutine are inconsistent or do not match expected values. Class extends base `oneapi::mkl::invalid_argument` with ability to access conventional status info code.
`oneapi::mkl::lapack::computation_er`	Reports computation errors that have occurred during call to LAPACK subroutine. Class extends base `oneapi::mkl::computation_error` with ability to access conventional status info code familiar to LAPACK users.
`oneapi::mkl::lapack::batch_error`	Reports errors that have occurred during batch LAPACK computations. Class extends base `oneapi::mkl::batch_error` with ability to access individual exception objects for each of the issues observed in a batch and an info code. The info code contains the number of errors that occurred in a batch. Positions of problems in a supplied batch that experienced issues during computations can be retrieved with *ids()* method, and list of particular exceptions can be obtained with *exceptions()* method of the exception object. Possible exceptions for a batch are documented for corresponding non-batch API.

9.1.5 Other Features

This section covers all other features in the design of oneMKL architecture.

Current Version of this oneMKL Specification

This is the oneMKL specification which is part of the oneAPI specification version 1.0.0.

Pre/Post Condition Checking

The individual oneMKL computational routines will define any preconditions and postconditions and will define in this specification any specific checks or verifications that should be enabled for all implementations.

9.2 oneMKL Domains

This section describes the Data Parallel C++ (DPC++) interface.

9.2.1 Dense Linear Algebra

This section contains information about dense linear algebra routines:

Matrix Storage provides information about dense matrix and vector storage formats that are used by oneMKL *BLAS Routines* and *LAPACK Routines*.

BLAS Routines provides vector, matrix-vector, and matrix-matrix routines for dense matrices and vector operations.

Scalar Arguments in BLAS describes some details of how scalar parameters (such as `alpha` and `beta`) are handled so that users may pass either values or pointers for these parameters.

LAPACK Routines provides more complex dense linear algebra routines, e.g., matrix factorization, solving dense systems of linear equations, least square problems, eigenvalue and singular value problems, and performing a number of related computational tasks.

Matrix Storage

The oneMKL BLAS and LAPACK routines for DPC++ use several matrix and vector storage formats. These are the same formats used in traditional Fortran BLAS/LAPACK. LAPACK routines require column major layout.

General Matrix

A general matrix `A` of `m` rows and `n` columns with leading dimension `lda` is represented as a one dimensional array `a` of size of at least `lda` * `n` if column major layout is used and at least `lda` * `m` if row major layout is used. Before entry in any BLAS function using a general matrix, the leading `m` by `n` part of the array `a` must contain the matrix `A`. For column (respectively row) major layout, the elements of each column (respectively row) are contiguous in memory while the elements of each row (respectively column) are at distance `lda` from the element in the same row (respectively column) and the previous column (respectively row).

Visually, the matrix

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & \dots & A_{1n} \\ A_{21} & A_{22} & A_{23} & \dots & A_{2n} \\ A_{31} & A_{32} & A_{33} & \dots & A_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{m1} & A_{m2} & A_{m3} & \dots & A_{mn} \end{bmatrix}$$

is stored in memory as an array

- For column major layout,

$$a=[\underbrace{\underbrace{A_{11}, A_{21}, A_{31}, ..., A_{m1}, *, ..., *}_{\text{lda}}, \underbrace{A_{12}, A_{22}, A_{32}, ..., A_{m2}, *, ..., *}_{\text{lda}}, ..., \underbrace{A_{1n}, A_{2n}, A_{3n}, ..., A_{mn}, *, ..., *}_{\text{lda}}}_{\text{lda x n}}]$$

- For row major layout,

$$a=[\underbrace{\underbrace{A_{11}, A_{12}, A_{13}, ..., A_{1n}, *, ..., *}_{\text{lda}}, \underbrace{A_{21}, A_{22}, A_{23}, ..., A_{2n}, *, ..., *}_{\text{lda}}, ..., \underbrace{A_{m1}, A_{m2}, A_{m3}, ..., A_{mn}, *, ..., *}_{\text{lda}}}_{\text{m x lda}}]$$

Triangular Matrix

A triangular matrix `A` of `n` rows and `n` columns with leading dimension `lda` is represented as a one dimensional array `a`, of a size of at least `lda * n`. When column (respectively row) major layout is used, the elements of each column (respectively row) are contiguous in memory while the elements of each row (respectively column) are at distance `lda` from the element in the same row (respectively column) and the previous column (respectively row).

Before entry in any BLAS function using a triangular matrix,

- If `upper_lower = uplo::upper`, the leading `n` by `n` upper triangular part of the array `a` must contain the upper triangular part of the matrix `A`. The strictly lower triangular part of the array `a` is not referenced. In other words, the matrix

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & \dots & A_{1n} \\ * & A_{22} & A_{23} & \dots & A_{2n} \\ * & * & A_{33} & \dots & A_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ * & * & * & \dots & A_{nn} \end{bmatrix}$$

  is stored in memory as the array

  - For column major layout,

$$a=[\underbrace{\underbrace{A_{11}, *, ..., *}_{\text{lda}}, \underbrace{A_{12}, A_{22}, *, ..., *}_{\text{lda}}, ..., \underbrace{A_{1n}, A_{2n}, A_{3n}, ..., A_{nn}, *, ..., *}_{\text{lda}}}_{\text{lda x n}}]$$

  - For row major layout,

$$a=[\underbrace{\underbrace{A_{11}, A_{12}, A_{13}, ..., A_{1n}, *, ..., *}_{\text{lda}}, \underbrace{*, A_{22}, A_{23}, ..., A_{2n}, *, ..., *}_{\text{lda}}, ..., \underbrace{*, ..., *, A_{nn}, *, ..., *}_{\text{lda}}}_{\text{lda x n}}]$$

- If `upper_lower = uplo::lower`, the leading `n` by `n` lower triangular part of the array `a` must contain the lower triangular part of the matrix `A`. The strictly upper triangular part of the array `a` is not referenced. That is, the matrix

$$A = \begin{bmatrix} A_{11} & * & * & \dots & * \\ A_{21} & A_{22} & * & \dots & * \\ A_{31} & A_{32} & A_{33} & \dots & * \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & A_{n3} & \dots & A_{nn} \end{bmatrix}$$

  is stored in memory as the array

  - For column major layout,

$$a=[\underbrace{\underbrace{A_{11}, A_{21}, A_{31}, .., A_{n1}, *, ..., *}_{\text{lda}}, \underbrace{*, A_{22}, A_{32}, ..., A_{n2}, *, ..., *}_{\text{lda}}, ..., \underbrace{*, ..., *, A_{nn}, *, ..., *}_{\text{lda}}}_{\text{lda x n}}]$$

  - For row major layout,

$$a=[\underbrace{\underbrace{A_{11}, *, ..., *}_{\text{lda}}, \underbrace{A_{21}, A_{22}, *, ..., *}_{\text{lda}}, ..., \underbrace{A_{n1}, A_{n2}, A_{n3}, ..., A_{nn}, *, ..., *}_{\text{lda}}}_{\text{lda x n}}]$$

Band Matrix

A general band matrix `A` of `m` rows and `n` columns with `kl` sub-diagonals, `ku` super-diagonals, and leading dimension `lda` is represented as a one dimensional array `a` of a size of at least `lda` * `n` (respectively `lda` * `m`) if column (respectively row) major layout is used.

Before entry in any BLAS function using a general band matrix, the leading (`kl` + `ku` + 1) by `n` (respectively `m`) part of the array `a` must contain the matrix `A`. This matrix must be supplied column-by-column (respectively row-by-row), with the main diagonal of the matrix in row `ku` (respectively `kl`) of the array (0-based indexing), the first super-diagonal starting at position 1 (respectively 0) in row (`ku` - 1) (respectively column (`kl` + 1)), the first sub-diagonal starting at position 0 (respectively 1) in row (`ku` + 1) (respectively column (`kl` - 1)), and so on. Elements in the array `a` that do not correspond to elements in the band matrix (such as the top left `ku` by `ku` triangle) are not referenced.

Visually, the matrix `A`

$$A = \begin{bmatrix}
A_{11} & A_{12} & A_{13} & \dots & A_{1,ku+1} & * & \dots & \dots & \dots & \dots & \dots & * \\
A_{21} & A_{22} & A_{23} & A_{24} & \dots & A_{2,ku+2} & * & \dots & \dots & \dots & \dots & * \\
A_{31} & A_{32} & A_{33} & A_{34} & A_{35} & \dots & A_{3,ku+3} & * & \dots & \dots & \dots & * \\
\vdots & A_{42} & A_{43} & \ddots & \ddots & \ddots & \ddots & \ddots & * & \dots & \dots & \vdots \\
A_{kl+1,1} & \vdots & A_{53} & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & * & \dots & \vdots \\
* & A_{kl+2,2} & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\
\vdots & * & A_{kl+3,3} & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & * \\
\vdots & \vdots & * & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & A_{n-ku,n} \\
\vdots & \vdots & \vdots & * & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\
\vdots & \vdots & \vdots & \vdots & * & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & A_{m-2,n} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & A_{m-1,n} \\
* & * & * & \dots & \dots & \dots & * & A_{m,m-kl} & \dots & A_{m,n-2} & A_{m,n-1} & A_{m,n}
\end{bmatrix}$$

is stored in memory as an array

- For column major layout,

$$a=[\underbrace{*,\dots,*}_{\text{ku}},A_{11},A_{12},\dots,A_{min(kl+1,m),1},*,\dots,*,\underbrace{*,\dots,*}_{\text{ku-1}},A_{max(1,2-ku),2},\dots,A_{min(kl+2,m),2},*,\dots*,\dots,\underbrace{*,\dots,*}_{\text{max(0,ku-n+1)}},A_{max(1,n-ku),n},\dots,$$

lda | lda | lda

lda x n

- For row major layout,

$$a=[\underbrace{*,\dots,*}_{\text{kl}},A_{11},A_{12},\dots,A_{1,min(ku+1,n)},*,\dots,*,\underbrace{*,\dots,*}_{\text{kl-1}},A_{2,max(1,2-kl)},\dots,A_{2,min(ku+2,n)},*,\dots*,\dots,\underbrace{*,\dots,*}_{\text{max(0,kl-m+1)}},A_{m,max(1,m-kl)},\dots,$$

lda | lda | lda

lda x m

The following program segment transfers a band matrix from conventional full matrix storage (variable `matrix`, with leading dimension `ldm`) to band storage (variable `a`, with leading dimension `lda`):

- Using matrices stored with column major layout,

```
for (j = 0; j < n; j++) {
    k = ku - j;
    for (i = max(0, j - ku); i < min(m, j + kl + 1); i++) {
        a[(k + i) + j * lda] = matrix[i + j * ldm];
    }
}
```

- Using matrices stored with row major layout,

```
for (i = 0; i < m; i++) {
    k = kl - i;
    for (j = max(0, i - kl); j < min(n, i + ku + 1); j++) {
        a[(k + j) + i * lda] = matrix[j + i * ldm];
    }
}
```

Triangular Band Matrix

A triangular band matrix `A` of `n` rows and `n` columns with `k` sub/super-diagonals and leading dimension `lda` is represented as a one dimensional array `a` of size at least `lda` * `n`.

Before entry in any BLAS function using a triangular band matrix,

- If `upper_lower = uplo::upper`, the leading (`k` + 1) by `n` part of the array `a` must contain the upper triangular band part of the matrix `A`. When using column major layout, this matrix must be supplied column-by-column (respectively row-by-row) with the main diagonal of the matrix in row (`k`) (respectively column 0) of the array, the first super-diagonal starting at position 1 (respectively 0) in row (`k` - 1) (respectively column 1), and so on. Elements in the array `a` that do not correspond to elements in the triangular band matrix (such as the top left `k` by `k` triangle) are not referenced.

  Visually, the matrix

$$A = \begin{bmatrix}
A_{11} & A_{12} & A_{13} & \dots & A_{1,k+1} & * & \dots & \dots & \dots & \dots & \dots & * \\
* & A_{22} & A_{23} & A_{24} & \dots & A_{2,k+2} & * & \dots & \dots & \dots & \dots & * \\
\vdots & * & A_{33} & A_{34} & A_{35} & \dots & A_{3,k+3} & * & \dots & \dots & \dots & * \\
\vdots & \vdots & * & \ddots & \ddots & \ddots & \ddots & \ddots & * & \dots & \dots & \vdots \\
\vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & * & \dots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & * \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & A_{n-k,n} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & A_{n-2,n} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & A_{n-1,n} \\
* & * & * & \dots & \dots & \dots & \dots & \dots & \dots & \dots & * & A_{n,n}
\end{bmatrix}$$

  is stored as an array

- For column major layout,

$$a=[\underbrace{*, \dots, *}_{\text{ku}}, A_{11}, *, \dots, *, \underbrace{*, \dots, *}_{\text{ku-1}}, A_{max(1,2-k),2}, \dots, A_{2,2}, *, \dots *, \dots, \underbrace{*, \dots, *}_{\text{max(0,k-n+1)}}, A_{max(1,n-k),n}, \dots, A_{n,n}, *, \dots *]$$

  lda (ku), lda (ku-1), lda (max(0,k-n+1)), lda x n

- For row major layout,

$$a=[\underbrace{\underbrace{A_{11}, A_{21}, ..., A_{min(k+1,n),1}, *, ..., *}_{\text{lda}}, \underbrace{A_{2,2}, ..., A_{min(k+2,n),2}, *, ..., *}_{\text{lda}}, ..., \underbrace{A_{n,n}, *, ...*}_{\text{lda}}}_{\text{lda x n}}]$$

The following program segment transfers a band matrix from conventional full matrix storage (variable `matrix`, with leading dimension `ldm`) to band storage (variable `a`, with leading dimension `lda`):

- Using matrices stored with column major layout,

```
for (j = 0; j < n; j++) {
    m = k - j;
    for (i = max(0, j - k); i <= j; i++) {
        a[(m + i) + j * lda] = matrix[i + j * ldm];
    }
}
```

- Using matrices stored with row major layout,

```
for (i = 0; i < n; i++) {
    m = -i;
    for (j = i; j < min(n, i + k + 1); j++) {
        a[(m + j) + i * lda] = matrix[j + i * ldm];
    }
}
```

- If `upper_lower = uplo::lower`, the leading (`k` + 1) by `n` part of the array `a` must contain the upper triangular band part of the matrix `A`. This matrix must be supplied column-by-column with the main diagonal of the matrix in row 0 of the array, the first sub-diagonal starting at position 0 in row 1, and so on. Elements in the array `a` that do not correspond to elements in the triangular band matrix (such as the bottom right `k` by `k` triangle) are not referenced.

  That is, the matrix

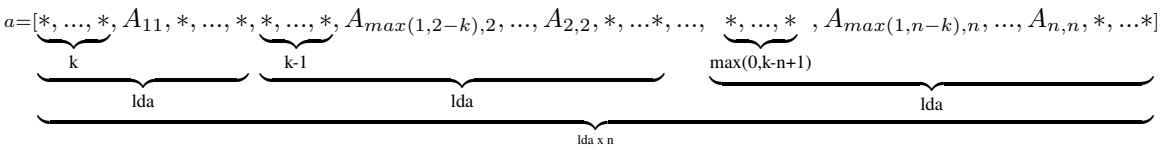
$$A = \begin{bmatrix}
A_{11} & * & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & * \\
A_{21} & A_{22} & * & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & * \\
A_{31} & A_{32} & A_{33} & * & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & * \\
\vdots & A_{42} & A_{43} & \ddots & \ddots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \vdots \\
A_{k+1,1} & \vdots & A_{53} & \ddots & \ddots & \ddots & \cdots & \cdots & \cdots & \cdots & \cdots & \vdots \\
* & A_{k+2,2} & \vdots & \ddots & \ddots & \ddots & \ddots & \cdots & \cdots & \cdots & \cdots & \vdots \\
\vdots & * & A_{k+3,3} & \ddots & \ddots & \ddots & \ddots & \ddots & \cdots & \cdots & \cdots & \vdots \\
\vdots & \vdots & * & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \cdots & \cdots & \vdots \\
\vdots & \vdots & \vdots & * & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \cdots & \vdots \\
\vdots & \vdots & \vdots & \vdots & * & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & * \\
* & * & * & \cdots & \cdots & \cdots & * & A_{n,n-k} & \cdots & A_{n,n-2} & A_{n,n-1} & A_{n,n}
\end{bmatrix}$$

  is stored as the array

- For column major layout,

$$a=[\underbrace{\underbrace{A_{11}, A_{21}, ..., A_{min(k+1,n),1}, *, ..., *}_{\text{lda}}, \underbrace{A_{2,2}, ..., A_{min(k+2,n),2}, *, ..., *}_{\text{lda}}, ..., \underbrace{A_{n,n}, *, ...*}_{\text{lda}}}_{\text{lda x n}}]$$

- For row major layout,



The following program segment transfers a band matrix from conventional full matrix storage (variable `matrix`, with leading dimension `ldm`) to band storage (variable `a`, with leading dimension `lda`):

- Using matrices stored with column major layout,

```
for (j = 0; j < n; j++) {
    m = -j;
    for (i = j; i < min(n, j + k + 1); i++) {
        a[(m + i) + j * lda] = matrix[i + j * ldm];
    }
}
```

- Using matrices stored with row major layout,

```
for (i = 0; i < n; i++) {
    m = k - i;
    for (j = max(0, i - k); j <= i; j++) {
        a[(m + j) + i * lda] = matrix[j + i * ldm];
    }
}
```

Packed Triangular Matrix

A triangular matrix `A` of `n` rows and `n` columns is represented in packed format as a one dimensional array `a` of size at least (`n`*(`n` + 1))/2. All elements in the upper or lower part of the matrix `A` are stored contiguously in the array `a`.

Before entry in any BLAS function using a triangular packed matrix,

- If `upper_lower = uplo::upper`, if column (respectively row) major layout is used, the first (`n`*(`n` + 1))/2 elements in the array `a` must contain the upper triangular part of the matrix `A` packed sequentially, column by column (respectively row by row) so that `a`[0] contains A_{11}, `a`[1] and `a`[2] contain A_{12} and A_{22} (respectively A_{13}) respectively, and so on. Hence, the matrix

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & \dots & A_{1n} \\ * & A_{22} & A_{23} & \dots & A_{2n} \\ * & * & A_{33} & \dots & A_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ * & * & * & \dots & A_{nn} \end{bmatrix}$$

  is stored as the array

  - For column major layout,

$$a=[A_{11},A_{12},A_{22},A_{13},A_{23},A_{33},...,A_{(n-1),n},A_{nn}]$$

  - For row major layout,

$$a=[A_{11},A_{12},A_{13},...,A_{1n},A_{22},A_{23},...,A_{2n},...,A_{(n-1),(n-1)},A_{(n-1),n},A_{nn}]$$

- If `upper_lower = uplo::lower`, if column (respectively row) major layout is used, the first (`n`*(`n` + 1))/2 elements in the array `a` must contain the lower triangular part of the matrix `A` packed sequentially, column by column (row by row) so that `a`[0] contains A_{11}, `a`[1] and `a`[2] contain A_{21} and A_{31} (respectively A_{22}) respectively, and so on. The matrix

$$A = \begin{bmatrix} A_{11} & * & * & \dots & * \\ A_{21} & A_{22} & * & \dots & * \\ A_{31} & A_{32} & A_{33} & \dots & * \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & A_{n3} & \dots & A_{nn} \end{bmatrix}$$

  is stored as the array

  - For column major layout,

$$a=[A_{11},A_{21},A_{31},...,A_{n1},A_{22},A_{32},...,A_{n2},...,A_{(n-1),(n-1)},A_{n,(n-1)},A_{nn}]$$

  - For row major layout,

$$a=[A_{11},A_{21},A_{22},A_{31},A_{32},A_{33},...,A_{n,(n-1)},A_{nn}]$$

Vector

A vector `X` of `n` elements with increment `incx` is represented as a one dimensional array `x` of size at least (1 + (`n` - 1) * abs(`incx`)).

Visually, the vector

$$X = (X_1, X_2, X_3, ..., X_n)$$

is stored in memory as an array

$$x=[\underbrace{\underbrace{X_1, *, ..., *}_{\text{incx}}, \underbrace{X_2, *, ..., *}_{\text{incx}}, ..., \underbrace{X_{n-1}, *, ..., *}_{\text{incx}}, X_n}_{\text{1 + (n-1) x incx}}] \quad if\ incx > 0$$

$$x=[\underbrace{\underbrace{X_n, *, ..., *}_{\text{|incx|}}, \underbrace{X_{n-1}, *, ..., *}_{\text{|incx|}}, ..., \underbrace{X_2, *, ..., *}_{\text{|incx|}}, X_1}_{\text{1 + (1-n) x incx}}] \quad if\ incx < 0$$

Parent topic: *Dense Linear Algebra*

Scalar Arguments in BLAS

The USM version of oneMKL BLAS routines for DPC++ will accept either a scalar (for example `float`) or pointer (`float*`) for parameters that represent a single fixed value (not a vector or matrix). These parameters are often named `alpha` or `beta` in BLAS.

Basic Use

Users can call gemv with pointers:

```
float *alpha_ptr = sycl::malloc_shared<float>(1, queue);
float *beta_ptr = sycl::malloc_shared<float>(1, queue);
// fill alpha_ptr and beta_ptr with desired values
oneapi::mkl::blas::column_major::gemv(queue, trans, m, n, alpha_ptr, lda, x,␣
↪incx, beta_ptr,
                                      y, incy).wait();
```

or with literal values:

```
oneapi::mkl::blas::column_major::gemv(queue, trans, m, n, 2, lda, x, incx, 2.7,
                                      y, incy).wait();
```

Users can even mix scalar and pointer parameters in a single call:

```
float *alpha_ptr = sycl::malloc_shared<float>(1, queue);
oneapi::mkl::blas::column_major::gemv(queue, trans, m, n, alpha_ptr, lda, x,␣
↪incx, 2.7,
                                    y, incy).wait();
```

Pointers provided for scalar parameters may be SYCL-managed pointers to either device or host memory (for example pointers created with sycl::malloc_device, sycl::malloc_shared, or sycl::malloc_host), or they may be raw pointers created with malloc or new.

For most users, this is all they need to know. A few details about how this is implemented are provided below.

Wrapper type

The USM version of oneMKL BLAS routines use a templated value_or_pointer<T> wrapper to enable either pointers or values to be passed to routines that take a scalar parameter.

In general, users should not explicitly use this type in their code. There is no need to construct an object of type value_or_pointer in order to use the oneMKL functions that include it in their function signatures. Instead, values and pointers in user code will be implicitly converted to this type when a user calls a oneMKL function.

The value_or_pointer<T> wrapper has two constructors, one that converts a value of type T (or anything convertible to T) to value_or_pointer<T>, and another that converts a pointer to T to value_or_pointer<T>. Internally, the oneMKL functions can behave slightly differently depending on whether the underlying data is a value or a pointer, and if it points to host-side memory or device-side memory, but these uses should be transparent to users.

Dependencies

For scalar parameters passed to oneMKL BLAS routines as pointers, the timing of pointer dereferencing depends on whether it is a USM-managed pointer or a raw pointer.

For a USM-managed pointer, it is dereferenced at kernel launch after the dependencies passed to the function have been resolved, so the value may be assigned asynchronously in another event passed as a dependency to the routine.

A raw pointer (such as those allocated with malloc or new) is dereferenced at the function call, so it must be valid when the function is called. In this case the data must be valid when the function is called and it may not be assigned asynchronously.

Parent topic: *Dense Linear Algebra*

BLAS Routines

oneMKL provides DPC++ interfaces to the Basic Linear Algebra Subprograms (BLAS) routines (Level1, Level2, Level3), as well as several BLAS-like extension routines.

BLAS Level 1 Routines

BLAS Level 1 includes routines which perform vector-vector operations as described in the following table.

Routines	Description
asum	Sum of vector magnitudes
axpy	Scalar-vector product
copy	Copy vector
dot	Dot product
sdsdot	Dot product with double precision
dotc	Dot product conjugated
dotu	Dot product unconjugated
nrm2	Vector 2-norm (Euclidean norm)
rot	Plane rotation of points
rotg	Generate Givens rotation of points
rotm	Modified Givens plane rotation of points
rotmg	Generate modified Givens plane rotation of points
scal	Vector-scalar product
swap	Vector-vector swap
iamax	Index of the maximum absolute value element of a vector
iamin	Index of the minimum absolute value element of a vector

asum

Computes the sum of magnitudes of the vector elements.

Description

The `asum` routine computes the sum of the magnitudes of elements of a real vector, or the sum of magnitudes of the real and imaginary parts of elements of a complex vector:

$$result = \sum_{i=1}^{n} (|Re(x_i)| + |Im(x_i)|)$$

where `x` is a vector with `n` elements.

`asum` supports the following precisions for data:

T	Tres
`float`	`float`
`double`	`double`
`std::complex<float>`	`float`
`std::complex<double>`	`double`

asum (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void asum(sycl::queue &queue,
              std::int64_t n,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<Tres,1> &result)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void asum(sycl::queue &queue,
              std::int64_t n,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<Tres,1> &result)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

n
: Number of elements in vector `x`.

x
: Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
: Stride of vector `x`.

Output Parameters

result
: Buffer where the scalar result is stored (the sum of magnitudes of the real and imaginary parts of all elements of the vector).

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

asum (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event asum(sycl::queue &queue,
                     std::int64_t n,
                     const T *x,
                     std::int64_t incx,
                     Tres *result,
                     const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event asum(sycl::queue &queue,
                     std::int64_t n,
                     const T *x,
                     std::int64_t incx,
                     Tres *result,
                     const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
: The queue where the routine should be executed.

n
: Number of elements in vector `x`.

x
: Pointer to input vector `x`. The array holding the vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
: Stride of vector `x`.

dependencies
: List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

result
: Pointer to the output matrix where the scalar result is stored (the sum of magnitudes of the real and imaginary parts of all elements of the vector).

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

axpy

Computes a vector-scalar product and adds the result to a vector.

Description

The `axpy` routines compute a scalar-vector product and add the result to a vector:

$$y \leftarrow alpha * x + y$$

where:

`x` and `y` are vectors of `n` elements,

`alpha` is a scalar.

`axpy` supports the following precisions.

T
`half`
`bfloat16`
`float`
`double`
`std::complex<float>`
`std::complex<double>`

axpy (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void axpy(sycl::queue &queue,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void axpy(sycl::queue &queue,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

n
: Number of elements in vector `x`.

alpha
: Specifies the scalar alpha.

x
: Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` – 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
: Stride of vector `x`.

y
: Buffer holding input vector `y`. The buffer must be of size at least (1 + (`n` – 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
: Stride of vector `y`.

Output Parameters

y
Buffer holding the updated vector y.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

axpy (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event axpy(sycl::queue &queue,
                     std::int64_t n,
                     value_or_pointer<T> alpha,
                     const T *x,
                     std::int64_t incx,
                     T *y,
                     std::int64_t incy,
                     const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event axpy(sycl::queue &queue,
                     std::int64_t n,
                     value_or_pointer<T> alpha,
                     const T *x,
                     std::int64_t incx,
                     T *y,
                     std::int64_t incy,
                     const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

n
Number of elements in vector `x`.

alpha
Specifies the scalar alpha. See *Scalar Arguments in BLAS* for more details.

x
Pointer to the input vector `x`. The array holding the vector `x` must be of size at least (1 + (`n` – 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`.

y
Pointer to the input vector `y`. The array holding the vector `y` must be of size at least (1 + (`n` – 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

y
Pointer to the updated vector `y`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

copy

Copies a vector to another vector.

Description

The copy routines copy one vector to another:

$$y \leftarrow x$$

where x and y are vectors of n elements.

copy supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

copy (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void copy(sycl::queue &queue,
              std::int64_t n,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void copy(sycl::queue &queue,
              std::int64_t n,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

Input Parameters

queue
The queue where the routine should be executed.

n
Number of elements in vector `x`.

x
Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` – 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`.

incy
Stride of vector `y`.

Output Parameters

y
Buffer holding the updated vector `y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

copy (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event copy(sycl::queue &queue,
                     std::int64_t n,
                     const T *x,
                     std::int64_t incx,
                     T *y,
                     std::int64_t incy,
                     const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event copy(sycl::queue &queue,
                     std::int64_t n,
                     const T *x,
                     std::int64_t incx,
                     T *y,
                     std::int64_t incy,
                     const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

n
Number of elements in vector `x`.

x
Pointer to the input vector `x`. The array holding the vector `x` must be of size at least (1 + (`n` – 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`.

incy
Stride of vector `y`.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

y
Pointer to the updated vector `y`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

dot

Computes the dot product of two real vectors.

Description

The `dot` routines perform a dot product between two vectors:

$$result = \sum_{i=1}^{n} X_i Y_i$$

`dot` supports the following precisions for data.

T	Tres
half	half
bfloat16	bfloat16
float	float
double	double
float	double

Note

For the mixed precision version (inputs are float while result is double), the dot product is computed with double precision.

dot (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void dot(sycl::queue &queue,
             std::int64_t n,
             sycl::buffer<T,1> &x,
             std::int64_t incx,
             sycl::buffer<T,1> &y,
             std::int64_t incy,
             sycl::buffer<Tres,1> &result)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void dot(sycl::queue &queue,
             std::int64_t n,
             sycl::buffer<T,1> &x,
             std::int64_t incx,
             sycl::buffer<T,1> &y,
             std::int64_t incy,
             sycl::buffer<Tres,1> &result)
}
```

Input Parameters

queue
The queue where the routine should be executed.

n
Number of elements in vectors `x` and `y`.

x
Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` – 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`.

y
Buffer holding input vector `y`. The buffer must be of size at least (1 + (`n` – 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`.

Output Parameters

result
Buffer where the result (a scalar) will be stored.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

dot (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event dot(sycl::queue &queue,
                    std::int64_t n,
                    const T *x,
                    std::int64_t incx,
                    const T *y,
                    std::int64_t incy,
                    Tres *result,
                    const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event dot(sycl::queue &queue,
                    std::int64_t n,
                    const T *x,
                    std::int64_t incx,
                    const T *y,
                    std::int64_t incy,
                    Tres *result,
                    const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

n
Number of elements in vectors `x` and `y`.

x
Pointer to the input vector `x`. The array holding the vector `x` must be of size at least (1 + (`n` – 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`.

y
Pointer to the input vector `y`. The array holding the vector `y` must be of size at least (1 + (`n` – 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

result
Pointer to where the result (a scalar) will be stored.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

sdsdot

Computes a vector-vector dot product with double precision.

Description

The `sdsdot` routines perform a dot product between two vectors with double precision:

$$result = sb + \sum_{i=1}^{n} X_i Y_i$$

sdsdot (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void sdsdot(sycl::queue &queue,
                std::int64_t n,
                float sb,
                sycl::buffer<float,1> &x,
                std::int64_t incx,
                sycl::buffer<float,1> &y,
                std::int64_t incy,
                sycl::buffer<float,1> &result)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void sdsdot(sycl::queue &queue,
                std::int64_t n,
                float sb,
                sycl::buffer<float,1> &x,
                std::int64_t incx,
                sycl::buffer<float,1> &y,
                std::int64_t incy,
                sycl::buffer<float,1> &result)
}
```

Input Parameters

queue
The queue where the routine should be executed.

n
Number of elements in vectors `x` and `y`.

sb
Single precision scalar to be added to the dot product.

x
Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`.

y
Buffer holding input vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incxy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`.

Output Parameters

result
Buffer where the result (a scalar) will be stored. If `n` < 0 the result is `sb`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

sdsdot (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event sdsdot(sycl::queue &queue,
                       std::int64_t n,
                       float sb,
                       const float *x,
                       std::int64_t incx,
                       const float *y,
```

(continues on next page)

(continued from previous page)

```
                        std::int64_t incy,
                        float *result,
                        const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event sdsdot(sycl::queue &queue,
                        std::int64_t n,
                        float sb,
                        const float *x,
                        std::int64_t incx,
                        const float *y,
                        std::int64_t incy,
                        float *result,
                        const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
: The queue where the routine should be executed.

n
: Number of elements in vectors `x` and `y`.

sb
: Single precision scalar to be added to the dot product.

x
: Pointer to the input vector `x`. The array must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
: Stride of vector `x`.

y
: Pointer to the input vector `y`. The array must be of size at least (1 + (`n` - 1)*abs(`incxy`)). See *Matrix Storage* for more details.

incy
: Stride of vector `y`.

dependencies
: List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

result
Pointer to where the result (a scalar) will be stored. If `n` < 0 the result is `sb`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

dotc

Computes the dot product of two complex vectors, conjugating the first vector.

Description

The `dotc` routines perform a dot product between two complex vectors, conjugating the first of them:

$$result = \sum_{i=1}^{n} \overline{X_i} Y_i$$

`dotc` supports the following precisions for data.

T
`std::complex<float>`
`std::complex<double>`

dotc (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void dotc(sycl::queue &queue,
              std::int64_t n,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy,
              sycl::buffer<T,1> &result)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void dotc(sycl::queue &queue,
              std::int64_t n,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy,
              sycl::buffer<T,1> &result)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

n
: The number of elements in vectors `x` and `y`.

x
: Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
: The stride of vector `x`.

y
: Buffer holding input vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details..

incy
: The stride of vector `y`.

Output Parameters

result
The buffer where the result (a scalar) is stored.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

dotc (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void dotc(sycl::queue &queue,
              std::int64_t n,
              const T *x,
              std::int64_t incx,
              const T *y,
              std::int64_t incy,
              T *result,
              const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    void dotc(sycl::queue &queue,
              std::int64_t n,
              const T *x,
              std::int64_t incx,
              const T *y,
              std::int64_t incy,
              T *result,
              const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

n
The number of elements in vectors `x` and `y`.

x
Pointer to input vector `x`. The array holding the input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
The stride of vector `x`.

y
Pointer to input vector `y`. The array holding the input vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details..

incy
The stride of vector `y`.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

result
The pointer to where the result (a scalar) is stored.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

dotu

Computes the dot product of two complex vectors.

Description

The `dotu` routines perform a dot product between two complex vectors:

$$result = \sum_{i=1}^{n} X_i Y_i$$

`dotu` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

dotu (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void dotu(sycl::queue &queue,
              std::int64_t n,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy,
              sycl::buffer<T,1> &result)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void dotu(sycl::queue &queue,
              std::int64_t n,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy,
              sycl::buffer<T,1> &result)
}
```

Input Parameters

queue
The queue where the routine should be executed.

n
Number of elements in vectors `x` and `y`.

x
Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`.

y
Buffer holding input vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`.

Output Parameters

result
Buffer where the result (a scalar) is stored.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

dotu (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event dotu(sycl::queue &queue,
                     std::int64_t n,
                     const T *x,
                     std::int64_t incx,
                     const T *y,
                     std::int64_t incy,
                     T *result,
                     const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event dotu(sycl::queue &queue,
                     std::int64_t n,
                     const T *x,
                     std::int64_t incx,
                     const T *y,
                     std::int64_t incy,
                     T *result,
                     const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

n
Number of elements in vectors `x` and `y`.

x
Pointer to the input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`.

y
Pointer to input vector `y`. The array holding input vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

result
Pointer to where the result (a scalar) is stored.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

nrm2

Computes the Euclidean norm of a vector.

Description

The `nrm2` routines computes Euclidean norm of a vector

$$result = \|x\|$$

where:

`x` is a vector of `n` elements.

`nrm2` supports the following precisions.

T	Tres
`half`	`half`
`bfloat16`	`bfloat16`
`float`	`float`
`double`	`double`
`std::complex<float>`	`float`
`std::complex<double>`	`double`

nrm2 (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void nrm2(sycl::queue &queue,
              std::int64_t n,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<Tres,1> &result)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void nrm2(sycl::queue &queue,
              std::int64_t n,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<Tres,1> &result)
}
```

Input Parameters

queue
The queue where the routine should be executed.

n
Number of elements in vector `x`.

x
Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`.

Output Parameters

result
Buffer where the Euclidean norm of the vector `x` will be stored.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

nrm2 (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event nrm2(sycl::queue &queue,
                     std::int64_t n,
                     const T *x,
                     std::int64_t incx,
                     Tres *result,
```

(continues on next page)

(continued from previous page)

```
                    const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event nrm2(sycl::queue &queue,
                     std::int64_t n,
                     const T *x,
                     std::int64_t incx,
                     Tres *result,
                     const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
: The queue where the routine should be executed.

n
: Number of elements in vector `x`.

x
: Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
: Stride of vector `x`.

dependencies
: List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

result
: Pointer to where the Euclidean norm of the vector `x` will be stored.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

rot

Performs rotation of points in the plane.

Description

Given two vectors `x` and `y` of `n` elements, the `rot` routines compute four scalar-vector products and update the input vectors with the sum of two of these scalar-vector products as follows:

$$\begin{bmatrix} x \\ y \end{bmatrix} \leftarrow \begin{bmatrix} c * x + s * y \\ -s * x + c * y \end{bmatrix}$$

If `s` is a complex type, the operation is defined as:

$$\begin{bmatrix} x \\ y \end{bmatrix} \leftarrow \begin{bmatrix} c * x + s * y \\ -conj(s) * x + c * y \end{bmatrix}$$

`rot` supports the following precisions.

T	Tc	Ts
sycl::half	sycl::half	sycl::half
oneapi::mkl::bfloat16	oneapi::mkl::bfloat16	oneapi::mkl::bfloat16
float	float	float
double	double	double
std::complex<float>	float	std::complex<float>
std::complex<double>	double	std::complex<double>
std::complex<float>	float	float
std::complex<double>	double	double

rot (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void rot(sycl::queue &queue,
             std::int64_t n,
             sycl::buffer<T,1> &x,
             std::int64_t incx,
             sycl::buffer<T,1> &y,
             std::int64_t incy,
             Tc c,
             Ts s)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void rot(sycl::queue &queue,
             std::int64_t n,
             sycl::buffer<T,1> &x,
```

(continues on next page)

(continued from previous page)

```
            std::int64_t incx,
            sycl::buffer<T,1> &y,
            std::int64_t incy,
            Tc c,
            Ts s)
}
```

Input Parameters

queue
The queue where the routine should be executed.

n
Number of elements in vector `x`.

x
Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`.

y
Buffer holding input vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`.

c
Scaling factor.

s
Scaling factor.

Output Parameters

x
Buffer holding updated buffer `x`.

y
Buffer holding updated buffer `y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

rot (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event rot(sycl::queue &queue,
                    std::int64_t n,
                    T *x,
                    std::int64_t incx,
                    T *y,
                    std::int64_t incy,
                    value_or_pointer<Tc> c,
                    value_or_pointer<Ts> s,
                    const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event rot(sycl::queue &queue,
                    std::int64_t n,
                    T *x,
                    std::int64_t incx,
                    T *y,
                    std::int64_t incy,
                    value_or_pointer<Tc> c,
                    value_or_pointer<Ts> s,
                    const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
: The queue where the routine should be executed.

n
: Number of elements in vector `x`.

x
: Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
: Stride of vector `x`.

y
: Pointer to input vector `y`. The array holding input vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
: Stride of vector `y`.

c
: Scaling factor. See *Scalar Arguments in BLAS* for more details.

s
: Scaling factor. See *Scalar Arguments in BLAS* for more details.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

x
Pointer to the updated matrix `x`.

y
Pointer to the updated matrix `y`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

rotg

Computes the parameters for a Givens rotation.

Description

Given the Cartesian coordinates `(a, b)` of a point, the `rotg` routines return the parameters `c`, `s`, `r`, and `z` associated with the Givens rotation. The parameters `c` and `s` define a unitary matrix such that:

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \cdot \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

The parameter `z` is defined such that if |a| > |b|, `z` is `s`; otherwise if `c` is not 0 `z` is 1/`c`; otherwise `z` is 1.

`rotg` supports the following precisions.

T	Tc
`float`	`float`
`double`	`double`
`std::complex<float>`	`float`
`std::complex<double>`	`double`

rotg (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void rotg(sycl::queue &queue,
              sycl::buffer<T,1> &a,
              sycl::buffer<T,1> &b,
              sycl::buffer<Tc,1> &c,
              sycl::buffer<T,1> &s)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void rotg(sycl::queue &queue,
              sycl::buffer<T,1> &a,
              sycl::buffer<T,1> &b,
              sycl::buffer<Tc,1> &c,
              sycl::buffer<T,1> &s)
}
```

Input Parameters

queue
: The queue where the routine should be executed

a
: Buffer holding the `x`-coordinate of the point.

b
: Buffer holding the `y`-coordinate of the point.

Output Parameters

a
: Buffer holding the parameter `r` associated with the Givens rotation.

b
: Buffer holding the parameter `z` associated with the Givens rotation.

c
: Buffer holding the parameter `c` associated with the Givens rotation.

s
: Buffer holding the parameter `s` associated with the Givens rotation.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

rotg (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event rotg(sycl::queue &queue,
                     T *a,
                     T *b,
                     Tc *c,
                     T *s,
                     const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event rotg(sycl::queue &queue,
                     T *a,
                     T *b,
                     Tc *c,
                     T *s,
                     const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
: The queue where the routine should be executed

a
: Pointer to the `x`-coordinate of the point.

b
: Pointer to the `y`-coordinate of the point.

dependencies
: List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

a
Pointer to the parameter `r` associated with the Givens rotation.

b
Pointer to the parameter `z` associated with the Givens rotation.

c
Pointer to the parameter `c` associated with the Givens rotation.

s
Pointer to the parameter `s` associated with the Givens rotation.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

rotm

Performs modified Givens rotation of points in the plane.

Description

Given two vectors `x` and `y`, each vector element of these vectors is replaced as follows:

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} = H \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

for `i` from 1 to `n`, where `H` is a modified Givens transformation matrix.

`rotm` supports the following precisions.

T
`float`
`double`

rotm (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void rotm(sycl::queue &queue,
              std::int64_t n,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy,
              sycl::buffer<T,1> &param)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void rotm(sycl::queue &queue,
              std::int64_t n,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy,
              sycl::buffer<T,1> &param)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

n
: Number of elements in vector `x`.

x
: Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
: Stride of vector `x`.

y
: Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
: Stride of vector `y`.

param
: Buffer holding an array of size 5.

  The elements of the `param` array are:

  `param[0]` contains a switch, `flag`. The other array elements `param[1-4]` contain the components of the modified Givens transformation matrix `H`: h_{11}, h_{21}, h_{12}, and h_{22}, respectively.

  Depending on the values of `flag`, the components of `H` are set as follows:

`flag = -1.0`:

$$H = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}$$

`flag = 0.0`:

$$H = \begin{bmatrix} 1.0 & h_{12} \\ h_{21} & 1.0 \end{bmatrix}$$

`flag = 1.0`:

$$H = \begin{bmatrix} h_{11} & 1.0 \\ -1.0 & h_{22} \end{bmatrix}$$

`flag = -2.0`:

$$H = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix}$$

In the last three cases, the matrix entries of 1.0, -1.0, and 0.0 are assumed based on the value of `flag` and are not required to be set in the `param` vector.

Output Parameters

x
Buffer holding updated buffer `x`.

y
Buffer holding updated buffer `y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

rotm (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event rotm(sycl::queue &queue,
                     std::int64_t n,
                     T *x,
                     std::int64_t incx,
                     T *y,
                     std::int64_t incy,
```

(continues on next page)

(continued from previous page)

```
                    const T *param,
                    const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event rotm(sycl::queue &queue,
                    std::int64_t n,
                    T *x,
                    std::int64_t incx,
                    T *y,
                    std::int64_t incy,
                    const T *param,
                    const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

n
Number of elements in vector `x`.

x
Pointer to the input vector `x`. The array holding the vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`.

yparam
Pointer to the input vector `y`. The array holding the vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`.

param
Buffer holding an array of size 5.

The elements of the `param` array are:

`param[0]` contains a switch, `flag`. The other array elements `param[1-4]` contain the components of the modified Givens transformation matrix `H`: h_{11}, h_{21}, h_{12}, and h_{22}, respectively.

Depending on the values of `flag`, the components of `H` are set as follows:

`flag = -1.0`:

$$H = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}$$

`flag = 0.0`:

$$H = \begin{bmatrix} 1.0 & h_{12} \\ h_{21} & 1.0 \end{bmatrix}$$

`flag = 1.0`:

$$H = \begin{bmatrix} h_{11} & 1.0 \\ -1.0 & h_{22} \end{bmatrix}$$

`flag = -2.0`:

$$H = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix}$$

In the last three cases, the matrix entries of 1.0, -1.0, and 0.0 are assumed based on the value of `flag` and are not required to be set in the `param` vector.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

x
Pointer to the updated array `x`.

y
Pointer to the updated array `y`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

rotmg

Computes the parameters for a modified Givens rotation.

Description

Given Cartesian coordinates (`x1`, `y1`) of an input vector, the `rotmg` routines compute the components of a modified Givens transformation matrix `H` that zeros the `y`-component of the resulting vector:

$$\begin{bmatrix} x1 \\ 0 \end{bmatrix} = H \begin{bmatrix} x1\sqrt{d1} \\ y1\sqrt{d2} \end{bmatrix}$$

`rotmg` supports the following precisions.

T
`float`
`double`

rotmg (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void rotmg(sycl::queue &queue,
               sycl::buffer<T,1> &d1,
               sycl::buffer<T,1> &d2,
               sycl::buffer<T,1> &x1,
               sycl::buffer<T,1>  y1,
               sycl::buffer<T,1> &param)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void rotmg(sycl::queue &queue,
               sycl::buffer<T,1> &d1,
               sycl::buffer<T,1> &d2,
               sycl::buffer<T,1> &x1,
               sycl::buffer<T,1>  y1,
               sycl::buffer<T,1> &param)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

d1
: Buffer holding the scaling factor for the `x`-coordinate of the input vector.

d2
: Buffer holding the scaling factor for the `y`-coordinate of the input vector.

x1
: Buffer holding the `x`-coordinate of the input vector.

y1
: Scalar specifying the `y`-coordinate of the input vector.

Output Parameters

d1
Buffer holding the first diagonal element of the updated matrix.

d2
Buffer holding the second diagonal element of the updated matrix.

x1
Buffer holding the `x`-coordinate of the rotated vector before scaling

param
Buffer holding an array of size 5.

The elements of the `param` array are:

`param[0]` contains a switch, `flag`. The other array elements `param[1-4]` contain the components of the modified Givens transformation matrix `H`: h_{11}, h_{21}, h_{12}, and h_{22}, respectively.

Depending on the values of `flag`, the components of `H` are set as follows:

`flag = -1.0`:

$$H = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}$$

`flag = 0.0`:

$$H = \begin{bmatrix} 1.0 & h_{12} \\ h_{21} & 1.0 \end{bmatrix}$$

`flag = 1.0`:

$$H = \begin{bmatrix} h_{11} & 1.0 \\ -1.0 & h_{22} \end{bmatrix}$$

`flag = -2.0`:

$$H = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix}$$

In the last three cases, the matrix entries of 1.0, -1.0, and 0.0 are assumed based on the value of `flag` and are not required to be set in the `param` vector.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

rotmg (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event rotmg(sycl::queue &queue,
                      T *d1,
                      T *d2,
                      T *x1,
                      value_or_pointer<T>  y1,
                      T *param,
                      const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event rotmg(sycl::queue &queue,
                      T *d1,
                      T *d2,
                      T *x1,
                      value_or_pointer<T>  y1,
                      T *param,
                      const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
: The queue where the routine should be executed.

d1
: Pointer to the scaling factor for the `x`-coordinate of the input vector.

d2
: Pointer to the scaling factor for the `y`-coordinate of the input vector.

x1
: Pointer to the `x`-coordinate of the input vector.

y1
: Scalar specifying the `y`-coordinate of the input vector. See *Scalar Arguments in BLAS* for more details.

dependencies
: List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

d1
Pointer to the first diagonal element of the updated matrix.

d2
Pointer to the second diagonal element of the updated matrix.

x1
Pointer to the `x`-coordinate of the rotated vector before scaling

param
Buffer holding an array of size 5.

The elements of the `param` array are:

`param[0]` contains a switch, `flag`. The other array elements `param[1-4]` contain the components of the modified Givens transformation matrix `H`: h_{11}, h_{21}, h_{12}, and h_{22}, respectively.

Depending on the values of `flag`, the components of `H` are set as follows:

`flag = -1.0`:

$$H = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}$$

`flag = 0.0`:

$$H = \begin{bmatrix} 1.0 & h_{12} \\ h_{21} & 1.0 \end{bmatrix}$$

`flag = 1.0`:

$$H = \begin{bmatrix} h_{11} & 1.0 \\ -1.0 & h_{22} \end{bmatrix}$$

`flag = -2.0`:

$$H = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix}$$

In the last three cases, the matrix entries of 1.0, -1.0, and 0.0 are assumed based on the value of `flag` and are not required to be set in the `param` vector.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

scal

Computes the product of a vector by a scalar.

Description

The `scal` routines computes a scalar-vector product:

$$x \leftarrow alpha * x$$

where:

`x` is a vector of `n` elements,

`alpha` is a scalar.

`scal` supports the following precisions.

T	Ts
half	half
bfloat16	bfloat16
float	float
double	double
std::complex<float>	std::complex<float>
std::complex<double>	std::complex<double>
std::complex<float>	float
std::complex<double>	double

scal (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void scal(sycl::queue &queue,
              std::int64_t n,
              Ts alpha,
              sycl::buffer<T,1> &x,
              std::int64_t incx)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void scal(sycl::queue &queue,
              std::int64_t n,
              Ts alpha,
              sycl::buffer<T,1> &x,
              std::int64_t incx)
}
```

Input Parameters

queue
The queue where the routine should be executed.

n
Number of elements in vector `x`.

alpha
Specifies the scalar `alpha`.

x
Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`.

Output Parameters

x
Buffer holding updated buffer `x`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

scal (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event scal(sycl::queue &queue,
                     std::int64_t n,
                     value_or_pointer<Ts> alpha,
                     T *x,
                     std::int64_t incx,
                     const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event scal(sycl::queue &queue,
                     std::int64_t n,
                     value_or_pointer<Ts> alpha,
```

(continues on next page)

(continued from previous page)

```
                        T *x,
                        std::int64_t incx,
                        const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

n
Number of elements in vector `x`.

alpha
Specifies the scalar `alpha`. See *Scalar Arguments in BLAS* for more details.

x
Pointer to the input vector `x`. The array must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`.

Output Parameters

x
Pointer to the updated array `x`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

swap

Swaps a vector with another vector.

Description

Given two vectors of `n` elements, `x` and `y`, the `swap` routines return vectors `y` and `x` swapped, each replacing the other.

$$\begin{bmatrix} y \\ x \end{bmatrix} \leftarrow \begin{bmatrix} x \\ y \end{bmatrix}$$

`swap` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

swap (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void swap(sycl::queue &queue,
              std::int64_t n,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void swap(sycl::queue &queue,
              std::int64_t n,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

Input Parameters

queue
The queue where the routine should be executed.

n
Number of elements in vector `x`.

x
Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`.

y
Buffer holding input vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`.

Output Parameters

x
Buffer holding updated buffer `x`, that is, the input vector `y`.

y
Buffer holding updated buffer `y`, that is, the input vector `x`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

swap (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event swap(sycl::queue &queue,
                     std::int64_t n,
                     T *x,
                     std::int64_t incx,
                     T *y,
                     std::int64_t incy,
```

(continues on next page)

(continued from previous page)

```
                    const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event swap(sycl::queue &queue,
                     std::int64_t n,
                     T *x,
                     std::int64_t incx,
                     T *y,
                     std::int64_t incy,
                     const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

n
Number of elements in vector `x`.

x
Pointer to the input vector `x`. The array must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`.

y
Pointer to the input vector `y`. The array must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

x
Pointer to the updated array `x`, that is, the input vector `y`.

y
Pointer to the updated array `y`, that is, the input vector `x`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

iamax

Finds the index of the element with the largest absolute value in a vector.

Description

The `iamax` routines return an index `i` such that `x[i]` has the maximum absolute value of all elements in vector `x` (real variants), or such that (|Re(`x[i]`)| + |Im(`x[i]`)|) is maximal (complex variants).

The index is zero-based if `base` is set to `oneapi::mkl::index_base::zero` (default) or one-based if it is set to `oneapi::mkl::index_base::one`.

If either `n` or `incx` is not positive, the routine returns `0`, regardless of the base of the index selected.

If more than one vector element is found with the same largest absolute value, the index of the first one encountered is returned.

If the vector contains `NaN` values, then the routine returns the index of the first `NaN`.

`iamax` supports the following precisions.

T
float
double
std::complex<float>
std:complex<double>

iamax (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void iamax(sycl::queue &queue,
               std::int64_t n,
               sycl::buffer<T,1> &x,
               std::int64_t incx,
               sycl::buffer<std::int64_t,1> &result,
               oneapi::mkl::index_base base = oneapi::mkl::index_base::zero)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void iamax(sycl::queue &queue,
               std::int64_t n,
               sycl::buffer<T,1> &x,
               std::int64_t incx,
               sycl::buffer<std::int64_t,1> &result,
               oneapi::mkl::index_base base = oneapi::mkl::index_base::zero)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

n
: The number of elements in vector `x`.

x
: The buffer that holds the input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
: The stride of vector `x`.

base
: Indicates how the output value is indexed. If omitted, defaults to zero-based indexing.

Output Parameters

result
: The buffer where the index `i` of the maximal element is stored.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

iamax (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event iamax(sycl::queue &queue,
                      std::int64_t n,
                      const T *x,
                      std::int64_t incx,
                      std::int64_t *result,
                      oneapi::mkl::index_base base = oneapi::mkl::index_base::zero,
                      const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event iamax(sycl::queue &queue,
                      std::int64_t n,
                      const T *x,
                      std::int64_t incx,
                      std::int64_t *result,
                      oneapi::mkl::index_base base = oneapi::mkl::index_base::zero,
                      const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
: The queue where the routine should be executed.

n
: The number of elements in vector `x`.

x
: The pointer to the input vector `x`. The array holding the input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
: The stride of vector `x`.

base
: Indicates how the output value is indexed. If omitted, defaults to zero-based indexing.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

result
The pointer to where the index `i` of the maximal element is stored.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

iamin

Finds the index of the element with the smallest absolute value.

Description

The `iamin` routines return an index `i` such that `x[i]` has the minimum absolute value of all elements in vector `x` (real variants), or such that (|Re(`x[i]`)| + |Im(`x[i]`)|) is minimal (complex variants).

The index is zero-based if `base` is set to `oneapi::mkl::index_base::zero` (default) or one-based if it is set to `oneapi::mkl::index_base::one`.

If either `n` or `incx` is not positive, the routine returns `0`, regardless of the base of the index selected.

If more than one vector element is found with the same smallest absolute value, the index of the first one encountered is returned.

If the vector contains `NaN` values, then the routine returns the index of the first `NaN`.

`iamin` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

iamin (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void iamin(sycl::queue &queue,
               std::int64_t n,
               sycl::buffer<T,1> &x,
               std::int64_t incx,
               sycl::buffer<std::int64_t,1> &result,
               oneapi::mkl::index_base base = oneapi::mkl::index_base::zero)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void iamin(sycl::queue &queue,
               std::int64_t n,
               sycl::buffer<T,1> &x,
               std::int64_t incx,
               sycl::buffer<std::int64_t,1> &result,
               oneapi::mkl::index_base base = oneapi::mkl::index_base::zero)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

n
: Number of elements in vector `x`.

x
: Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
: Stride of vector x.

base
: Indicates how the output value is indexed. If omitted, defaults to zero-based indexing.

Output Parameters

result
: Buffer where the index `i` of the minimum element will be stored.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

iamin (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event iamin(sycl::queue &queue,
                      std::int64_t n,
                      const T *x,
                      std::int64_t incx,
                      std::int64_t *result,
                      oneapi::mkl::index_base base = oneapi::mkl::index_base::zero,
                      const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event iamin(sycl::queue &queue,
                      std::int64_t n,
                      const T *x,
                      std::int64_t incx,
                      std::int64_t *result,
                      oneapi::mkl::index_base base = oneapi::mkl::index_base::zero,
                      const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
: The queue where the routine should be executed.

n
: Number of elements in vector `x`.

x
: The pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
: Stride of vector x.

base
: Indicates how the output value is indexed. If omitted, defaults to zero-based indexing.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

result
Pointer to where the index `i` of the minimum element will be stored.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

Parent topic: *BLAS Routines*

BLAS Level 2 Routines

BLAS Level 2 includes routines which perform matrix-vector operations as described in the following table.

Routines	Description
gbmv	Matrix-vector product using a general band matrix
gemv	Matrix-vector product using a general matrix
ger	Rank-1 update of a general matrix
gerc	Rank-1 update of a conjugated general matrix
geru	Rank-1 update of a general matrix, unconjugated
hbmv	Matrix-vector product using a Hermitian band matrix
hemv	Matrix-vector product using a Hermitian matrix
her	Rank-1 update of a Hermitian matrix
her2	Rank-2 update of a Hermitian matrix
hpmv	Matrix-vector product using a Hermitian packed matrix
hpr	Rank-1 update of a Hermitian packed matrix
hpr2	Rank-2 update of a Hermitian packed matrix
sbmv	Matrix-vector product using symmetric band matrix
spmv	Matrix-vector product using a symmetric packed matrix
spr	Rank-1 update of a symmetric packed matrix
spr2	Rank-2 update of a symmetric packed matrix
symv	Matrix-vector product using a symmetric matrix
syr	Rank-1 update of a symmetric matrix
syr2	Rank-2 update of a symmetric matrix
tbmv	Matrix-vector product using a triangular band matrix
tbsv	Solution of a linear system of equations with a triangular band matrix
tpmv	Matrix-vector product using a triangular packed matrix
tpsv	Solution of a linear system of equations with a triangular packed matrix
trmv	Matrix-vector product using a triangular matrix
trsv	Solution of a linear system of equations with a triangular matrix

gbmv

Computes a matrix-vector product with a general band matrix.

Description

The `gbmv` routines compute a scalar-matrix-vector product and add the result to a scalar-vector product, with a general band matrix. The operation is defined as

$$y \leftarrow alpha * op(A) * x + beta * y$$

where:

op(`A`) is one of op(`A`) = `A`, or op(`A`) = $\mathtt{A}^{\mathrm{T}}$, or op(`A`) = $\mathtt{A}^{\mathrm{H}}$,

`alpha` and `beta` are scalars,

`A` is an `m`-by-`n` matrix with `kl` sub-diagonals and `ku` super-diagonals,

`x` and `y` are vectors.

`gbmv` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

gbmv (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void gbmv(sycl::queue &queue,
              oneapi::mkl::transpose trans,
              std::int64_t m,
              std::int64_t n,
              std::int64_t kl,
              std::int64_t ku,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              T beta,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void gbmv(sycl::queue &queue,
              oneapi::mkl::transpose trans,
              std::int64_t m,
              std::int64_t n,
              std::int64_t kl,
              std::int64_t ku,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              T beta,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

Input Parameters

queue
The queue where the routine should be executed.

trans
Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

m
Number of rows of `A`. Must be at least zero.

n
Number of columns of `A`. Must be at least zero.

kl
Number of sub-diagonals of the matrix `A`. Must be at least zero.

ku
Number of super-diagonals of the matrix `A`. Must be at least zero.

alpha
Scaling factor for the matrix-vector product.

a
Buffer holding input matrix `A`. Must have size at least `lda`*`n` if column major layout is used or at least `lda`*`m` if row major layout is used. See *Matrix Storage* for more details.

lda
Leading dimension of matrix `A`. Must be at least (`kl` + `ku` + 1), and positive.

x
Buffer holding input vector `x`. The length `len` of vector `x` is `n` if `A` is not transposed, and `m` if `A` is transposed. The buffer must be of size at least (1 + (`len` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

beta
Scaling factor for vector `y`.

y
Buffer holding input/output vector `y`. The length `len` of vector `y` is `m`, if `A` is not transposed, and `n` if `A` is transposed. The buffer must be of size at least (1 + (`len` - 1)*abs(`incy`)) where `len` is this length. See *Matrix Storage* for more details.

incy
Stride of vector `y`.

Output Parameters

y
Buffer holding the updated vector `y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

gbmv (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event gbmv(sycl::queue &queue,
                     oneapi::mkl::transpose trans,
                     std::int64_t m,
                     std::int64_t n,
                     std::int64_t kl,
                     std::int64_t ku,
                     value_or_pointer<T> alpha,
                     const T *a,
                     std::int64_t lda,
                     const T *x,
                     std::int64_t incx,
                     value_or_pointer<T> beta,
                     T *y,
                     std::int64_t incy,
                     const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event gbmv(sycl::queue &queue,
                     oneapi::mkl::transpose trans,
                     std::int64_t m,
                     std::int64_t n,
                     std::int64_t kl,
                     std::int64_t ku,
                     value_or_pointer<T> alpha,
                     const T *a,
                     std::int64_t lda,
                     const T *x,
                     std::int64_t incx,
                     value_or_pointer<T> beta,
                     T *y,
                     std::int64_t incy,
                     const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

trans
Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

m
Number of rows of `A`. Must be at least zero.

n
Number of columns of `A`. Must be at least zero.

kl
Number of sub-diagonals of the matrix `A`. Must be at least zero.

ku
Number of super-diagonals of the matrix `A`. Must be at least zero.

alpha
Scaling factor for the matrix-vector product. See *Scalar Arguments in BLAS* for more details.

a
Pointer to input matrix `A`. The array holding input matrix `A` must have size at least `lda`*`n` if column major layout is used or at least `lda`*`m` if row major layout is used. See *Matrix Storage* for more details.

lda
Leading dimension of matrix `A`. Must be at least (`kl` + `ku` + 1), and positive.

x
Pointer to input vector `x`. The length `len` of vector `x` is `n` if `A` is not transposed, and `m` if `A` is transposed. The array holding input vector `x` must be of size at least (1 + (`len` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

beta
Scaling factor for vector `y`. See *Scalar Arguments in BLAS* for more details.

y
Pointer to input/output vector `y`. The length `len` of vector `y` is `m`, if `A` is not transposed, and `n` if `A` is transposed. The array holding input/output vector `y` must be of size at least (1 + (`len` - 1)*abs(`incy`)) where `len` is this length. See *Matrix Storage* for more details.

incy
Stride of vector `y`.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

y
Pointer to the updated vector `y`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

gemv

Computes a matrix-vector product using a general matrix.

Description

The `gemv` routines compute a scalar-matrix-vector product and add the result to a scalar-vector product, with a general matrix. The operation is defined as:

$$y \leftarrow alpha * op(A) * x + beta * y$$

where:

op(`A`) is one of op(`A`) = `A`, or op(`A`) = $\mathtt{A}^{T}$, or op(`A`) = $\mathtt{A}^{H}$,

`alpha` and `beta` are scalars,

`A` is an `m`-by-`n` matrix, and `x`, `y` are vectors.

`gemv` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

gemv (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void gemv(sycl::queue &queue,
              oneapi::mkl::transpose trans,
              std::int64_t m,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              T beta,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void gemv(sycl::queue &queue,
              oneapi::mkl::transpose trans,
              std::int64_t m,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              T beta,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

trans
: Specifies `op(A)`, the transposition operation applied to `A`.

m
: Specifies the number of rows of the matrix `A`. The value of `m` must be at least zero.

n
: Specifies the number of columns of the matrix `A`. The value of `n` must be at least zero.

alpha
: Scaling factor for the matrix-vector product.

a
: The buffer holding the input matrix `A`. Must have a size of at least `lda*n` if column major layout is used or at least `lda*m` if row major layout is used. See *Matrix Storage* for more details.

lda
Leading dimension of matrix `A`. Must be positive and at least `m` if column major layout is used or at least `n` if row major layout is used.

x
Buffer holding input vector `x`. The length `len` of vector `x` is `n` if `A` is not transposed, and `m` if `A` is transposed. The buffer must be of size at least (1 + (`len` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
The stride of vector `x`. Must not be zero.

beta
The scaling factor for vector `y`.

y
Buffer holding input/output vector `y`. The length `len` of vector `y` is `m`, if `A` is not transposed, and `n` if `A` is transposed. The buffer must be of size at least (1 + (`len` - 1)*abs(`incy`)) where `len` is this length. See *Matrix Storage* for more details.

incy
The stride of vector `y`. Must not be zero.

Output Parameters

y
The buffer holding updated vector `y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

gemv (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event gemv(sycl::queue &queue,
                     oneapi::mkl::transpose trans,
                     std::int64_t m,
                     std::int64_t n,
                     value_or_pointer<T> alpha,
                     const T *a,
                     std::int64_t lda,
                     const T *x,
```

(continues on next page)

(continued from previous page)

```
                    std::int64_t incx,
                    value_or_pointer<T> beta,
                    T *y,
                    std::int64_t incy,
                    const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event gemv(sycl::queue &queue,
                    oneapi::mkl::transpose trans,
                    std::int64_t m,
                    std::int64_t n,
                    value_or_pointer<T> alpha,
                    const T *a,
                    std::int64_t lda,
                    const T *x,
                    std::int64_t incx,
                    value_or_pointer<T> beta,
                    T *y,
                    std::int64_t incy,
                    const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

trans
Specifies `op(A)`, the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

m
Specifies the number of rows of the matrix `A`. The value of `m` must be at least zero.

n
Specifies the number of columns of the matrix `A`. The value of `n` must be at least zero.

alpha
Scaling factor for the matrix-vector product. See *Scalar Arguments in BLAS* for more details.

a
Pointer to the input matrix `A`. Must have a size of at least `lda*n` if column major layout is used or at least `lda*m` if row major layout is used. See *Matrix Storage* for more details.

lda
Leading dimension of matrix `A`. Must be positive and at least `m` if column major layout is used or at least `n` if row major layout is used.

x
Pointer to the input vector `x`. The length `len` of vector `x` is `n` if `A` is not transposed, and `m` if `A` is transposed. The array holding vector `x` must be of size at least (1 + (`len` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
The stride of vector `x`. Must not be zero.

beta
The scaling factor for vector `y`. See *Scalar Arguments in BLAS* for more details.

y
Pointer to input/output vector `y`. The length `len` of vector `y` is `m`, if `A` is not transposed, and `n` if `A` is transposed. The array holding input/output vector `y` must be of size at least (1 + (`len` - 1)*abs(`incy`)) where `len` is this length. See *Matrix Storage* for more details.

incy
The stride of vector `y`. Must not be zero.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

y
The pointer to updated vector `y`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

ger

Computes a rank-1 update of a general matrix.

Description

The `ger` routines compute a scalar-vector-vector product and add the result to a general matrix. The operation is defined as:

$$A \leftarrow alpha * x * y^T + A$$

where:

`alpha` is scalar,

`A` is an `m`-by-`n` matrix,

`x` is a vector of length `m`,

`y` is a vector of length `n`.

`ger` supports the following precisions.

T
float
double

ger (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void ger(sycl::queue &queue,
             std::int64_t m,
             std::int64_t n,
             T alpha,
             sycl::buffer<T,1> &x,
             std::int64_t incx,
             sycl::buffer<T,1> &y,
             std::int64_t incy,
             sycl::buffer<T,1> &a,
             std::int64_t lda)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void ger(sycl::queue &queue,
             std::int64_t m,
             std::int64_t n,
             T alpha,
             sycl::buffer<T,1> &x,
             std::int64_t incx,
             sycl::buffer<T,1> &y,
             std::int64_t incy,
             sycl::buffer<T,1> &a,
             std::int64_t lda)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

m
: Number of rows of `A`. Must be at least zero.

n
: Number of columns of `A`. Must be at least zero.

alpha
Scaling factor for the matrix-vector product.

x
Buffer holding input vector `x`. The buffer must be of size at least (1 + (`m` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

y
Buffer holding input/output vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`. Must not be zero.

a
Buffer holding input matrix `A`. Must have size at least `lda`*`n` if column major layout is used or at least `lda`*`m` if row major layout is used. See *Matrix Storage* for more details.

lda
Leading dimension of matrix `A`. Must be positive and at least `m` if column major layout is used or at least `n` if row major layout is used.

Output Parameters

a
Buffer holding the updated matrix `A`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

ger (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event ger(sycl::queue &queue,
                    std::int64_t m,
                    std::int64_t n,
                    value_or_pointer<T> alpha,
                    const T *x,
                    std::int64_t incx,
```

(continues on next page)

(continued from previous page)

```
                    const T *y,
                    std::int64_t incy,
                    T *a,
                    std::int64_t lda,
                    const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event ger(sycl::queue &queue,
                    std::int64_t m,
                    std::int64_t n,
                    value_or_pointer<T> alpha,
                    const T *x,
                    std::int64_t incx,
                    const T *y,
                    std::int64_t incy,
                    T *a,
                    std::int64_t lda,
                    const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
: The queue where the routine should be executed.

m
: Number of rows of `A`. Must be at least zero.

n
: Number of columns of `A`. Must be at least zero.

alpha
: Scaling factor for the matrix-vector product. See *Scalar Arguments in BLAS* for more details.

x
: Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`m` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
: Stride of vector `x`. Must not be zero.

y
: Pointer to input/output vector `y`. The array holding input/output vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
: Stride of vector `y`. Must not be zero.

a
: Pointer to input matrix `A`. Must have size at least `lda`*`n` if column major layout is used or at least `lda`*`m` if row major layout is used. See *Matrix Storage* for more details.

lda
: Leading dimension of matrix `A`. Must be positive and at least `m` if column major layout is used or at least `n` if row major layout is used.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

a
Pointer to the updated matrix `A`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

gerc

Computes a rank-1 update (conjugated) of a general complex matrix.

Description

The `gerc` routines compute a scalar-vector-vector product and add the result to a general matrix. The operation is defined as:

$$A \leftarrow alpha * x * y^H + A$$

where:

`alpha` is a scalar,

`A` is an `m`-by-`n` matrix,

`x` is a vector of length `m`,

`y` is vector of length `n`.

`gerc` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

gerc (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void gerc(sycl::queue &queue,
              std::int64_t m,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy,
              sycl::buffer<T,1> &a,
              std::int64_t lda)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void gerc(sycl::queue &queue,
              std::int64_t m,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy,
              sycl::buffer<T,1> &a,
              std::int64_t lda)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

m
: Number of rows of `A`. Must be at least zero.

n
: Number of columns of `A`. Must be at least zero.

alpha
: Scaling factor for the matrix-vector product.

x
: Buffer holding input vector `x`. The buffer must be of size at least (1 + (`m` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
: Stride of vector `x`. Must not be zero.

y
: Buffer holding input/output vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`. Must not be zero.

a
Buffer holding input matrix `A`. Must have size at least `lda*n` if column major layout is used or at least `lda*m` if row major layout is used. See *Matrix Storage* for more details.

lda
Leading dimension of matrix `A`. Must be positive and at least `m` if column major layout is used or at least `n` if row major layout is used.

Output Parameters

a
Buffer holding the updated matrix `A`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

gerc (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event gerc(sycl::queue &queue,
                     std::int64_t m,
                     std::int64_t n,
                     value_or_pointer<T> alpha,
                     const T *x,
                     std::int64_t incx,
                     const T *y,
                     std::int64_t incy,
                     T *a,
                     std::int64_t lda,
                     const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event gerc(sycl::queue &queue,
                     std::int64_t m,
                     std::int64_t n,
                     value_or_pointer<T> alpha,
```

(continues on next page)

(continued from previous page)

```
                        const T *x,
                        std::int64_t incx,
                        const T *y,
                        std::int64_t incy,
                        T *a,
                        std::int64_t lda,
                        const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

m
Number of rows of `A`. Must be at least zero.

n
Number of columns of `A`. Must be at least zero.

alpha
Scaling factor for the matrix-vector product. See *Scalar Arguments in BLAS* for more details.

x
Pointer to the input vector `x`. The array holding input vector `x` must be of size at least (1 + (`m` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

y
Pointer to the input/output vector `y`. The array holding the input/output vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`. Must not be zero.

a
Pointer to input matrix `A`. The array holding input matrix `A`must have size at least `lda`*`n` if column major layout is used or at least `lda`*`m` if row major layout is used. See *Matrix Storage* for more details.

lda
Leading dimension of matrix `A`. Must be positive and at least `m` if column major layout is used or at least `n` if row major layout is used.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

a
Pointer to the updated matrix `A`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

geru

Computes a rank-1 update (unconjugated) of a general complex matrix.

Description

The `geru` routines routines compute a scalar-vector-vector product and add the result to a general matrix. The operation is defined as

$$A \leftarrow alpha * x * y^T + A$$

where:

`alpha` is a scalar,

`A` is an `m`-by-`n` matrix,

`x` is a vector of length `m`,

`y` is a vector of length `n`.

`geru` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

geru (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void geru(sycl::queue &queue,
              std::int64_t m,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy,
              sycl::buffer<T,1> &a,
              std::int64_t lda)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void geru(sycl::queue &queue,
              std::int64_t m,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy,
              sycl::buffer<T,1> &a,
              std::int64_t lda)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

m
: Number of rows of `A`. Must be at least zero.

n
: Number of columns of `A`. Must be at least zero.

alpha
: Scaling factor for the matrix-vector product.

x
: Buffer holding input vector `x`. The buffer must be of size at least (1 + (`m` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
: Stride of vector `x`. Must not be zero.

y
: Buffer holding input/output vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`. Must not be zero.

a
Buffer holding input matrix `A`. Must have size at least `lda*n` if column major layout is used or at least `lda*m` if row major layout is used. See *Matrix Storage* for more details.

lda
Leading dimension of matrix `A`. Must be positive and at least `m` if column major layout is used or at least `n` if row major layout is used.

Output Parameters

a
Buffer holding the updated matrix `A`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

geru (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event geru(sycl::queue &queue,
                     std::int64_t m,
                     std::int64_t n,
                     value_or_pointer<T> alpha,
                     const T *x,
                     std::int64_t incx,
                     const T *y,
                     std::int64_t incy,
                     T *a,
                     std::int64_t lda,
                     const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event geru(sycl::queue &queue,
                     std::int64_t m,
                     std::int64_t n,
                     value_or_pointer<T> alpha,
```

(continues on next page)

(continued from previous page)

```
                        const T *x,
                        std::int64_t incx,
                        const T *y,
                        std::int64_t incy,
                        T *a,
                        std::int64_t lda,
                        const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

m
Number of rows of `A`. Must be at least zero.

n
Number of columns of `A`. Must be at least zero.

alpha
Scaling factor for the matrix-vector product. See *Scalar Arguments in BLAS* for more details.

x
Pointer to the input vector `x`. The array holding input vector `x` must be of size at least (1 + (`m` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

y
Pointer to input/output vector `y`. The array holding input/output vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`. Must not be zero.

a
Pointer to input matrix `A`. The array holding input matrix `A` must have size at least `lda`*`n` if column major layout is used or at least `lda`*`m` if row major layout is used. See *Matrix Storage* for more details.

lda
Leading dimension of matrix `A`. Must be positive and at least `m` if column major layout is used or at least `n` if row major layout is used.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

a
Pointer to the updated matrix `A`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

hbmv

Computes a matrix-vector product using a Hermitian band matrix.

Description

The `hbmv` routines compute a scalar-matrix-vector product and add the result to a scalar-vector product, with a Hermitian band matrix. The operation is defined as

$$y \leftarrow alpha * A * x + beta * y$$

where:

`alpha` and `beta` are scalars,

`A` is an `n`-by-`n` Hermitian band matrix, with `k` super-diagonals,

`x` and `y` are vectors of length `n`.

`hbmv` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

hbmv (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void hbmv(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              std::int64_t n,
              std::int64_t k,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              T beta,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void hbmv(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              std::int64_t n,
              std::int64_t k,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              T beta,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n
Number of rows and columns of `A`. Must be at least zero.

k
Number of super-diagonals of the matrix `A`. Must be at least zero.

alpha
Scaling factor for the matrix-vector product.

a
Buffer holding input matrix `A`. Must have size at least `lda*n`. See *Matrix Storage* for more details.

lda
Leading dimension of matrix `A`. Must be at least (`k` + 1), and positive.

x
Buffer holding input vector `x`. The buffer must be of size at least (1 + (`m` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

beta
Scaling factor for vector `y`.

y
Buffer holding input/output vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`. Must not be zero.

Output Parameters

y
Buffer holding the updated vector `y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

hbmv (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event hbmv(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     std::int64_t n,
                     std::int64_t k,
                     value_or_pointer<T> alpha,
                     const T *a,
                     std::int64_t lda,
                     const T *x,
                     std::int64_t incx,
                     value_or_pointer<T> beta,
```

(continues on next page)

(continued from previous page)

```
                    T *y,
                    std::int64_t incy,
                    const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event hbmv(sycl::queue &queue,
                    oneapi::mkl::uplo upper_lower,
                    std::int64_t n,
                    std::int64_t k,
                    value_or_pointer<T> alpha,
                    const T *a,
                    std::int64_t lda,
                    const T *x,
                    std::int64_t incx,
                    value_or_pointer<T> beta,
                    T *y,
                    std::int64_t incy,
                    const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n
Number of rows and columns of `A`. Must be at least zero.

k
Number of super-diagonals of the matrix `A`. Must be at least zero.

alpha
Scaling factor for the matrix-vector product. See *Scalar Arguments in BLAS* for more details.

a
Pointer to the input matrix `A`. The array holding input matrix `A` must have size at least `lda*n`. See *Matrix Storage* for more details.

lda
Leading dimension of matrix `A`. Must be at least (`k` + 1), and positive.

x
Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`m` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

beta
Scaling factor for vector `y`. See *Scalar Arguments in BLAS* for more details.

y
Pointer to input/output vector `y`. The array holding input/output vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`. Must not be zero.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

y
Pointer to the updated vector `y`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

hemv

Computes a matrix-vector product using a Hermitian matrix.

Description

The `hemv` routines compute a scalar-matrix-vector product and add the result to a scalar-vector product, with a Hermitian matrix. The operation is defined as

$$y \leftarrow alpha * A * x + beta * y$$

where:

`alpha` and `beta` are scalars,

`A` is an `n`-by-`n` Hermitian matrix,

`x` and `y` are vectors of length `n`.

`hemv` supports the following precisions.

T
std::complex<float>
std::complex<double>

hemv (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void hemv(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              T beta,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void hemv(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              T beta,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

upper_lower
: Specifies whether *A* is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n
: Number of rows and columns of `A`. Must be at least zero.

alpha
: Scaling factor for the matrix-vector product.

a
Buffer holding input matrix `A`. Must have size at least `lda`*`n`. See *Matrix Storage* for more details.

lda
Leading dimension of matrix `A`. Must be at least `m`, and positive.

x
Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

beta
Scaling factor for vector `y`.

y
Buffer holding input/output vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`. Must not be zero.

Output Parameters

y
Buffer holding the updated vector `y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

hemv (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event hemv(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     std::int64_t n,
                     value_or_pointer<T> alpha,
                     const T *a,
                     std::int64_t lda,
                     const T *x,
                     std::int64_t incx,
```

(continues on next page)

(continued from previous page)

```
                    value_or_pointer<T> beta,
                    T *y,
                    std::int64_t incy,
                    const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event hemv(sycl::queue &queue,
                    oneapi::mkl::uplo upper_lower,
                    std::int64_t n,
                    value_or_pointer<T> alpha,
                    const T *a,
                    std::int64_t lda,
                    const T *x,
                    std::int64_t incx,
                    value_or_pointer<T> beta,
                    T *y,
                    std::int64_t incy,
                    const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether *A* is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n
Number of rows and columns of `A`. Must be at least zero.

alpha
Scaling factor for the matrix-vector product. See *Scalar Arguments in BLAS* for more details.

a
Pointer to input matrix `A`. The array holding input matrix `A` must have size at least `lda*n`. See *Matrix Storage* for more details.

lda
Leading dimension of matrix `A`. Must be at least `m`, and positive.

x
Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

beta
Scaling factor for vector `y`. See *Scalar Arguments in BLAS* for more details.

y
Pointer to input/output vector `y`. The array holding input/output vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`. Must not be zero.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

y
Pointer to the updated vector `y`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

her

Computes a rank-1 update of a Hermitian matrix.

Description

The `her` routines compute a scalar-vector-vector product and add the result to a Hermitian matrix. The operation is defined as:

$$A \leftarrow alpha * x * x^H + A$$

where:

`alpha` is scalar,

`A` is an `n`-by-`n` Hermitian matrix,

`x` is a vector of length `n`.

`her` supports the following precisions.

T	Treal
std::complex<float>	float
std::complex<double>	double

her (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void her(sycl::queue &queue,
             oneapi::mkl::uplo upper_lower,
             std::int64_t n,
             Treal alpha,
             sycl::buffer<T,1> &x,
             std::int64_t incx,
             sycl::buffer<T,1> &a,
             std::int64_t lda)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void her(sycl::queue &queue,
             oneapi::mkl::uplo upper_lower,
             std::int64_t n,
             Treal alpha,
             sycl::buffer<T,1> &x,
             std::int64_t incx,
             sycl::buffer<T,1> &a,
             std::int64_t lda)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

upper_lower
: Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n
: Number of rows and columns of `A`. Must be at least zero.

alpha
: Scaling factor for the matrix-vector product.

x
: Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
: Stride of vector `x`. Must not be zero.

a

Buffer holding input matrix `A`. Must have size at least `lda*n`. See *Matrix Storage* for more details.

lda

Leading dimension of matrix `A`. Must be at least `n`, and positive.

Output Parameters

a

Buffer holding the updated upper triangular part of the Hermitian matrix `A` if `upper_lower = upper` or the updated lower triangular part of the Hermitian matrix `A` if `upper_lower=lower`.

If `alpha` is zero, `A` matrix is unchanged, otherwise imaginary parts of the diagonal elements are set to zero.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

her (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event her(sycl::queue &queue,
                    oneapi::mkl::uplo upper_lower,
                    std::int64_t n,
                    value_or_pointer<Treal> alpha,
                    const T *x,
                    std::int64_t incx,
                    T *a,
                    std::int64_t lda,
                    const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event her(sycl::queue &queue,
                    oneapi::mkl::uplo upper_lower,
                    std::int64_t n,
                    value_or_pointer<Treal> alpha,
                    const T *x,
                    std::int64_t incx,
                    T *a,
                    std::int64_t lda,
```

(continues on next page)

(continued from previous page)

```
                    const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether *A* is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n
Number of rows and columns of `A`. Must be at least zero.

alpha
Scaling factor for the matrix-vector product. See *Scalar Arguments in BLAS* for more details.

x
Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

a
Pointer to input matrix `A`. The array holding input matrix `A` must have size at least `lda`*`n`. See *Matrix Storage* for more details.

lda
Leading dimension of matrix `A`. Must be at least `n`, and positive.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

a
Pointer to the updated upper triangular part of the Hermitian matrix `A` if `upper_lower=upper` or the updated lower triangular part of the Hermitian matrix `A` if `upper_lower=lower`.

If `alpha` is zero, `A` matrix is unchanged, otherwise imaginary parts of the diagonal elements are set to zero.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

her2

Computes a rank-2 update of a Hermitian matrix.

Description

The `her2` routines compute two scalar-vector-vector products and add them to a Hermitian matrix. The operation is defined as:

$$A \leftarrow alpha * x * y^H + conjg(alpha) * y * x^H + A$$

where:

`alpha` is a scalar,

`A` is an `n`-by-`n` Hermitian matrix,

`x` and `y` are vectors or length `n`.

`her2` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

her2 (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void her2(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
```

(continues on next page)

(continued from previous page)

```
                std::int64_t incy,
                sycl::buffer<T,1> &a,
                std::int64_t lda)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void her2(sycl::queue &queue,
                oneapi::mkl::uplo upper_lower,
                std::int64_t n,
                T alpha,
                sycl::buffer<T,1> &x,
                std::int64_t incx,
                sycl::buffer<T,1> &y,
                std::int64_t incy,
                sycl::buffer<T,1> &a,
                std::int64_t lda)
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n
Number of columns of `A`. Must be at least zero.

alpha
Scaling factor for the matrix-vector product.

x
Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

y
Buffer holding input/output vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`. Must not be zero.

a
Buffer holding input matrix `A`. Must have size at least `lda`*`n`. See *Matrix Storage* for more details.

lda
Leading dimension of matrix `A`. Must be at least `n`, and positive.

Output Parameters

a

Buffer holding the updated upper triangular part of the Hermitian matrix `A` if `upper_lower=upper`, or the updated lower triangular part of the Hermitian matrix `A` if `upper_lower=lower`.

If `alpha` is zero, `A` matrix is unchanged, otherwise imaginary parts of the diagonal elements are set to zero.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

her2 (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event her2(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     std::int64_t n,
                     value_or_pointer<T> alpha,
                     const T *x,
                     std::int64_t incx,
                     const T *y,
                     std::int64_t incy,
                     T *a,
                     std::int64_t lda,
                     const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event her2(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     std::int64_t n,
                     value_or_pointer<T> alpha,
                     const T *x,
                     std::int64_t incx,
                     const T *y,
                     std::int64_t incy,
                     T *a,
                     std::int64_t lda,
                     const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n
Number of columns of `A`. Must be at least zero.

alpha
Scaling factor for the matrix-vector product. See *Scalar Arguments in BLAS* for more details.

x
Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

y
Pointer to input/output vector `y`. The array holding input/output vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`. Must not be zero.

a
Pointer to input matrix `A`. The array holding input matrix `A` must have size at least `lda`*`n`. See *Matrix Storage* for more details.

lda
Leading dimension of matrix `A`. Must be at least `n`, and positive.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

a
Pointer to the updated upper triangular part of the Hermitian matrix `A` if `upper_lower=upper`, or the updated lower triangular part of the Hermitian matrix `A` if `upper_lower=lower`.

If `alpha` is zero, `A` matrix is unchanged, otherwise imaginary parts of the diagonal elements are set to zero.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

hpmv

Computes a matrix-vector product using a Hermitian packed matrix.

Description

The hpmv routines compute a scalar-matrix-vector product and add the result to a scalar-vector product, with a Hermitian packed matrix. The operation is defined as

$$y \leftarrow alpha * A * x + beta * y$$

where:

alpha and beta are scalars,

A is an n-by-n Hermitian matrix supplied in packed form,

x and y are vectors of length n.

hpmv supports the following precisions.

T
std::complex<float>
std::complex<double>

hpmv (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void hpmv(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
```

(continues on next page)

(continued from previous page)

```
                T beta,
                sycl::buffer<T,1> &y,
                std::int64_t incy)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void hpmv(sycl::queue &queue,
                oneapi::mkl::uplo upper_lower,
                std::int64_t n,
                T alpha,
                sycl::buffer<T,1> &a,
                sycl::buffer<T,1> &x,
                std::int64_t incx,
                T beta,
                sycl::buffer<T,1> &y,
                std::int64_t incy)
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n
Number of rows and columns of `A`. Must be at least zero.

alpha
Scaling factor for the matrix-vector product.

a
Buffer holding input matrix `A`. Must have size at least (`n`*(`n`+1))/2. See *Matrix Storage* for more details.

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

x
Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

beta
Scaling factor for vector `y`.

y
Buffer holding input/output vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`. Must not be zero.

Output Parameters

y
Buffer holding the updated vector y.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

hpmv (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event hpmv(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     std::int64_t n,
                     value_or_pointer<T> alpha,
                     const T *a,
                     const T *x,
                     std::int64_t incx,
                     value_or_pointer<T> beta,
                     T *y,
                     std::int64_t incy,
                     const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event hpmv(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     std::int64_t n,
                     value_or_pointer<T> alpha,
                     const T *a,
                     const T *x,
                     std::int64_t incx,
                     value_or_pointer<T> beta,
                     T *y,
                     std::int64_t incy,
                     const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n
Number of rows and columns of `A`. Must be at least zero.

alpha
Scaling factor for the matrix-vector product. See *Scalar Arguments in BLAS* for more details.

a
Pointer to input matrix `A`. The array holding input matrix `A` must have size at least (`n`*(`n`+1))/2. See *Matrix Storage* for more details.

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

x
Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

beta
Scaling factor for vector `y`. See *Scalar Arguments in BLAS* for more details.

y
Pointer to input/output vector `y`. The array holding input/output vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`. Must not be zero.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

y
Pointer to the updated vector `y`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

hpr

Computes a rank-1 update of a Hermitian packed matrix.

Description

The `hpr` routines compute a scalar-vector-vector product and add the result to a Hermitian packed matrix. The operation is defined as

$$A \leftarrow alpha * x * x^H + A$$

where:

`alpha` is scalar,

`A` is an `n`-by-`n` Hermitian matrix, supplied in packed form,

`x` is a vector of length `n`.

`hpr` supports the following precisions.

T	Treal
`std::complex<float>`	`float`
`std::complex<double>`	`double`

hpr (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void hpr(sycl::queue &queue,
             oneapi::mkl::uplo upper_lower,
             std::int64_t n,
             Treal alpha,
             sycl::buffer<T,1> &x,
             std::int64_t incx,
             sycl::buffer<T,1> &a)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void hpr(sycl::queue &queue,
             oneapi::mkl::uplo upper_lower,
             std::int64_t n,
             Treal alpha,
             sycl::buffer<T,1> &x,
             std::int64_t incx,
             sycl::buffer<T,1> &a)
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n
Number of rows and columns of `A`. Must be at least zero.

alpha
Scaling factor for the matrix-vector product.

x
Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

a
Buffer holding input matrix `A`. Must have size at least (`n`*(`n`-1))/2. See *Matrix Storage* for more details.

The imaginary part of the diagonal elements need not be set and are assumed to be zero.

Output Parameters

a
Buffer holding the updated upper triangular part of the Hermitian matrix `A` if `upper_lower=upper`, or the updated lower triangular part of the Hermitian matrix `A` if `upper_lower=lower`.

If `alpha` is zero, `A` matrix is unchanged, otherwise imaginary parts of the diagonal elements are set to zero.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

hpr (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event hpr(sycl::queue &queue,
                    oneapi::mkl::uplo upper_lower,
                    std::int64_t n,
                    value_or_pointer<Treal> alpha,
                    const T *x,
                    std::int64_t incx,
                    T *a,
                    const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event hpr(sycl::queue &queue,
                    oneapi::mkl::uplo upper_lower,
                    std::int64_t n,
                    value_or_pointer<Treal> alpha,
                    const T *x,
                    std::int64_t incx,
                    T *a,
                    const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n
Number of rows and columns of `A`. Must be at least zero.

alpha
Scaling factor for the matrix-vector product. See *Scalar Arguments in BLAS* for more details.

x
Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

a
Pointer to input matrix `A`. The array holding input matrix `A` must have size at least (`n`*(`n`-1))/2. See *Matrix Storage* for more details.

The imaginary part of the diagonal elements need not be set and are assumed to be zero.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

a
Pointer to the updated upper triangular part of the Hermitian matrix `A` if `upper_lower=upper`, or the updated lower triangular part of the Hermitian matrix `A` if `upper_lower=lower`.

If `alpha` is zero, `A` matrix is unchanged, otherwise imaginary parts of the diagonal elements are set to zero.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

hpr2

Performs a rank-2 update of a Hermitian packed matrix.

Description

The `hpr2` routines compute two scalar-vector-vector products and add them to a Hermitian packed matrix. The operation is defined as

$$A \leftarrow alpha * x * y^H + conjg(alpha) * y * x^H + A$$

where:

`alpha` is a scalar,

`A` is an `n`-by-`n` Hermitian matrix, supplied in packed form,

`x` and `y` are vectors of length `n`.

`hpr2` supports the following precisions.

T
std::complex<float>
std::complex<double>

hpr2 (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void hpr2(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy,
              sycl::buffer<T,1> &a)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void hpr2(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy,
              sycl::buffer<T,1> &a)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

upper_lower
: Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n
: Number of rows and columns of `A`. Must be at least zero.

alpha
: Scaling factor for the matrix-vector product.

x
: Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
: Stride of vector `x`. Must not be zero.

y

Buffer holding input/output vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy

Stride of vector `y`. Must not be zero.

a

Buffer holding input matrix `A`. Must have size at least (`n`*(`n`-1))/2. See *Matrix Storage* for more details.

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

Output Parameters

a

Buffer holding the updated upper triangular part of the Hermitian matrix `A` if `upper_lower=upper`, or the updated lower triangular part of the Hermitian matrix `A` if `upper_lower=lower`.

If `alpha` is zero, `A` matrix is unchanged, otherwise imaginary parts of the diagonal elements are set to zero.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

hpr2 (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event hpr2(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     std::int64_t n,
                     value_or_pointer<T> alpha,
                     const T *x,
                     std::int64_t incx,
                     const T *y,
                     std::int64_t incy,
                     T *a,
                     const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event hpr2(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
```

(continues on next page)

(continued from previous page)

```
                        std::int64_t n,
                        value_or_pointer<T> alpha,
                        const T *x,
                        std::int64_t incx,
                        const T *y,
                        std::int64_t incy,
                        T *a,
                        const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n
Number of rows and columns of `A`. Must be at least zero.

alpha
Scaling factor for the matrix-vector product. See *Scalar Arguments in BLAS* for more details.

x
Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

y
Pointer to input/output vector `y`. The array holding input/output vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`. Must not be zero.

a
Pointer to input matrix `A`. The array holding input matrix `A` must have size at least (`n`*(`n`-1))/2. See *Matrix Storage* for more details.

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

a

Pointer to the updated upper triangular part of the Hermitian matrix `A` if `upper_lower=upper`, or the updated lower triangular part of the Hermitian matrix `A` if `upper_lower=lower`.

If `alpha` is zero, `A` matrix is unchanged, otherwise imaginary parts of the diagonal elements are set to zero.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

sbmv

Computes a matrix-vector product with a symmetric band matrix.

Description

The `sbmv` routines compute a scalar-matrix-vector product and add the result to a scalar-vector product, with a symmetric band matrix. The operation is defined as:

$$y \leftarrow alpha * A * x + beta * y$$

where:

`alpha` and `beta` are scalars,

`A` is an `n`-by-`n` symmetric matrix with `k` super-diagonals,

`x` and `y` are vectors of length `n`.

`sbmv` supports the following precisions.

T
float
double

sbmv (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void sbmv(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              std::int64_t n,
              std::int64_t k,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              T beta,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void sbmv(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              std::int64_t n,
              std::int64_t k,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              T beta,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n
Number of rows and columns of `A`. Must be at least zero.

k
Number of super-diagonals of the matrix `A`. Must be at least zero.

alpha
Scaling factor for the matrix-vector product.

a
Buffer holding input matrix `A`. Must have size at least `lda*n`. See *Matrix Storage* for more details.

lda
Leading dimension of matrix `A`. Must be at least (`k` + 1), and positive.

x
Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

beta
Scaling factor for vector `y`.

y
Buffer holding input/output vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`. Must not be zero.

Output Parameters

y
Buffer holding the updated vector `y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

sbmv (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event sbmv(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     std::int64_t n,
                     std::int64_t k,
                     value_or_pointer<T> alpha,
                     const T *a,
                     std::int64_t lda,
                     const T *x,
                     std::int64_t incx,
                     value_or_pointer<T> beta,
```

(continues on next page)

(continued from previous page)

```
                    T *y,
                    std::int64_t incy,
                    const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event sbmv(sycl::queue &queue,
                    oneapi::mkl::uplo upper_lower,
                    std::int64_t n,
                    std::int64_t k,
                    value_or_pointer<T> alpha,
                    const T *a,
                    std::int64_t lda,
                    const T *x,
                    std::int64_t incx,
                    value_or_pointer<T> beta,
                    T *y,
                    std::int64_t incy,
                    const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n
Number of rows and columns of `A`. Must be at least zero.

k
Number of super-diagonals of the matrix `A`. Must be at least zero.

alpha
Scaling factor for the matrix-vector product. See *Scalar Arguments in BLAS* for more details.

a
Pointer to input matrix `A`. The array holding input matrix `A` must have size at least `lda*n`. See *Matrix Storage* for more details.

lda
Leading dimension of matrix `A`. Must be at least (`k` + 1), and positive.

x
Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

beta
Scaling factor for vector `y`. See *Scalar Arguments in BLAS* for more details.

y
Pointer to input/output vector `y`. The array holding input/output vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`. Must not be zero.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

y
Pointer to the updated vector `y`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

spmv

Computes a matrix-vector product with a symmetric packed matrix.

Description

The `spmv` routines compute a scalar-matrix-vector product and add the result to a scalar-vector product, with a symmetric packed matrix. The operation is defined as:

$$y \leftarrow alpha * A * x + beta * y$$

where:

`alpha` and `beta` are scalars,

`A` is an `n`-by-`n` symmetric matrix, supplied in packed form,

`x` and `y` are vectors of length `n`.

`spmv` supports the following precisions.

T
float
double

spmv (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void spmv(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              T beta,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void spmv(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              T beta,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

upper_lower
: Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n
: Number of rows and columns of `A`. Must be at least zero.

alpha
: Scaling factor for the matrix-vector product.

a
: Buffer holding input matrix `A`. Must have size at least (n*(n+1))/2. See *Matrix Storage* for more details.

x
: Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
: Stride of vector `x`. Must not be zero.

beta
: Scaling factor for vector `y`.

y
: Buffer holding input/output vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
: Stride of vector `y`. Must not be zero.

Output Parameters

y
: Buffer holding the updated vector `y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

spmv (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event spmv(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     std::int64_t n,
                     value_or_pointer<T> alpha,
                     const T *a,
                     const T *x,
                     std::int64_t incx,
                     value_or_pointer<T> beta,
                     T *y,
                     std::int64_t incy,
                     const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event spmv(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     std::int64_t n,
                     value_or_pointer<T> alpha,
                     const T *a,
                     const T *x,
                     std::int64_t incx,
                     value_or_pointer<T> beta,
                     T *y,
                     std::int64_t incy,
                     const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n
Number of rows and columns of `A`. Must be at least zero.

alpha
Scaling factor for the matrix-vector product. See *Scalar Arguments in BLAS* for more details.

a
Pointer to input matrix `A`. The array holding input matrix `A` must have size at least (`n`*(`n`+1))/2. See *Matrix Storage* for more details.

x
Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

beta
Scaling factor for vector `y`. See *Scalar Arguments in BLAS* for more details.

y
Pointer to input/output vector `y`. The array holding input/output vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`. Must not be zero.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

y
Pointer to the updated vector `y`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

spr

Performs a rank-1 update of a symmetric packed matrix.

Description

The `spr` routines compute a scalar-vector-vector product and add the result to a symmetric packed matrix. The operation is defined as:

$$A \leftarrow alpha * x * x^T + A$$

where:

`alpha` is scalar,

`A` is an `n`-by-`n` symmetric matrix, supplied in packed form,

`x` is a vector of length `n`.

`spr` supports the following precisions.

T
`float`
`double`

spr (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void spr(sycl::queue &queue,
             oneapi::mkl::uplo upper_lower,
             std::std::int64_t n,
             T alpha,
             sycl::buffer<T,1> &x,
             std::int64_t incx,
             sycl::buffer<T,1> &a)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void spr(sycl::queue &queue,
             oneapi::mkl::uplo upper_lower,
             std::std::int64_t n,
             T alpha,
             sycl::buffer<T,1> &x,
             std::int64_t incx,
             sycl::buffer<T,1> &a)
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n
Number of rows and columns of `A`. Must be at least zero.

alpha
Scaling factor for the matrix-vector product.

x
Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

a
Buffer holding input matrix `A`. Must have size at least (`n`*(`n` + 1))/2. See *Matrix Storage* for more details.

Output Parameters

a

Buffer holding the updated upper triangular part of the symmetric matrix `A` if `upper_lower=upper`, or the updated lower triangular part of the symmetric matrix `A` if `upper_lower=lower`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

spr (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event spr(sycl::queue &queue,
                    oneapi::mkl::uplo upper_lower,
                    std::int64_t n,
                    value_or_pointer<T> alpha,
                    const T *x,
                    std::int64_t incx,
                    T *a,
                    const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event spr(sycl::queue &queue,
                    oneapi::mkl::uplo upper_lower,
                    std::int64_t n,
                    value_or_pointer<T> alpha,
                    const T *x,
                    std::int64_t incx,
                    T *a,
                    const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n
Number of rows and columns of `A`. Must be at least zero.

alpha
Scaling factor for the matrix-vector product. See *Scalar Arguments in BLAS* for more details.

x
Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

a
Pointer to input matrix `A`. The array holding input matrix `A` must have size at least (`n`*(`n` + 1))/2. See *Matrix Storage* for more details.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

a
Pointer to the updated upper triangular part of the symmetric matrix `A` if `upper_lower=upper`, or the updated lower triangular part of the symmetric matrix `A` if `upper_lower=lower`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

spr2

Computes a rank-2 update of a symmetric packed matrix.

Description

The spr2 routines compute two scalar-vector-vector products and add them to a symmetric packed matrix. The operation is defined as:

$$A \leftarrow alpha * x * y^T + alpha * y * x^T + A$$

where:

alpha is scalar,

A is an n-by-n symmetric matrix, supplied in packed form,

x and y are vectors of length n.

spr supports the following precisions.

T
float
double

spr2 (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void spr2(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy,
              sycl::buffer<T,1> &a)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void spr2(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy,
              sycl::buffer<T,1> &a)
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n
Number of rows and columns of `A`. Must be at least zero.

alpha
Scaling factor for the matrix-vector product.

x
Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

y
Buffer holding input/output vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`. Must not be zero.

a
Buffer holding input matrix `A`. Must have size at least (`n`*(`n`-1))/2. See *Matrix Storage* for more details.

Output Parameters

a
Buffer holding the updated upper triangular part of the symmetric matrix `A` if `upper_lower=upper` or the updated lower triangular part of the symmetric matrix `A` if `upper_lower=lower`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

spr2 (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event spr2(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     std::int64_t n,
                     value_or_pointer<T> alpha,
                     const T *x,
                     std::int64_t incx,
                     const T *y,
                     std::int64_t incy,
                     T *a,
                     const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event spr2(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     std::int64_t n,
                     value_or_pointer<T> alpha,
                     const T *x,
                     std::int64_t incx,
                     const T *y,
                     std::int64_t incy,
                     T *a,
                     const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
: The queue where the routine should be executed.

upper_lower
: Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n
: Number of rows and columns of `A`. Must be at least zero.

alpha
: Scaling factor for the matrix-vector product. See *Scalar Arguments in BLAS* for more details.

x
: Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
: Stride of vector `x`. Must not be zero.

y
: Pointer to input/output vector `y`. The array holding input/output vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`. Must not be zero.

a
Pointer to input matrix `A`. The array holding input matrix `A` must have size at least (`n`*(`n`-1))/2. See *Matrix Storage* for more details.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

a
Pointer to the updated upper triangular part of the symmetric matrix `A` if `upper_lower=upper` or the updated lower triangular part of the symmetric matrix `A` if `upper_lower=lower`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

symv

Computes a matrix-vector product for a symmetric matrix.

Description

The `symv` routines routines compute a scalar-matrix-vector product and add the result to a scalar-vector product, with a symmetric matrix. The operation is defined as:

$$y \leftarrow alpha * A * x + beta * y$$

where:

`alpha` and `beta` are scalars,

`A` is an `n`-by-`n` symmetric matrix,

`x` and `y` are vectors of length `n`.

`symv` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

symv (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void symv(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              T beta,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void symv(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              T beta,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

upper_lower
: Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n
: Number of rows and columns of `A`. Must be at least zero.

alpha
: Scaling factor for the matrix-vector product.

a
Buffer holding input matrix `A`. Must have size at least `lda`*`n`. See *Matrix Storage* for more details.

lda
Leading dimension of matrix `A`. Must be at least `m`, and positive.

x
Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

beta
Scaling factor for the vector `y`.

y
Buffer holding input/output vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`. Must not be zero.

Output Parameters

y
Buffer holding the updated vector `y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

symv (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event symv(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     std::int64_t n,
                     value_or_pointer<T> alpha,
                     const T *a,
                     std::int64_t lda,
                     const T *x,
                     std::int64_t incx,
```

(continues on next page)

(continued from previous page)

```
                    value_or_pointer<T> beta,
                    T *y,
                    std::int64_t incy,
                    const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event symv(sycl::queue &queue,
                    oneapi::mkl::uplo upper_lower,
                    std::int64_t n,
                    value_or_pointer<T> alpha,
                    const T *a,
                    std::int64_t lda,
                    const T *x,
                    std::int64_t incx,
                    value_or_pointer<T> beta,
                    T *y,
                    std::int64_t incy,
                    const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n
Number of rows and columns of `A`. Must be at least zero.

alpha
Scaling factor for the matrix-vector product. See *Scalar Arguments in BLAS* for more details.

a
Pointer to input matrix `A`. The array holding input matrix `A` must have size at least `lda*n`. See *Matrix Storage* for more details.

lda
Leading dimension of matrix `A`. Must be at least `m`, and positive.

x
Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

beta
Scaling factor for the vector `y`. See *Scalar Arguments in BLAS* for more details.

y
Pointer to input/output vector `y`. The array holding input/output vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`. Must not be zero.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

y
Pointer to the updated vector `y`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

syr

Computes a rank-1 update of a symmetric matrix.

Description

The `syr` routines compute a scalar-vector-vector product add them and add the result to a matrix, with a symmetric matrix. The operation is defined as:

$$A \leftarrow alpha * x * x^T + A$$

where:

`alpha` is scalar,

`A` is an `n`-by-`n` symmetric matrix,

`x` is a vector of length `n`.

`syr` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

syr (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void syr(sycl::queue &queue,
             oneapi::mkl::uplo upper_lower,
             std::int64_t n,
             T alpha,
             sycl::buffer<T,1> &x,
             std::int64_t incx,
             sycl::buffer<T,1> &a,
             std::int64_t lda)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void syr(sycl::queue &queue,
             oneapi::mkl::uplo upper_lower,
             std::int64_t n,
             T alpha,
             sycl::buffer<T,1> &x,
             std::int64_t incx,
             sycl::buffer<T,1> &a,
             std::int64_t lda)
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n
Number of columns of `A`. Must be at least zero.

alpha
Scaling factor for the matrix-vector product.

x
Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

a
: Buffer holding input matrix `A`. Must have size at least `lda*n`. See *Matrix Storage* for more details.

lda
: Leading dimension of matrix `A`. Must be at least `n`, and positive.

Output Parameters

a
: Buffer holding the updated upper triangular part of the symmetric matrix `A` if `upper_lower=upper` or the updated lower triangular part of the symmetric matrix `A` if `upper_lower=lower`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

syr (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event syr(sycl::queue &queue,
                    oneapi::mkl::uplo upper_lower,
                    std::int64_t n,
                    value_or_pointer<T> alpha,
                    const T *x,
                    std::int64_t incx,
                    T *a,
                    std::int64_t lda,
                    const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event syr(sycl::queue &queue,
                    oneapi::mkl::uplo upper_lower,
                    std::int64_t n,
                    value_or_pointer<T> alpha,
                    const T *x,
                    std::int64_t incx,
                    T *a,
                    std::int64_t lda,
                    const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n
Number of columns of `A`. Must be at least zero.

alpha
Scaling factor for the matrix-vector product. See *Scalar Arguments in BLAS* for more details.

x
Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

a
Pointer to input matrix `A`. The array holding input matrix `A` must have size at least `lda`*`n`. See *Matrix Storage* for more details.

lda
Leading dimension of matrix `A`. Must be at least `n`, and positive.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

a
Pointer to the updated upper triangular part of the symmetric matrix `A` if `upper_lower=upper` or the updated lower triangular part of the symmetric matrix `A` if `upper_lower=lower`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

syr2

Computes a rank-2 update of a symmetric matrix.

Description

The `syr2` routines compute two scalar-vector-vector product add them and add the result to a matrix, with a symmetric matrix. The operation is defined as:

$$A \leftarrow alpha * x * y^T + alpha * y * x^T + A$$

where:

`alpha` is a scalar,

`A` is an `n`-by-`n` symmetric matrix,

`x` and `y` are vectors of length `n`.

`syr2` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

syr2 (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void syr2(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy,
              sycl::buffer<T,1> &a,
              std::int64_t lda)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void syr2(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
```

(continues on next page)

(continued from previous page)

```
                sycl::buffer<T,1> &y,
                std::int64_t incy,
                sycl::buffer<T,1> &a,
                std::int64_t lda)
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n
Number of columns of `A`. Must be at least zero.

alpha
Scaling factor for the matrix-vector product.

x
Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

y
Buffer holding input/output vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`. Must not be zero.

a
Buffer holding input matrix `A`. Must have size at least `lda`*`n`. See *Matrix Storage* for more details.

lda
Leading dimension of matrix `A`. Must be at least `n`, and positive.

Output Parameters

a
Buffer holding the updated upper triangular part of the symmetric matrix `A` if `upper_lower=upper`, or the updated lower triangular part of the symmetric matrix `A` if `upper_lower=lower`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

syr2 (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event syr2(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     std::int64_t n,
                     value_or_pointer<T> alpha,
                     const T *x,
                     std::int64_t incx,
                     const T *y,
                     std::int64_t incy,
                     T *a,
                     std::int64_t lda,
                     const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event syr2(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     std::int64_t n,
                     value_or_pointer<T> alpha,
                     const T *x,
                     std::int64_t incx,
                     const T *y,
                     std::int64_t incy,
                     T *a,
                     std::int64_t lda,
                     const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n
Number of columns of `A`. Must be at least zero.

alpha
Scaling factor for the matrix-vector product. See *Scalar Arguments in BLAS* for more details.

x
Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

y
Pointer to input/output vector `y`. The array holding input/output vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride of vector `y`. Must not be zero.

a
Pointer to input matrix `A`. The array holding input matrix `A` must have size at least `lda`*`n`. See *Matrix Storage* for more details.

lda
Leading dimension of matrix `A`. Must be at least `n`, and positive.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

a
Pointer to the updated upper triangular part of the symmetric matrix `A` if `upper_lower=upper`, or the updated lower triangular part of the symmetric matrix `A` if `upper_lower=lower`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

tbmv

Computes a matrix-vector product using a triangular band matrix.

Description

The `tbmv` routines compute a matrix-vector product with a triangular band matrix. The operation is defined as:

$$x \leftarrow op(A) * x$$

where:

op(`A`) is one of op(`A`) = `A`, or op(`A`) = $\mathtt{A}^{\mathrm{T}}$, or op(`A`) = $\mathtt{A}^{\mathrm{H}}$,

`A` is an `n`-by-`n` unit or non-unit, upper or lower triangular band matrix, with (`k` + 1) diagonals,

`x` is a vector of length `n`.

`tbmv` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

tbmv (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void tbmv(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              oneapi::mkl::transpose trans,
              oneapi::mkl::diag unit_nonunit,
              std::int64_t n,
              std::int64_t k,
```

(continues on next page)

(continued from previous page)

```
            sycl::buffer<T,1> &a,
            std::int64_t lda,
            sycl::buffer<T,1> &x,
            std::int64_t incx)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void tbmv(sycl::queue &queue,
            oneapi::mkl::uplo upper_lower,
            oneapi::mkl::transpose trans,
            oneapi::mkl::diag unit_nonunit,
            std::int64_t n,
            std::int64_t k,
            sycl::buffer<T,1> &a,
            std::int64_t lda,
            sycl::buffer<T,1> &x,
            std::int64_t incx)
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans
Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_nonunit
Specifies whether the matrix `A` is unit triangular or not. See *oneMKL defined datatypes* for more details.

n
Numbers of rows and columns of `A`. Must be at least zero.

k
Number of sub/super-diagonals of the matrix `A`. Must be at least zero.

a
Buffer holding input matrix `A`. Must have size at least `lda`*`n`. See *Matrix Storage* for more details.

lda
Leading dimension of matrix `A`. Must be at least (`k` + 1), and positive.

x
Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

Output Parameters

x
Buffer holding the updated vector x.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

tbmv (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event tbmv(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     oneapi::mkl::transpose trans,
                     oneapi::mkl::diag unit_nonunit,
                     std::int64_t n,
                     std::int64_t k,
                     const T *a,
                     std::int64_t lda,
                     T *x,
                     std::int64_t incx,
                     const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event tbmv(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     oneapi::mkl::transpose trans,
                     oneapi::mkl::diag unit_nonunit,
                     std::int64_t n,
                     std::int64_t k,
                     const T *a,
                     std::int64_t lda,
                     T *x,
                     std::int64_t incx,
                     const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans
Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_nonunit
Specifies whether the matrix `A` is unit triangular or not. See *oneMKL defined datatypes* for more details.

n
Numbers of rows and columns of `A`. Must be at least zero.

k
Number of sub/super-diagonals of the matrix `A`. Must be at least zero.

a
Pointer to input matrix `A`. The array holding input matrix `A` must have size at least `lda`*`n`. See *Matrix Storage* for more details.

lda
Leading dimension of matrix `A`. Must be at least (`k` + 1), and positive.

x
Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

x
Pointer to the updated vector `x`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

tbsv

Solves a system of linear equations whose coefficients are in a triangular band matrix.

Description

The `tbsv` routines solve a system of linear equations whose coefficients are in a triangular band matrix. The operation is defined as:

$$op(A) * x = b$$

where:

op(`A`) is one of op(`A`) = `A`, or op(`A`) = $\mathtt{A}^T$, or op(`A`) = $\mathtt{A}^H$,

`A` is an `n`-by-`n` unit or non-unit, upper or lower triangular band matrix, with (`k` + 1) diagonals,

`b` and `x` are vectors of length `n`.

`tbsv` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

tbsv (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void tbsv(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              oneapi::mkl::transpose trans,
              oneapi::mkl::diag unit_nonunit,
              std::int64_t n,
              std::int64_t k,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void tbsv(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
```

(continues on next page)

(continued from previous page)

```
                oneapi::mkl::transpose trans,
                oneapi::mkl::diag unit_nonunit,
                std::int64_t n,
                std::int64_t k,
                sycl::buffer<T,1> &a,
                std::int64_t lda,
                sycl::buffer<T,1> &x,
                std::int64_t incx)
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans
Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_nonunit
Specifies whether the matrix `A` is unit triangular or not. See *oneMKL defined datatypes* for more details.

n
Number of rows and columns of `A`. Must be at least zero.

k
Number of sub/super-diagonals of the matrix `A`. Must be at least zero.

a
Buffer holding input matrix `A`. Must have size at least `lda`*`n`. See *Matrix Storage* for more details.

lda
Leading dimension of matrix `A`. Must be at least (`k` + 1), and positive.

x
Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

Output Parameters

x
Buffer holding the solution vector `x`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

tbsv (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event tbsv(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     oneapi::mkl::transpose trans,
                     oneapi::mkl::diag unit_nonunit,
                     std::int64_t n,
                     std::int64_t k,
                     const T *a,
                     std::int64_t lda,
                     T *x,
                     std::int64_t incx,
                     const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event tbsv(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     oneapi::mkl::transpose trans,
                     oneapi::mkl::diag unit_nonunit,
                     std::int64_t n,
                     std::int64_t k,
                     const T *a,
                     std::int64_t lda,
                     T *x,
                     std::int64_t incx,
                     const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans
Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_nonunit
Specifies whether the matrix `A` is unit triangular or not. See *oneMKL defined datatypes* for more details.

n
Number of rows and columns of `A`. Must be at least zero.

k
Number of sub/super-diagonals of the matrix `A`. Must be at least zero.

a
Pointer to input matrix `A`. The array holding input matrix `A` must have size at least `lda`*`n`. See *Matrix Storage* for more details.

lda
Leading dimension of matrix `A`. Must be at least (`k` + 1), and positive.

x
Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

x
Pointer to the solution vector `x`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

tpmv

Computes a matrix-vector product using a triangular packed matrix.

Description

The `tpmv` routines compute a matrix-vector product with a triangular packed matrix. The operation is defined as:

$$x \leftarrow op(A) * x$$

where:

op(`A`) is one of op(`A`) = `A`, or op(`A`) = $\mathtt{A}^{\mathrm{T}}$, or op(`A`) = $\mathtt{A}^{\mathrm{H}}$,

`A` is an `n`-by-`n` unit or non-unit, upper or lower triangular band matrix, supplied in packed form,

`x` is a vector of length `n`.

`tpmv` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

tpmv (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void tpmv(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              oneapi::mkl::transpose trans,
              oneapi::mkl::diag unit_nonunit,
              std::int64_t n,
              sycl::buffer<T,1> &a,
              sycl::buffer<T,1> &x,
              std::int64_t incx)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void tpmv(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              oneapi::mkl::transpose trans,
              oneapi::mkl::diag unit_nonunit,
              std::int64_t n,
```

(continues on next page)

(continued from previous page)

```
                sycl::buffer<T,1> &a,
                sycl::buffer<T,1> &x,
                std::int64_t incx)
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans
Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_nonunit
Specifies whether the matrix `A` is unit triangular or not. See *oneMKL defined datatypes* for more details.

n
Numbers of rows and columns of `A`. Must be at least zero.

a
Buffer holding input matrix `A`. Must have size at least (`n`*(`n`+1))/2. See *Matrix Storage* for more details.

x
Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

Output Parameters

x
Buffer holding the updated vector `x`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

tpmv (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event tpmv(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     oneapi::mkl::transpose trans,
                     oneapi::mkl::diag unit_nonunit,
                     std::int64_t n,
                     const T *a,
                     T *x,
                     std::int64_t incx,
                     const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event tpmv(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     oneapi::mkl::transpose trans,
                     oneapi::mkl::diag unit_nonunit,
                     std::int64_t n,
                     const T *a,
                     T *x,
                     std::int64_t incx,
                     const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
: The queue where the routine should be executed.

upper_lower
: Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans
: Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_nonunit
: Specifies whether the matrix `A` is unit triangular or not. See *oneMKL defined datatypes* for more details.

n
: Numbers of rows and columns of `A`. Must be at least zero.

a
: Pointer to input matrix `A`. The array holding input matrix `A` must have size at least (`n`*(`n`+1))/2. See *Matrix Storage* for more details.

x
: Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
: Stride of vector `x`. Must not be zero.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

x
Pointer to the updated vector `x`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

tpsv

Solves a system of linear equations whose coefficients are in a triangular packed matrix.

Description

The `tpsv` routines solve a system of linear equations whose coefficients are in a triangular packed matrix. The operation is defined as:

$$op(A) * x = b$$

where:

op(`A`) is one of op(`A`) = `A`, or op(`A`) = $\mathtt{A}^T$, or op(`A`) = $\mathtt{A}^H$,

`A` is an `n`-by-`n` unit or non-unit, upper or lower triangular band matrix, supplied in packed form,

`b` and `x` are vectors of length `n`.

`tpsv` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

tpsv (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void tpsv(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              oneapi::mkl::transpose trans,
              oneapi::mkl::diag unit_nonunit,
              std::int64_t n,
              std::int64_t k,
              sycl::buffer<T,1> &a,
              sycl::buffer<T,1> &x,
              std::int64_t incx)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void tpsv(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              oneapi::mkl::transpose trans,
              oneapi::mkl::diag unit_nonunit,
              std::int64_t n,
              std::int64_t k,
              sycl::buffer<T,1> &a,
              sycl::buffer<T,1> &x,
              std::int64_t incx)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

upper_lower
: Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans
: Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_nonunit
: Specifies whether the matrix `A` is unit triangular or not. See *oneMKL defined datatypes* for more details.

n
: Numbers of rows and columns of `A`. Must be at least zero.

a
: Buffer holding input matrix `A`. Must have size at least (`n`*(`n`+1))/2. See *Matrix Storage* for more details.

x
: Buffer holding the `n`-element right-hand side vector `b`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
: Stride of vector `x`. Must not be zero.

Output Parameters

x
Buffer holding the solution vector **x**.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

tpsv (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event tpsv(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     oneapi::mkl::transpose trans,
                     oneapi::mkl::diag unit_nonunit,
                     std::int64_t n,
                     std::int64_t k,
                     const T *a,
                     T *x,
                     std::int64_t incx,
                     const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event tpsv(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     oneapi::mkl::transpose trans,
                     oneapi::mkl::diag unit_nonunit,
                     std::int64_t n,
                     std::int64_t k,
                     const T *a,
                     T *x,
                     std::int64_t incx,
                     const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans
Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_nonunit
Specifies whether the matrix `A` is unit triangular or not. See *oneMKL defined datatypes* for more details.

n
Numbers of rows and columns of `A`. Must be at least zero.

a
Pointer to input matrix `A`. The array holding input matrix `A` must have size at least (`n`*(`n`+1))/2. See *Matrix Storage* for more details.

x
Pointer to the `n`-element right-hand side vector `b`. The array holding the `n`-element right-hand side vector `b` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

x
Pointer to the solution vector `x`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

trmv

Computes a matrix-vector product using a triangular matrix.

Description

The `trmv` routines compute a matrix-vector product with a triangular matrix. The operation is defined as:

$$x \leftarrow op(A) * x$$

where:

op(`A`) is one of op(`A`) = `A`, or op(`A`) = $\mathtt{A}^T$, or op(`A`) = $\mathtt{A}^H$,

`A` is an `n`-by-`n` unit or non-unit, upper or lower triangular band matrix,

`x` is a vector of length `n`.

`trmv` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

trmv (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void trmv(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              oneapi::mkl::transpose trans,
              oneapi::mkl::diag unit_nonunit,
              std::int64_t n,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void trmv(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              oneapi::mkl::transpose trans,
              oneapi::mkl::diag unit_nonunit,
              std::int64_t n,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
```

(continues on next page)

(continued from previous page)

```
                std::int64_t incx)
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans
Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_nonunit
Specifies whether the matrix `A` is unit triangular or not. See *oneMKL defined datatypes* for more details.

n
Numbers of rows and columns of `A`. Must be at least zero.

a
Buffer holding input matrix `A`. Must have size at least `lda`*`n`. See *Matrix Storage* for more details.

lda
Leading dimension of matrix `A`. Must be at least `n`, and positive.

x
Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

Output Parameters

x
Buffer holding the updated vector `x`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

trmv (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event trmv(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     oneapi::mkl::transpose trans,
                     oneapi::mkl::diag unit_nonunit,
                     std::int64_t n,
                     const T *a,
                     std::int64_t lda,
                     T *x,
                     std::int64_t incx,
                     const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event trmv(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     oneapi::mkl::transpose trans,
                     oneapi::mkl::diag unit_nonunit,
                     std::int64_t n,
                     const T *a,
                     std::int64_t lda,
                     T *x,
                     std::int64_t incx,
                     const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
: The queue where the routine should be executed.

upper_lower
: Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans
: Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_nonunit
: Specifies whether the matrix `A` is unit triangular or not. See *oneMKL defined datatypes* for more details.

n
: Numbers of rows and columns of `A`. Must be at least zero.

a
: Pointer to input matrix `A`. The array holding input matrix `A` must have size at least `lda*n`. See *Matrix Storage* for more details.

lda
: Leading dimension of matrix `A`. Must be at least `n`, and positive.

x
Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

x
Pointer to the updated vector `x`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

trsv

Solves a system of linear equations whose coefficients are in a triangular matrix.

Description

The `trsv` routines compute a matrix-vector product with a triangular band matrix. The operation is defined as:

$$op(A) * x = b$$

where:

op(`A`) is one of op(`A`) = `A`, or op(`A`) = $\mathtt{A}^{\mathrm{T}}$, or op(`A`) = $\mathtt{A}^{\mathrm{H}}$,

`A` is an `n`-by-`n` unit or non-unit, upper or lower triangular matrix,

`b` and `x` are vectors of length `n`.

`trsv` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

trsv (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void trsv(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              oneapi::mkl::transpose trans,
              oneapi::mkl::diag unit_nonunit,
              std::int64_t n,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void trsv(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              oneapi::mkl::transpose trans,
              oneapi::mkl::diag unit_nonunit,
              std::int64_t n,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

upper_lower
: Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans
: Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_nonunit
: Specifies whether the matrix `A` is unit triangular or not. See *oneMKL defined datatypes* for more details.

n
: Numbers of rows and columns of `A`. Must be at least zero.

a
Buffer holding input matrix `A`. Must have size at least `lda`*`n`. See *Matrix Storage* for more details.

lda
Leading dimension of matrix `A`. Must be at least `n`, and positive.

x
Buffer holding the `n`-element right-hand side vector `b`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

Output Parameters

x
Buffer holding the solution vector `x`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

trsv (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event trsv(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     oneapi::mkl::transpose trans,
                     oneapi::mkl::diag unit_nonunit,
                     std::int64_t n,
                     const T *a,
                     std::int64_t lda,
                     T *x,
                     std::int64_t incx,
                     const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event trsv(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     oneapi::mkl::transpose trans,
```

(continues on next page)

(continued from previous page)

```
                        oneapi::mkl::diag unit_nonunit,
                        std::int64_t n,
                        const T *a,
                        std::int64_t lda,
                        T *x,
                        std::int64_t incx,
                        const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans
Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_nonunit
Specifies whether the matrix `A` is unit triangular or not. See *oneMKL defined datatypes* for more details.

n
Numbers of rows and columns of `A`. Must be at least zero.

a
Pointer to input matrix `A`. The array holding input matrix `A` must have size at least `lda*n`. See *Matrix Storage* for more details.

lda
Leading dimension of matrix `A`. Must be at least `n`, and positive.

x
Pointer to the `n`-element right-hand side vector `b`. The array holding the `n`-element right-hand side vector `b` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride of vector `x`. Must not be zero.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

x
Pointer to the solution vector `x`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

Parent topic: *BLAS Routines*

BLAS Level 3 Routines

BLAS Level 3 includes routines which perform matrix-matrix operations as described in the following table.

Routines	Description
gemm	Computes a matrix-matrix product with general matrices.
hemm	Computes a matrix-matrix product where one input matrix is Hermitian and one is general.
herk	Performs a Hermitian rank-k update.
her2k	Performs a Hermitian rank-2k update.
symm	Computes a matrix-matrix product where one input matrix is symmetric and one matrix is general.
syrk	Performs a symmetric rank-k update.
syr2k	Performs a symmetric rank-2k update.
trmm	Computes a matrix-matrix product where one input matrix is triangular and one input matrix is general.
trsm	Solves a triangular matrix equation (forward or backward solve).

gemm

Computes a matrix-matrix product with general matrices.

Description

The gemm routines compute a scalar-matrix-matrix product and add the result to a scalar-matrix product, with general matrices. The operation is defined as:

$$C \leftarrow alpha * op(A) * op(B) + beta * C$$

where:

op(X) is one of op(X) = X, or op(X) = X^T, or op(X) = X^H,

alpha and beta are scalars,

A, B and C are matrices,

op(A) is an m-by-k matrix,

op(B) is a k-by-n matrix,

C is an m-by-n matrix.

gemm supports the following precisions.

Ta (A matrix)	Tb (B matrix)	Tc (C matrix)	Ts (alpha/beta)
std::int8_t	std::int8_t	std::int32_t	float
std::int8_t	std::int8_t	float	float
half	half	float	float
half	half	half	half
bfloat16	bfloat16	float	float
bfloat16	bfloat16	bfloat16	float
float	float	float	float
double	double	double	double
std::complex<float>	std::complex<float>	std::complex<float>	std::complex<float>
std::complex<double	std::complex<double	std::complex<double	std::complex<double>

gemm (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void gemm(sycl::queue &queue,
              oneapi::mkl::transpose transa,
              oneapi::mkl::transpose transb,
              std::int64_t m,
              std::int64_t n,
              std::int64_t k,
              Ts alpha,
              sycl::buffer<Ta,1> &a,
              std::int64_t lda,
              sycl::buffer<Tb,1> &b,
```

(continues on next page)

(continued from previous page)

```
            std::int64_t ldb,
            Ts beta,
            sycl::buffer<Tc,1> &c,
            std::int64_t ldc)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void gemm(sycl::queue &queue,
            oneapi::mkl::transpose transa,
            oneapi::mkl::transpose transb,
            std::int64_t m,
            std::int64_t n,
            std::int64_t k,
            Ts alpha,
            sycl::buffer<Ta,1> &a,
            std::int64_t lda,
            sycl::buffer<Tb,1> &b,
            std::int64_t ldb,
            Ts beta,
            sycl::buffer<Tc,1> &c,
            std::int64_t ldc)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

transa
: Specifies the form of op(`A`), the transposition operation applied to `A`.

transb
: Specifies the form of op(`B`), the transposition operation applied to `B`.

m
: Specifies the number of rows of the matrix op(`A`) and of the matrix `C`. The value of m must be at least zero.

n
: Specifies the number of columns of the matrix op(`B`) and the number of columns of the matrix `C`. The value of n must be at least zero.

k
: Specifies the number of columns of the matrix op(`A`) and the number of rows of the matrix op(`B`). The value of k must be at least zero.

alpha
: Scaling factor for the matrix-matrix product.

a
: The buffer holding the input matrix `A`.

	A not transposed	A transposed
Column major	A is an m-by-k matrix so the array a must have size at least lda*k.	A is an k-by-m matrix so the array a must have size at least lda*m
Row major	A is an m-by-k matrix so the array a must have size at least lda*m.	A is an k-by-m matrix so the array a must have size at least lda*k

See *Matrix Storage* for more details.

lda

The leading dimension of A. It must be positive.

	A not transposed	A transposed
Column major	lda must be at least m.	lda must be at least k.
Row major	lda must be at least k.	lda must be at least m.

b

The buffer holding the input matrix B.

	B not transposed	B transposed
Column major	B is an k-by-n matrix so the array b must have size at least ldb*n.	B is an n-by-k matrix so the array b must have size at least ldb*k
Row major	B is an k-by-n matrix so the array b must have size at least ldb*k.	B is an n-by-k matrix so the array b must have size at least ldb*n

See *Matrix Storage* for more details.

ldb

The leading dimension of B. It must be positive.

	B not transposed	B transposed
Column major	ldb must be at least k.	ldb must be at least n.
Row major	ldb must be at least n.	ldb must be at least k.

beta

Scaling factor for matrix C.

c

The buffer holding the input/output matrix C. It must have a size of at least ldc*n if column major layout is used to store matrices or at least ldc*m if row major layout is used to store matrices . See *Matrix Storage* for more details.

ldc

The leading dimension of C. It must be positive and at least m if column major layout is used to store matrices or at least n if row major layout is used to store matrices.

Output Parameters

c
The buffer, which is overwritten by `alpha`*op(`A`)*op(`B`) + `beta`*`C`.

Notes

If `beta` = 0, matrix `C` does not need to be initialized before calling `gemm`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

gemm (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event gemm(sycl::queue &queue,
                     oneapi::mkl::transpose transa,
                     oneapi::mkl::transpose transb,
                     std::int64_t m,
                     std::int64_t n,
                     std::int64_t k,
                     value_or_pointer<Ts> alpha,
                     const Ta *a,
                     std::int64_t lda,
                     const Tb *b,
                     std::int64_t ldb,
                     value_or_poitner<Ts> beta,
                     Tc *c,
                     std::int64_t ldc,
                     const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event gemm(sycl::queue &queue,
                     oneapi::mkl::transpose transa,
                     oneapi::mkl::transpose transb,
                     std::int64_t m,
                     std::int64_t n,
                     std::int64_t k,
```

(continues on next page)

(continued from previous page)

```
                        value_or_pointer<Ts> alpha,
                        const Ta *a,
                        std::int64_t lda,
                        const Tb *b,
                        std::int64_t ldb,
                        value_or_pointer<Ts> beta,
                        Tc *c,
                        std::int64_t ldc,
                        const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

transa
Specifies the form of op(`A`), the transposition operation applied to `A`.

transb
Specifies the form of op(`B`), the transposition operation applied to `B`.

m
Specifies the number of rows of the matrix op(`A`) and of the matrix `C`. The value of m must be at least zero.

n
Specifies the number of columns of the matrix op(`B`) and the number of columns of the matrix `C`. The value of n must be at least zero.

k
Specifies the number of columns of the matrix op(`A`) and the number of rows of the matrix op(`B`). The value of k must be at least zero.

alpha
Scaling factor for the matrix-matrix product. See *Scalar Arguments in BLAS* for more details.

a
Pointer to input matrix `A`.

	`A` not transposed	`A` transposed
Column major	`A` is an `m`-by-`k` matrix so the array `a` must have size at least `lda*k`.	`A` is an `k`-by-`m` matrix so the array `a` must have size at least `lda*m`
Row major	`A` is an `m`-by-`k` matrix so the array `a` must have size at least `lda*m`.	`A` is an `k`-by-`m` matrix so the array `a` must have size at least `lda*k`

See *Matrix Storage* for more details.

lda
The leading dimension of `A`. It must be positive.

	`A` not transposed	`A` transposed
Column major	`lda` must be at least `m`.	`lda` must be at least `k`.
Row major	`lda` must be at least `k`.	`lda` must be at least `m`.

b

Pointer to input matrix `B`.

	`B` not transposed	`B` transposed
Column major	`B` is an `k`-by-`n` matrix so the array `b` must have size at least `ldb*n`.	`B` is an `n`-by-`k` matrix so the array `b` must have size at least `ldb*k`
Row major	`B` is an `k`-by-`n` matrix so the array `b` must have size at least `ldb*k`.	`B` is an `n`-by-`k` matrix so the array `b` must have size at least `ldb*n`

See *Matrix Storage* for more details.

ldb

The leading dimension of `B`. It must be positive.

	`B` not transposed	`B` transposed
Column major	`ldb` must be at least `k`.	`ldb` must be at least `n`.
Row major	`ldb` must be at least `n`.	`ldb` must be at least `k`.

beta

Scaling factor for matrix `C`. See *Scalar Arguments in BLAS* for more details.

c

The pointer to input/output matrix `C`. It must have a size of at least `ldc*n` if column major layout is used to store matrices or at least `ldc*m` if row major layout is used to store matrices . See *Matrix Storage* for more details.

ldc

The leading dimension of `C`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if row major layout is used to store matrices.

dependencies

List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

c

Pointer to the output matrix, overwritten by `alpha`*op(`A`)*op(`B`) + `beta`*`C`.

Notes

If `beta` = 0, matrix `C` does not need to be initialized before calling `gemm`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 3 Routines*

hemm

Computes a matrix-matrix product where one input matrix is Hermitian and one is general.

Description

The `hemm` routines compute a scalar-matrix-matrix product and add the result to a scalar-matrix product, where one of the matrices in the multiplication is Hermitian. The argument `left_right` determines if the Hermitian matrix, `A`, is on the left of the multiplication (`left_right = side::left`) or on the right (`left_right = side::right`). Depending on `left_right`, the operation is defined as:

$$C \leftarrow alpha * A * B + beta * C$$

or

$$C \leftarrow alpha * B * A + beta * C$$

where:

`alpha` and `beta` are scalars,

`A` is a Hermitian matrix, either `m`-by-`m` or `n`-by-`n` matrices,

`B` and `C` are `m`-by-`n` matrices.

`hemm` supports the following precisions:

T
`std::complex<float>`
`std::complex<double>`

hemm (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void hemm(sycl::queue &queue,
              oneapi::mkl::side left_right,
              oneapi::mkl::uplo upper_lower,
              std::int64_t m,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &b,
              std::int64_t ldb,
              T beta,
              sycl::buffer<T,1> &c,
              std::int64_t ldc)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void hemm(sycl::queue &queue,
              oneapi::mkl::side left_right,
              oneapi::mkl::uplo upper_lower,
              std::int64_t m,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &b,
              std::int64_t ldb,
              T beta,
              sycl::buffer<T,1> &c,
              std::int64_t ldc)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

left_right
: Specifies whether `A` is on the left side of the multiplication (`side::left`) or on the right side (`side::right`). See *oneMKL defined datatypes* for more details.

uplo
: Specifies whether `A`'s data is stored in its upper or lower triangle. See *oneMKL defined datatypes* for more details.

m
: Specifies the number of rows of the matrix `B` and `C`.

  The value of `m` must be at least zero.

n
Specifies the number of columns of the matrix `B` and `C`.

The value of `n` must be at least zero.

alpha
Scaling factor for the matrix-matrix product.

a
Buffer holding input matrix `A`. Must have size at least `lda*m` if `A` is on the left of the multiplication, or `lda*n` if `A` is on the right. See *Matrix Storage* for more details.

lda
Leading dimension of `A`. Must be at least `m` if `A` is on the left of the multiplication, or at least `n` if `A` is on the right. Must be positive.

b
Buffer holding input matrix `B`. Must have size at least `ldb*n` if column major layout is used to store matrices or at least `ldb*m` if row major layout is used to store matrices. See *Matrix Storage* for more details.

ldb
Leading dimension of `B`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if column major layout is used to store matrices.

beta
Scaling factor for matrix `C`.

c
The buffer holding the input/output matrix `C`. It must have a size of at least `ldc*n` if column major layout is used to store matrices or at least `ldc*m` if row major layout is used to store matrices . See *Matrix Storage* for more details.

ldc
The leading dimension of `C`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if column major layout is used to store matrices.

Output Parameters

c
Output buffer, overwritten by `alpha*A*B + beta*C` (`left_right = side::left`) or `alpha*B*A + beta*C` (`left_right = side::right`).

Notes

If `beta = 0`, matrix `C` does not need to be initialized before calling `hemm`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

hemm (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event hemm(sycl::queue &queue,
                     oneapi::mkl::side left_right,
                     oneapi::mkl::uplo upper_lower,
                     std::int64_t m,
                     std::int64_t n,
                     value_or_pointer<T> alpha,
                     const T *a,
                     std::int64_t lda,
                     const T *b,
                     std::int64_t ldb,
                     value_or_pointer<T> beta,
                     T *c,
                     std::int64_t ldc,
                     const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event hemm(sycl::queue &queue,
                     oneapi::mkl::side left_right,
                     oneapi::mkl::uplo upper_lower,
                     std::int64_t m,
                     std::int64_t n,
                     value_or_pointer<T> alpha,
                     const T *a,
                     std::int64_t lda,
                     const T *b,
                     std::int64_t ldb,
                     value_or_pointer<T> beta,
                     T *c,
                     std::int64_t ldc,
                     const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
: The queue where the routine should be executed.

left_right
: Specifies whether `A` is on the left side of the multiplication (`side::left`) or on the right side (`side::right`). See *oneMKL defined datatypes* for more details.

uplo
: Specifies whether `A`'s data is stored in its upper or lower triangle. See *oneMKL defined datatypes* for more details.

m
Specifies the number of rows of the matrix `B` and `C`.

The value of `m` must be at least zero.

n
Specifies the number of columns of the matrix `B` and `C`.

The value of `n` must be at least zero.

alpha
Scaling factor for the matrix-matrix product. See *Scalar Arguments in BLAS* for more details.

a
Pointer to input matrix `A`. Must have size at least `lda*m` if `A` is on the left of the multiplication, or `lda*n` if `A` is on the right. See *Matrix Storage* for more details.

lda
Leading dimension of `A`. Must be at least `m` if `A` is on the left of the multiplication, or at least `n` if `A` is on the right. Must be positive.

b
Pointer to input matrix `B`. Must have size at least `ldb*n` if column major layout is used to store matrices or at least `ldb*m` if row major layout is used to store matrices. See *Matrix Storage* for more details.

ldb
Leading dimension of `B`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if column major layout is used to store matrices.

beta
Scaling factor for matrix `C`. See *Scalar Arguments in BLAS* for more details.

c
The pointer to input/output matrix `C`. It must have a size of at least `ldc*n` if column major layout is used to store matrices or at least `ldc*m` if row major layout is used to store matrices . See *Matrix Storage* for more details.

ldc
The leading dimension of `C`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if column major layout is used to store matrices.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

c
Pointer to the output matrix, overwritten by `alpha*A*B + beta*C` (`left_right = side::left`) or `alpha*B*A + beta*C` (`left_right = side::right`).

Notes

If `beta` = 0, matrix `C` does not need to be initialized before calling `hemm`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 3 Routines*

herk

Performs a Hermitian rank-k update.

Description

The `herk` routines compute a rank-k update of a Hermitian matrix `C` by a general matrix `A`. The operation is defined as:

$$C \leftarrow alpha * op(A) * op(A)^H + beta * C$$

where:

op(`X`) is one of op(`X`) = `X` or op(`X`) = $\texttt{X}^{\texttt{H}}$,

`alpha` and `beta` are real scalars,

`C` is a Hermitian matrix and `A` is a general matrix.

Here op(`A`) is `n` x `k`, and `C` is `n` x `n`.

`herk` supports the following precisions:

T	Treal
`std::complex<float>`	`float`
`std::complex<double>`	`double`

herk (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void herk(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              oneapi::mkl::transpose trans,
              std::int64_t n,
              std::int64_t k,
              Treal alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              Treal beta,
              sycl::buffer<T,1> &c,
              std::int64_t ldc)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void herk(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              oneapi::mkl::transpose trans,
              std::int64_t n,
              std::int64_t k,
              Treal alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              Treal beta,
              sycl::buffer<T,1> &c,
              std::int64_t ldc)
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A`'s data is stored in its upper or lower triangle. See *oneMKL defined datatypes* for more details.

trans
Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details. Supported operations are `transpose::nontrans` and `transpose::conjtrans`.

n
The number of rows and columns in `C`.The value of `n` must be at least zero.

k
Number of columns in op(`A`).

The value of `k` must be at least zero.

alpha
Real scaling factor for the rank-k update.

a

Buffer holding input matrix `A`.

	`trans = transpose::nontrans`	`trans = transpose::trans` or `transpose::conjtrans`
Column major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*k`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*n`
Row major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*n`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*k`.

See *Matrix Storage* for more details.

lda

The leading dimension of `A`. It must be positive.

	`trans = transpose::nontrans`	`trans = transpose::trans` or `transpose::conjtrans`
Column major	`lda` must be at least `n`.	`lda` must be at least `k`.
Row major	`lda` must be at least `k`.	`lda` must be at least `n`.

beta

Real scaling factor for matrix `C`.

c

Buffer holding input/output matrix `C`. Must have size at least `ldc*n`. See *Matrix Storage* for more details.

ldc

Leading dimension of `C`. Must be positive and at least `n`.

Output Parameters

c

The output buffer, overwritten by `alpha`*op(`A`)*op(`A`)^T + `beta`*`C`. The imaginary parts of the diagonal elements are set to zero.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

herk (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event herk(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     oneapi::mkl::transpose trans,
                     std::int64_t n,
                     std::int64_t k,
                     value_or_pointer<Treal> alpha,
                     const T *a,
                     std::int64_t lda,
                     value_or_pointer<Treal> beta,
                     T *c,
                     std::int64_t ldc,
                     const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event herk(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     oneapi::mkl::transpose trans,
                     std::int64_t n,
                     std::int64_t k,
                     value_or_pointer<Treal> alpha,
                     const T *a,
                     std::int64_t lda,
                     value_or_pointer<Treal> beta,
                     T *c,
                     std::int64_t ldc,
                     const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A`'s data is stored in its upper or lower triangle. See *oneMKL defined datatypes* for more details.

trans
Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details. Supported operations are `transpose::nontrans` and `transpose::conjtrans`.

n
The number of rows and columns in `C`.The value of `n` must be at least zero.

k
Number of columns in op(`A`).

The value of `k` must be at least zero.

alpha
Real scaling factor for the rank-k update. See *Scalar Arguments in BLAS* for more details.

a
Pointer to input matrix `A`.

	`trans = transpose::nontrans`	`trans = transpose::trans` or `transpose::conjtrans`
Column major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*k`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*n`
Row major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*n`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*k`.

See *Matrix Storage* for more details.

lda
The leading dimension of `A`. It must be positive.

	`trans = transpose::nontrans`	`trans = transpose::trans` or `transpose::conjtrans`
Column major	`lda` must be at least `n`.	`lda` must be at least `k`.
Row major	`lda` must be at least `k`.	`lda` must be at least `n`.

beta
Real scaling factor for matrix `C`. See *Scalar Arguments in BLAS* for more details.

c
Pointer to input/output matrix `C`. Must have size at least `ldc*n`. See *Matrix Storage* for more details.

ldc
Leading dimension of `C`. Must be positive and at least `n`.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

c
Pointer to the output matrix, overwritten by `alpha`*op(`A`)*op(`A`)^T + `beta`*`C`. The imaginary parts of the diagonal elements are set to zero.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 3 Routines*

her2k

Performs a Hermitian rank-2k update.

Description

The `her2k` routines perform a rank-2k update of an `n` x `n` Hermitian matrix `C` by general matrices `A` and `B`.

If `trans` = `transpose::nontrans`, the operation is defined as:

$$C \leftarrow alpha * A * B^H + conjg(alpha) * B * A^H + beta * C$$

where `A` is `n` x `k` and `B` is `k` x `n`.

If `trans` = `transpose::conjtrans`, the operation is defined as:

$$C \leftarrow alpha * B * A^H + conjg(alpha) * A * B^H + beta * C$$

where `A` is `k` x `n` and `B` is `n` x `k`.

In both cases:

`alpha` is a complex scalar and `beta` is a real scalar.

`C` is a Hermitian matrix and `A` , `B` are general matrices.

The inner dimension of both matrix multiplications is `k`.

`her2k` supports the following precisions:

T	Treal
`std::complex<float>`	`float`
`std::complex<double>`	`double`

her2k (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void her2k(sycl::queue &queue,
               oneapi::mkl::uplo upper_lower,
               oneapi::mkl::transpose trans,
               std::int64_t n,
               std::int64_t k,
               T alpha,
               sycl::buffer<T,1> &a,
               std::int64_t lda,
               sycl::buffer<T,1> &b,
               std::int64_t ldb,
               Treal beta,
               sycl::buffer<T,1> &c,
               std::int64_t ldc)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void her2k(sycl::queue &queue,
               oneapi::mkl::uplo upper_lower,
               oneapi::mkl::transpose trans,
               std::int64_t n,
               std::int64_t k,
               T alpha,
               sycl::buffer<T,1> &a,
               std::int64_t lda,
               sycl::buffer<T,1> &b,
               std::int64_t ldb,
               Treal beta,
               sycl::buffer<T,1> &c,
               std::int64_t ldc)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

upper_lower
: Specifies whether `A`'s data is stored in its upper or lower triangle. See *oneMKL defined datatypes* for more details.

trans
: Specifies the operation to apply, as described above. Supported operations are `transpose::nontrans` and `transpose::conjtrans`.

n
: The number of rows and columns in `C`. The value of `n` must be at least zero.

k
: The inner dimension of matrix multiplications. The value of `k` must be at least equal to zero.

alpha
Complex scaling factor for the rank-2k update.

a
Buffer holding input matrix `A`.

	`trans = transpose::nontrans`	`trans = transpose::trans` or `transpose::conjtrans`
Column major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*k`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*n`
Row major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*n`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*k`.

See *Matrix Storage* for more details.

lda
The leading dimension of `A`. It must be positive.

	`trans = transpose::nontrans`	`trans = transpose::trans` or `transpose::conjtrans`
Column major	`lda` must be at least `n`.	`lda` must be at least `k`.
Row major	`lda` must be at least `k`.	`lda` must be at least `n`.

b
Buffer holding input matrix `B`.

	`trans = transpose::nontrans`	`trans = transpose::trans` or `transpose::conjtrans`
Column major	`B` is an `k`-by-`n` matrix so the array `b` must have size at least `ldb*n`.	`B` is an `n`-by-`k` matrix so the array `b` must have size at least `ldb*k`
Row major	`B` is an `k`-by-`n` matrix so the array `b` must have size at least `ldb*k`.	`B` is an `n`-by-`k` matrix so the array `b` must have size at least `ldb*n`.

See *Matrix Storage* for more details.

ldb
The leading dimension of `B`. It must be positive.

	`trans = transpose::nontrans`	`trans = transpose::trans` or `transpose::conjtrans`
Column major	`ldb` must be at least `k`.	`ldb` must be at least `n`.
Row major	`ldb` must be at least `n`.	`ldb` must be at least `k`.

beta
Real scaling factor for matrix `C`.

c
Buffer holding input/output matrix `C`. Must have size at least `ldc*n`. See *Matrix Storage* for more details.

ldc
Leading dimension of C. Must be positive and at least n.

Output Parameters

c
Output buffer, overwritten by the updated C matrix.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

her2k (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event her2k(sycl::queue &queue,
                      oneapi::mkl::uplo upper_lower,
                      oneapi::mkl::transpose trans,
                      std::int64_t n,
                      std::int64_t k,
                      value_or_pointer<T> alpha,
                      const T *a,
                      std::int64_t lda,
                      const T *b,
                      std::int64_t ldb,
                      value_or_pointer<Treal> beta,
                      T *c,
                      std::int64_t ldc,
                      const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event her2k(sycl::queue &queue,
                      oneapi::mkl::uplo upper_lower,
                      oneapi::mkl::transpose trans,
                      std::int64_t n,
                      std::int64_t k,
                      value_or_pointer<T> alpha,
                      const T *a,
                      std::int64_t lda,
```

(continues on next page)

(continued from previous page)

```
                        const T *b,
                        std::int64_t ldb,
                        value_or_pointer<Treal> beta,
                        T *c,
                        std::int64_t ldc,
                        const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `A`'s data is stored in its upper or lower triangle. See *oneMKL defined datatypes* for more details.

trans
Specifies the operation to apply, as described above. Supported operations are `transpose::nontrans` and `transpose::conjtrans`.

n
The number of rows and columns in `C`. The value of `n` must be at least zero.

k
The inner dimension of matrix multiplications. The value of `k` must be at least equal to zero.

alpha
Complex scaling factor for the rank-2k update. See *Scalar Arguments in BLAS* for more details.

a
Pointer to input matrix `A`.

	`trans = transpose::nontrans`	`trans = transpose::trans` or `transpose::conjtrans`
Column major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*k`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*n`
Row major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*n`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*k`.

See *Matrix Storage* for more details.

lda
The leading dimension of `A`. It must be positive.

	`trans = transpose::nontrans`	`trans = transpose::trans` or `transpose::conjtrans`
Column major	`lda` must be at least `n`.	`lda` must be at least `k`.
Row major	`lda` must be at least `k`.	`lda` must be at least `n`.

b
Pointer to input matrix `B`.

	trans = transpose::nontrans	trans = transpose::trans or transpose::conjtrans
Column major	B is an k-by-n matrix so the array b must have size at least ldb*n.	B is an n-by-k matrix so the array b must have size at least ldb*k
Row major	B is an k-by-n matrix so the array b must have size at least ldb*k.	B is an n-by-k matrix so the array b must have size at least ldb*n.

See *Matrix Storage* for more details.

ldb

The leading dimension of B. It must be positive.

	trans = transpose::nontrans	trans = transpose::trans or transpose::conjtrans
Column major	ldb must be at least k.	ldb must be at least n.
Row major	ldb must be at least n.	ldb must be at least k.

beta

Real scaling factor for matrix C. See *Scalar Arguments in BLAS* for more details.

c

Pointer to input/output matrix C. Must have size at least ldc*n. See *Matrix Storage* for more details.

ldc

Leading dimension of C. Must be positive and at least n.

dependencies

List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

c

Pointer to the output matrix, overwritten by the updated C matrix.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 3 Routines*

symm

Computes a matrix-matrix product where one input matrix is symmetric and one matrix is general.

Description

The `symm` routines compute a scalar-matrix-matrix product and add the result to a scalar-matrix product, where one of the matrices in the multiplication is symmetric. The argument `left_right` determines if the symmetric matrix, `A`, is on the left of the multiplication (`left_right = side::left`) or on the right (`left_right = side::right`). Depending on `left_right`, the operation is defined as:

$$C \leftarrow alpha * A * B + beta * C$$

or

$$C \leftarrow alpha * B * A + beta * C$$

where:

`alpha` and `beta` are scalars,

`A` is a symmetric matrix, either `m`-by-`m` or `n`-by-`n`,

`B` and `C` are `m`-by-`n` matrices.

`symm` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

symm (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void symm(sycl::queue &queue,
              oneapi::mkl::side left_right,
              oneapi::mkl::uplo upper_lower,
              std::int64_t m,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &b,
              std::int64_t ldb,
              T beta,
```

(continues on next page)

(continued from previous page)

```
                sycl::buffer<T,1> &c,
                std::int64_t ldc)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void symm(sycl::queue &queue,
                oneapi::mkl::side left_right,
                oneapi::mkl::uplo upper_lower,
                std::int64_t m,
                std::int64_t n,
                T alpha,
                sycl::buffer<T,1> &a,
                std::int64_t lda,
                sycl::buffer<T,1> &b,
                std::int64_t ldb,
                T beta,
                sycl::buffer<T,1> &c,
                std::int64_t ldc)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

left_right
: Specifies whether `A` is on the left side of the multiplication (`side::left`) or on the right side (`side::right`). See *oneMKL defined datatypes* for more details.

upper_lower
: Specifies whether `A`'s data is stored in its upper or lower triangle. See *oneMKL defined datatypes* for more details.

m
: Number of rows of `B` and `C`. The value of `m` must be at least zero.

n
: Number of columns of `B` and `C`. The value of `n` must be at least zero.

alpha
: Scaling factor for the matrix-matrix product.

a
: Buffer holding input matrix `A`. Must have size at least `lda*m` if `A` is on the left of the multiplication, or `lda*n` if `A` is on the right. See *Matrix Storage* for more details.

lda
: Leading dimension of `A`. Must be at least `m` if `A` is on the left of the multiplication, or at least `n` if `A` is on the right. Must be positive.

b
: Buffer holding input matrix `B`. Must have size at least `ldb*n` if column major layout is used to store matrices or at least `ldb*m` if row major layout is used to store matrices. See *Matrix Storage* for more details.

ldb
: Leading dimension of `B`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if column major layout is used to store matrices.

beta
Scaling factor for matrix `C`.

c
The buffer holding the input/output matrix `C`. It must have a size of at least `ldc*n` if column major layout is used to store matrices or at least `ldc*m` if row major layout is used to store matrices. See *Matrix Storage* for more details.

ldc
The leading dimension of `C`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if column major layout is used to store matrices.

Output Parameters

c
Output buffer, overwritten by `alpha*A*B + beta*C` (`left_right = side::left`) or `alpha*B*A + beta*C` (`left_right = side::right`).

Notes

If `beta` = 0, matrix `C` does not need to be initialized before calling `symm`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

symm (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event symm(sycl::queue &queue,
                     oneapi::mkl::side left_right,
                     oneapi::mkl::uplo upper_lower,
                     std::int64_t m,
                     std::int64_t n,
                     value_or_pointer<T> alpha,
                     const T *a,
                     std::int64_t lda,
                     const T *b,
                     std::int64_t ldb,
                     value_or_pointer<T> beta,
```

(continues on next page)

(continued from previous page)

```
                    T *c,
                    std::int64_t ldc,
                    const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event symm(sycl::queue &queue,
                    oneapi::mkl::side left_right,
                    oneapi::mkl::uplo upper_lower,
                    std::int64_t m,
                    std::int64_t n,
                    value_or_pointer<T> alpha,
                    const T *a,
                    std::int64_t lda,
                    const T *b,
                    std::int64_t ldb,
                    value_or_pointer<T> beta,
                    T *c,
                    std::int64_t ldc,
                    const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

left_right
Specifies whether `A` is on the left side of the multiplication (`side::left`) or on the right side (`side::right`). See *oneMKL defined datatypes* for more details.

upper_lower
Specifies whether `A`'s data is stored in its upper or lower triangle. See *oneMKL defined datatypes* for more details.

m
Number of rows of `B` and `C`. The value of `m` must be at least zero.

n
Number of columns of `B` and `C`. The value of `n` must be at least zero.

alpha
Scaling factor for the matrix-matrix product. See *Scalar Arguments in BLAS* for more details.

a
Pointer to input matrix `A`. Must have size at least `lda*m` if `A` is on the left of the multiplication, or `lda*n` if `A` is on the right. See *Matrix Storage* for more details.

lda
Leading dimension of `A`. Must be at least `m` if `A` is on the left of the multiplication, or at least `n` if `A` is on the right. Must be positive.

b
Pointer to input matrix `B`. Must have size at least `ldb*n` if column major layout is used to store matrices or at least `ldb*m` if row major layout is used to store matrices. See *Matrix Storage* for more details.

ldb
Leading dimension of `B`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if column major layout is used to store matrices.

beta
Scaling factor for matrix `C`. See *Scalar Arguments in BLAS* for more details.

c
The pointer to input/output matrix `C`. It must have a size of at least `ldc*n` if column major layout is used to store matrices or at least `ldc*m` if row major layout is used to store matrices . See *Matrix Storage* for more details.

ldc
The leading dimension of `C`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if column major layout is used to store matrices.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

c
Pointer to the output matrix, overwritten by `alpha*A*B + beta*C` (`left_right = side::left`) or `alpha*B*A + beta*C` (`left_right = side::right`).

Notes

If `beta = 0`, matrix `C` does not need to be initialized before calling `symm`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 3 Routines*

syrk

Performs a symmetric rank-k update.

Description

The `syrk` routines perform a rank-k update of a symmetric matrix `C` by a general matrix `A`. The operation is defined as:

$$C \leftarrow alpha * op(A) * op(A)^T + beta * C$$

where:

op(`X`) is one of op(`X`) = `X` or op(`X`) = $\mathtt{X}^{\mathrm{T}}$,

`alpha` and `beta` are scalars,

`C` is a symmetric matrix and `A`is a general matrix.

Here op(`A`) is `n`-by-`k`, and `C` is `n`-by-`n`.

`syrk` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

syrk (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void syrk(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              oneapi::mkl::transpose trans,
              std::int64_t n,
              std::int64_t k,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              T beta,
              sycl::buffer<T,1> &c,
              std::int64_t ldc)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void syrk(sycl::queue &queue,
              oneapi::mkl::uplo upper_lower,
              oneapi::mkl::transpose trans,
              std::int64_t n,
              std::int64_t k,
```

(continues on next page)

(continued from previous page)

```
                T alpha,
                sycl::buffer<T,1> &a,
                std::int64_t lda,
                T beta,
                sycl::buffer<T,1> &c,
                std::int64_t ldc)
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `C`'s data is stored in its upper or lower triangle. See *oneMKL defined datatypes* for more details.

trans
Specifies op(`A`), the transposition operation applied to `A` (See *oneMKL defined datatypes* for more details). Conjugation is never performed, even if `trans` = `transpose::conjtrans`.

n
Number of rows and columns in `C`. The value of `n` must be at least zero.

k
Number of columns in op(`A`).The value of `k` must be at least zero.

alpha
Scaling factor for the rank-k update.

a
Buffer holding input matrix `A`.

	`trans` = `transpose::nontrans`	`trans` = `transpose::trans` or `transpose::conjtrans`
Column major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda`*`k`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda`*`n`
Row major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda`*`n`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda`*`k`.

See *Matrix Storage* for more details.

lda
The leading dimension of `A`. It must be positive.

	`trans` = `transpose::nontrans`	`trans` = `transpose::trans` or `transpose::conjtrans`
Column major	`lda` must be at least `n`.	`lda` must be at least `k`.
Row major	`lda` must be at least `k`.	`lda` must be at least `n`.

beta
Scaling factor for matrix `C`.

c
Buffer holding input/output matrix `C`. Must have size at least `ldc`*`n`. See *Matrix Storage* for more details.

ldc
Leading dimension of `C`. Must be positive and at least `n`.

Output Parameters

c
Output buffer, overwritten by `alpha`*op(`A`)*op(`A`)^T + `beta`*`C`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

syrk (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event syrk(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     oneapi::mkl::transpose trans,
                     std::int64_t n,
                     std::int64_t k,
                     value_or_pointer<T> alpha,
                     const T *a,
                     std::int64_t lda,
                     value_or_pointer<T> beta,
                     T *c,
                     std::int64_t ldc,
                     const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event syrk(sycl::queue &queue,
                     oneapi::mkl::uplo upper_lower,
                     oneapi::mkl::transpose trans,
                     std::int64_t n,
                     std::int64_t k,
                     value_or_pointer<T> alpha,
                     const T *a,
```

(continues on next page)

(continued from previous page)

```
                        std::int64_t lda,
                        value_or_pointer<T> beta,
                        T *c,
                        std::int64_t ldc,
                        const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `C`'s data is stored in its upper or lower triangle. See *oneMKL defined datatypes* for more details.

trans
Specifies op(`A`), the transposition operation applied to `A` (See *oneMKL defined datatypes* for more details). Conjugation is never performed, even if `trans` = `transpose::conjtrans`.

n
Number of rows and columns in `C`. The value of `n` must be at least zero.

k
Number of columns in op(`A`). The value of `k` must be at least zero.

alpha
Scaling factor for the rank-k update. See *Scalar Arguments in BLAS* for more details.

a
Pointer to input matrix `A`.

	`trans` = `transpose::nontrans`	`trans` = `transpose::trans` or `transpose::conjtrans`
Column major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*k`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*n`
Row major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*n`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*k`.

See *Matrix Storage* for more details.

lda
The leading dimension of `A`. It must be positive.

	`trans` = `transpose::nontrans`	`trans` = `transpose::trans` or `transpose::conjtrans`
Column major	`lda` must be at least `n`.	`lda` must be at least `k`.
Row major	`lda` must be at least `k`.	`lda` must be at least `n`.

beta
Scaling factor for matrix `C`. See *Scalar Arguments in BLAS* for more details.

c
Pointer to input/output matrix `C`. Must have size at least `ldc*n`. See *Matrix Storage* for more details.

ldc
Leading dimension of `C`. Must be positive and at least `n`.

Output Parameters

c
Pointer to the output matrix, overwritten by `alpha`*op(`A`)*op(`A`)^T + `beta`*`C`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 3 Routines*

syr2k

Performs a symmetric rank-2k update.

Description

The `syr2k` routines perform a rank-2k update of an `n` x `n` symmetric matrix `C` by general matrices `A` and `B`.

If `trans` = `transpose::nontrans`, the operation is defined as:

$$C \leftarrow alpha * (A * B^T + B * A^T) + beta * C$$

where `A` and `B` are `n` x `k` matrices.

If `trans` = `transpose::trans`, the operation is defined as:

$$C \leftarrow alpha * (A^T * B + B^T * A) + beta * C$$

where `A` and `B` are `k` x `n` matrices.

In both cases:

`alpha` and `beta` are scalars,

`C` is a symmetric matrix and `A`,`B` are general matrices,

The inner dimension of both matrix multiplications is `k`.

`syr2k` supports the following precisions:

T
float
double
std::complex<float>
std::complex<double>

syr2k (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void syr2k(sycl::queue &queue,
               oneapi::mkl::uplo upper_lower,
               oneapi::mkl::transpose trans,
               std::int64_t n,
               std::int64_t k,
               T alpha,
               sycl::buffer<T,1> &a,
               std::int64_t lda,
               sycl::buffer<T,1> &b,
               std::int64_t ldb,
               T beta,
               sycl::buffer<T,1> &c,
               std::int64_t ldc)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void syr2k(sycl::queue &queue,
               oneapi::mkl::uplo upper_lower,
               oneapi::mkl::transpose trans,
               std::int64_t n,
               std::int64_t k,
               T alpha,
               sycl::buffer<T,1> &a,
               std::int64_t lda,
               sycl::buffer<T,1> &b,
               std::int64_t ldb,
               T beta,
               sycl::buffer<T,1> &c,
               std::int64_t ldc)
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether C's data is stored in its upper or lower triangle. See *oneMKL defined datatypes* for more details.

trans
Specifies the operation to apply, as described above. Conjugation is never performed, even if `trans = transpose::conjtrans`.

n
Number of rows and columns in C.The value of `n` must be at least zero.

k
Inner dimension of matrix multiplications.The value of `k` must be at least zero.

alpha
Scaling factor for the rank-2k update.

a
Buffer holding input matrix `A`.

	`trans = transpose::nontrans`	`trans = transpose::trans` or `transpose::conjtrans`
Column major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*k`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*n`
Row major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*n`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*k`.

See *Matrix Storage* for more details.

lda
The leading dimension of `A`. It must be positive.

	`trans = transpose::nontrans`	`trans = transpose::trans` or `transpose::conjtrans`
Column major	`lda` must be at least `n`.	`lda` must be at least `k`.
Row major	`lda` must be at least `k`.	`lda` must be at least `n`.

b
Buffer holding input matrix `B`.

	`trans = transpose::nontrans`	`trans = transpose::trans` or `transpose::conjtrans`
Column major	`B` is an `n`-by-`k` matrix so the array `b` must have size at least `ldb*k`.	`B` is an `k`-by-`n` matrix so the array `b` must have size at least `ldb*n`
Row major	`B` is an `n`-by-`k` matrix so the array `b` must have size at least `ldb*n`.	`B` is an `k`-by-`n` matrix so the array `b` must have size at least `ldb*k`.

See *Matrix Storage* for more details.

ldb
The leading dimension of B. It must be positive.

	trans = transpose::nontrans	trans = transpose::trans or transpose::conjtrans
Column major	ldb must be at least n.	ldb must be at least k.
Row major	ldb must be at least k.	ldb must be at least n.

beta
Scaling factor for matrix C.

c
Buffer holding input/output matrix C. Must have size at least ldc*n. See *Matrix Storage* for more details

ldc
Leading dimension of C. Must be positive and at least n.

Output Parameters

c
Output buffer, overwritten by the updated C matrix.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

syr2k (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event syr2k(sycl::queue &queue,
                      oneapi::mkl::uplo upper_lower,
                      oneapi::mkl::transpose trans,
                      std::int64_t n,
                      std::int64_t k,
                      value_or_pointer<T> alpha,
                      const T *a,
                      std::int64_t lda,
                      const T *b,
                      std::int64_t ldb,
```

(continues on next page)

(continued from previous page)

```
                        value_or_pointer<T> beta,
                        T *c,
                        std::int64_t ldc,
                        const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event syr2k(sycl::queue &queue,
                        oneapi::mkl::uplo upper_lower,
                        oneapi::mkl::transpose trans,
                        std::int64_t n,
                        std::int64_t k,
                        value_or_pointer<T> alpha,
                        const T *a,
                        std::int64_t lda,
                        const T *b,
                        std::int64_t ldb,
                        value_or_pointer<T> beta,
                        T *c,
                        std::int64_t ldc,
                        const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `C`'s data is stored in its upper or lower triangle. See *oneMKL defined datatypes* for more details.

trans
Specifies the operation to apply, as described above. Conjugation is never performed, even if `trans = transpose::conjtrans`.

n
Number of rows and columns in `C`. The value of `n` must be at least zero.

k
Inner dimension of matrix multiplications.The value of `k` must be at least zero.

alpha
Scaling factor for the rank-2k update. See *Scalar Arguments in BLAS* for more details.

a
Pointer to input matrix `A`.

	`trans = transpose::nontrans`	`trans = transpose::trans` or `transpose::conjtrans`
Column major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*k`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*n`
Row major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*n`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*k`.

See *Matrix Storage* for more details.

lda

The leading dimension of `A`. It must be positive.

	`trans` = `transpose::nontrans`	`trans` = `transpose::trans` or `transpose::conjtrans`
Column major	`lda` must be at least `n`.	`lda` must be at least `k`.
Row major	`lda` must be at least `k`.	`lda` must be at least `n`.

b

Pointer to input matrix `B`.

	`trans` = `transpose::nontrans`	`trans` = `transpose::trans` or `transpose::conjtrans`
Column major	`B` is an `n`-by-`k` matrix so the array `b` must have size at least `ldb*k`.	`B` is an `k`-by-`n` matrix so the array `b` must have size at least `ldb*n`
Row major	`B` is an `n`-by-`k` matrix so the array `b` must have size at least `ldb*n`.	`B` is an `k`-by-`n` matrix so the array `b` must have size at least `ldb*k`.

See *Matrix Storage* for more details.

ldb

The leading dimension of `B`. It must be positive.

	`trans` = `transpose::nontrans`	`trans` = `transpose::trans` or `transpose::conjtrans`
Column major	`ldb` must be at least `n`.	`ldb` must be at least `k`.
Row major	`ldb` must be at least `k`.	`ldb` must be at least `n`.

beta

Scaling factor for matrix `C`. See *Scalar Arguments in BLAS* for more details.

c

Pointer to input/output matrix `C`. Must have size at least `ldc*n`. See *Matrix Storage* for more details

ldc

Leading dimension of `C`. Must be positive and at least `n`.

dependencies

List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

c
Pointer to the output matrix, overwritten by the updated `C` matrix.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 3 Routines*

trmm

Computes a matrix-matrix product where one input matrix is triangular and one input matrix is general.

Description

The `trmm` routines compute a scalar-matrix-matrix product where one of the matrices in the multiplication is triangular. The argument `left_right` determines if the triangular matrix, `A`, is on the left of the multiplication (`left_right = side::left`) or on the right (`left_right = side::right`). Depending on `left_right`.

There are two operations available, an in-place operation and an out-of-place operation. The in-place operation is defined as:

$$B \leftarrow alpha * op(A) * B$$

or

$$B \leftarrow alpha * B * op(A)$$

The out-of-place operation is defined as:

$$C \leftarrow alpha * op(A) * B + beta * C$$

or

$$C \leftarrow alpha * B * op(A) + beta * C$$

where:

op(`A`) is one of op(`A`) = A, or op(`A`) = $\mathtt{A}^{\mathrm{T}}$, or op(`A`) = $\mathtt{A}^{\mathrm{H}}$,

`alpha` and `beta` are scalars,

`A` is a triangular matrix, and `B` and `C` are general matrices.

Here `B` and `C` are `m` x `n` and `A` is either `m` x `m` or `n` x `n`, depending on `left_right`.

`trmm` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

trmm (Buffer Version)

In-place API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void trmm(sycl::queue &queue,
              oneapi::mkl::side left_right,
              oneapi::mkl::uplo upper_lower,
              oneapi::mkl::transpose transa,
              oneapi::mkl::diag unit_diag,
              std::int64_t m,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &b,
              std::int64_t ldb)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void trmm(sycl::queue &queue,
              oneapi::mkl::side left_right,
              oneapi::mkl::uplo upper_lower,
              oneapi::mkl::transpose transa,
              oneapi::mkl::diag unit_diag,
              std::int64_t m,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &b,
              std::int64_t ldb)
}
```

Input Parameters

queue
The queue where the routine should be executed.

left_right
Specifies whether `A` is on the left side of the multiplication (`side::left`) or on the right side (`side::right`). See *oneMKL defined datatypes* for more details.

upper_lower
Specifies whether the matrix `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans
Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_diag
Specifies whether `A` is assumed to be unit triangular (all diagonal elements are 1). See *oneMKL defined datatypes* for more details.

m
Specifies the number of rows of `B`. The value of `m` must be at least zero.

n
Specifies the number of columns of `B`. The value of `n` must be at least zero.

alpha
Scaling factor for the matrix-matrix product.

a
Buffer holding input matrix `A`. Must have size at least `lda*m` if `left_right = side::left`, or `lda*n` if `left_right = side::right`. See *Matrix Storage* for more details.

lda
Leading dimension of `A`. Must be at least `m` if `left_right = side::left`, and at least `n` if `left_right = side::right`. Must be positive.

b
Buffer holding input/output matrix `B`. Must have size at least `ldb*n` if column major layout is used to store matrices or at least `ldb*m` if row major layout is used to store matrices. See *Matrix Storage* for more details.

ldb
Leading dimension of `B`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if row major layout is used to store matrices.

Output Parameters

b
Output buffer, overwritten by `alpha`*op(`A`)*`B` or `alpha`*`B`*op(`A`).

Notes

If `alpha` = 0, matrix `B` is set to zero, and `A` and `B` do not need to be initialized at entry.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Out-of-place API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void trmm(sycl::queue &queue,
              oneapi::mkl::side left_right,
              oneapi::mkl::uplo upper_lower,
              oneapi::mkl::transpose trans,
              oneapi::mkl::diag unit_diag,
              std::int64_t m,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &b,
              std::int64_t ldb,
              T beta,
              sycl::buffer<T,1> &c,
              std::int64_t ldc)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void trmm(sycl::queue &queue,
              oneapi::mkl::side left_right,
              oneapi::mkl::uplo upper_lower,
              oneapi::mkl::transpose trans,
              oneapi::mkl::diag unit_diag,
              std::int64_t m,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &b,
```

(continues on next page)

(continued from previous page)

```
                std::int64_t ldb,
                T beta,
                sycl::buffer<T,1> &c,
                std::int64_t ldc)
}
```

Input Parameters

queue
The queue where the routine should be executed.

left_right
Specifies whether `A` is on the left side of the multiplication (`side::left`) or on the right side (`side::right`). See *oneMKL defined datatypes* for more details.

upper_lower
Specifies whether the matrix `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans
Specifies op(`A`), the transposition operation applied to matrix `A`. See *oneMKL defined datatypes* for more details.

unit_diag
Specifies whether `A` is assumed to be unit triangular (all diagonal elements are 1). See *oneMKL defined datatypes* for more details.

m
Specifices the number of rows of `B`. The value of `m` must be at least zero.

n
Specifies the Number of columns of `B`. The value of `n` must be at least zero.

alpha
Scaling factor for matrix-matrix product.

a
Buffer holding input matrix `A`. Must have size at least `lda*m` if `left_right = side::left` or `lda*n` if `left_right = side::right`. See *Matrix Storage* for more details.

lda
Leading dimension of `A`. Must be at least `m` if `left_right = side::left` or at least `n` if `left_right = side::right`. Must be positive.

b
Buffer holding input matrix `B`. Must have size at least `ldb*n` if column major layout or at least `ldb*m` if row major layout is used. See *Matrix Storage* for more details.

ldb
Leading dimension of matrix `B`. It must be positive and at least `m` if column major layout or at least `n` if row major layout is used.

beta
Scaling factor for matrix `C`.

c
Buffer holding input/output matrix `C`. Size of the buffer must be at least `ldc*n` if column major layout or at least `ldc*m` if row major layout is used. See *Matrix Storage* for more details.

ldc

Leading dimension of matrix `C`. Must be at least `m` if column major layout or at least `n` if row major layout is used. Must be positive.

Output Parameters

c

Output buffer overwritten by `alpha*op(A)*B + beta*C` if `left_right = side::left` or `alpha*B*op(A) + beta*C` if `left_right = side::right`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

trmm (USM Version)

In-place API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event trmm(sycl::queue &queue,
                     oneapi::mkl::side left_right,
                     oneapi::mkl::uplo upper_lower,
                     oneapi::mkl::transpose transa,
                     oneapi::mkl::diag unit_diag,
                     std::int64_t m,
                     std::int64_t n,
                     value_or_pointer<T> alpha,
                     const T *a,
                     std::int64_t lda,
                     T *b,
                     std::int64_t ldb,
                     const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event trmm(sycl::queue &queue,
                     oneapi::mkl::side left_right,
                     oneapi::mkl::uplo upper_lower,
                     oneapi::mkl::transpose transa,
```

(continues on next page)

(continued from previous page)

```
                        oneapi::mkl::diag unit_diag,
                        std::int64_t m,
                        std::int64_t n,
                        value_or_pointer<T> alpha,
                        const T *a,
                        std::int64_t lda,
                        T *b,
                        std::int64_t ldb,
                        const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

left_right
Specifies whether `A` is on the left side of the multiplication (`side::left`) or on the right side (`side::right`). See *oneMKL defined datatypes* for more details.

upper_lower
Specifies whether the matrix `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans
Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_diag
Specifies whether `A` is assumed to be unit triangular (all diagonal elements are 1). See *oneMKL defined datatypes* for more details.

m
Specifies the number of rows of `B`. The value of `m` must be at least zero.

n
Specifies the number of columns of `B`. The value of `n` must be at least zero.

alpha
Scaling factor for the matrix-matrix product. See *Scalar Arguments in BLAS* for more details.

a
Pointer to input matrix `A`. Must have size at least `lda*m` if `left_right = side::left`, or `lda*n` if `left_right = side::right`. See *Matrix Storage* for more details.

lda
Leading dimension of `A`. Must be at least `m` if `left_right = side::left`, and at least `n` if `left_right = side::right`. Must be positive.

b
Pointer to input/output matrix `B`. Must have size at least `ldb*n` if column major layout is used to store matrices or at least `ldb*m` if row major layout is used to store matrices. See *Matrix Storage* for more details.

ldb
Leading dimension of `B`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if row major layout is used to store matrices.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

b
Pointer to the output matrix, overwritten by `alpha`*op(`A`)*`B` or `alpha`*`B`*op(`A`).

Notes

If `alpha` = 0, matrix `B` is set to zero, and `A` and `B` do not need to be initialized at entry.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Out-of-place API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void trmm(sycl::queue &queue,
              oneapi::mkl::side left_right,
              oneapi::mkl::uplo upper_lower,
              oneapi::mkl::transpose trans,
              oneapi::mkl::diag unit_diag,
              std::int64_t m,
              std::int64_t n,
              value_or_pointer<T> alpha,
              const T *a,
              std::int64_t lda,
              const T *b,
              std::int64_t ldb,
              value_or_pointer<T> beta,
              T *c,
              std::int64_t ldc,
              const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    void trmm(sycl::queue &queue,
              oneapi::mkl::side left_right,
              oneapi::mkl::uplo upper_lower,
              oneapi::mkl::transpose trans,
              oneapi::mkl::diag unit_diag,
              std::int64_t m,
              std::int64_t n,
              value_or_pointer<T> alpha,
              const T *a,
              std::int64_t lda,
              const T *b,
              std::int64_t ldb,
              value_or_pointer<T> beta,
              T *c,
              std::int64_t ldc,
              const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

left_right
Specifies whether `A` is on the left side of the multiplication (`side::left`) or on the right side (`side::right`). See *oneMKL defined datatypes* for more details.

upper_lower
Specifies whether the matrix `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans
Specifies op(`A`), the transposition operation applied to matrix `A`. See *oneMKL defined datatypes* for more details.

unit_diag
Specifies whether `A` is assumed to be unit triangular (all diagonal elements are 1). See *oneMKL defined datatypes* for more details.

m
Specifices the number of rows of `B`. The value of `m` must be at least zero.

n
Specifies the Number of columns of `B`. The value of `n` must be at least zero.

alpha
Scaling factor for matrix-matrix product. See *Scalar Arguments in BLAS* for more details.

a
Pointer to input matrix `A`. Must have size at least `lda*m` if `left_right = side::left` or `lda*n` if `left_right = side::right`. See *Matrix Storage* for more details.

lda
Leading dimension of `A`. Must be at least `m` if `left_right = side::left` or at least `n` if `left_right = side::right`. Must be positive.

b
Pointer to input matrix `B`. Must have size at least `ldb*n` if column major layout or at least `ldb*m` if row major

layout is used. See *Matrix Storage* for more details.

ldb
Leading dimension of matrix `B`. It must be positive and at least `m` if column major layout or at least `n` if row major layout is used.

beta
Scaling factor for matrix `C`. See *Scalar Arguments in BLAS* for more details.

c
Pointer to input/output matrix `C`. Must have size at least `ldc*n` if column major layout or at least `ldc*m` if row major layout is used. See *Matrix Storage* for more details.

ldc
Leading dimension of matrix `C`. Must be at least `m` if column major layout or at least `n` if row major layout is used. Must be positive.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

c
Pointer to the output matrix, overwritten by `alpha*op(A)*B + beta*C` if `left_right = side::left` or `alpha*B*op(A) + beta*C` if `left_right = side::right`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 3 Routines*

trsm

Solves a triangular matrix equation (forward or backward solve).

Description

The `trsm` routines solves a triangular matrix equations. There are two operations available, an in-place operation and an out-of-place operation. The in-place operation solves for `X` in:

$$op(A) * X = alpha * B$$

or

$$X * op(A) = alpha * B$$

The out-of-place operation solves for `X` and then adds that solution to a scaled matrix `C`:

$$op(A) * X = alpha * B, C \leftarrow X + beta * C$$

or

$$X * op(A) = alpha * B, C \leftarrow X + beta * C$$

where:

op(`A`) is one of op(`A`) = `A`, or op(`A`) = $\texttt{A}^{T}$, or op(`A`) = $\texttt{A}^{H}$,

`alpha` and `beta` are scalars,

`A` is a triangular matrix, and

`B`, `X`, and `C` are `m` x `n` general matrices.

`A` is either `m` x `m` or `n` x `n`, depending on whether it multiplies `X` on the left or right.

For the in-place operation, the matrix `B` is overwritten by the solution matrix `X` on return. For the out-of-place operation, `B` remains untouched and the solution is added to a scaled `C` matrix.

`trsm` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

trsm (Buffer Version)

In-place API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void trsm(sycl::queue &queue,
              oneapi::mkl::side left_right,
              oneapi::mkl::uplo upper_lower,
              oneapi::mkl::transpose transa,
              oneapi::mkl::diag unit_diag,
```

(continues on next page)

(continued from previous page)

```
                std::int64_t m,
                std::int64_t n,
                T alpha,
                sycl::buffer<T,1> &a,
                std::int64_t lda,
                sycl::buffer<T,1> &b,
                std::int64_t ldb)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void trsm(sycl::queue &queue,
                oneapi::mkl::side left_right,
                oneapi::mkl::uplo upper_lower,
                oneapi::mkl::transpose transa,
                oneapi::mkl::diag unit_diag,
                std::int64_t m,
                std::int64_t n,
                T alpha,
                sycl::buffer<T,1> &a,
                std::int64_t lda,
                sycl::buffer<T,1> &b,
                std::int64_t ldb)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

left_right
: Specifies whether `A` multiplies `X` on the left (`side::left`) or on the right (`side::right`). See *oneMKL defined datatypes* for more details.

upper_lower
: Specifies whether the matrix `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans
: Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_diag
: Specifies whether `A` is assumed to be unit triangular (all diagonal elements are 1). See *oneMKL defined datatypes* for more details.

m
: Specifies the number of rows of `B`. The value of `m` must be at least zero.

n
: Specifies the number of columns of `B`. The value of `n` must be at least zero.

alpha
: Scaling factor for the solution.

a
: Buffer holding input matrix `A`. Must have size at least `lda*m` if `left_right = side::left`, or `lda*n` if `left_right = side::right`. See *Matrix Storage* for more details.

lda
Leading dimension of `A`. Must be at least `m` if `left_right = side::left`, and at least `n` if `left_right = side::right`. Must be positive.

b
Buffer holding input/output matrix `B`. Must have size at least `ldb*n` if column major layout is used to store matrices or at least `ldb*m` if row major layout is used to store matrices. See *Matrix Storage* for more details.

ldb
Leading dimension of `B`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if row major layout is used to store matrices.

Output Parameters

b
Output buffer. Overwritten by the solution matrix `X`.

Notes

If `alpha` = 0, matrix `B` is set to zero, and `A` and `B` do not need to be initialized at entry.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Out-of-place API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void trsm(sycl::queue &queue,
              oneapi::mkl::side left_right,
              oneapi::mkl::uplo upper_lower,
              oneapi::mkl::transpose trans,
              oneapi::mkl::diag unit_diag,
              std::int64_t m,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &b,
              std::int64_t ldb,
```

(continues on next page)

(continued from previous page)

```
            T beta,
            sycl::buffer<T,1> &c,
            std::int64_t ldc)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void trsm(sycl::queue &queue,
            oneapi::mkl::side left_right,
            oneapi::mkl::uplo upper_lower,
            oneapi::mkl::transpose trans,
            oneapi::mkl::diag unit_diag,
            std::int64_t m,
            std::int64_t n,
            T alpha,
            sycl::buffer<T,1> &a,
            std::int64_t lda,
            sycl::buffer<T,1> &b,
            std::int64_t ldb,
            T beta,
            sycl::buffer<T,1> &c,
            std::int64_t ldc)
}
```

Input Parameters

queue
The queue where the routine should be executed.

left_right
Specifies whether `A` is on the left side of the matrix solve (`side::left`) or on the right side (`side::right`). See *oneMKL defined datatypes* for more details.

upper_lower
Specifies whether the matrix `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans
Specifies op(`A`), the transposition operation applied to matrix `A`. See *oneMKL defined datatypes* for more details.

unit_diag
Specifies whether `A` is assumed to be unit triangular (all diagonal elements are 1). See *oneMKL defined datatypes* for more details.

m
Specifices the number of rows of `B`. The value of `m` must be at least zero.

n
Specifies the Number of columns of `B`. The value of `n` must be at least zero.

alpha
Scaling factor for the solution.

a
Buffer holding input matrix `A`. Must have size at least `lda*m` if `left_right = side::left` or `lda*n` if `left_right = side::right`. See *Matrix Storage* for more details.

lda
Leading dimension of `A`. Must be at least `m` if `left_right = side::left` or at least `n` if `left_right = side::right`. Must be positive.

b
Buffer holding input matrix `B`. Must have size at least `ldb*n` if column major layout or at least `ldb*m` if row major layout is used. See *Matrix Storage* for more details.

ldb
Leading dimension of matrix `B`. It must be positive and at least `m` if column major layout or at least `n` if row major layout is used.

beta
Scaling factor for matrix `C`.

c
Buffer holding input/output matrix `C`. Size of the buffer must be at least `ldc*n` if column major layout or at least `ldc*m` if row major layout is used. See *Matrix Storage* for more details.

ldc
Leading dimension of matrix `C`. Must be at least `m` if column major layout or at least `n` if row major layout is used. Must be positive.

Output Parameters

c
Output buffer overwritten by solution matrix `X + beta*C`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

trsm (USM Version)

In-place API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event trsm(sycl::queue &queue,
                     oneapi::mkl::side left_right,
                     oneapi::mkl::uplo upper_lower,
                     oneapi::mkl::transpose transa,
                     oneapi::mkl::diag unit_diag,
```

(continues on next page)

(continued from previous page)

```
                    std::int64_t m,
                    std::int64_t n,
                    value_or_pointer<T> alpha,
                    const T *a,
                    std::int64_t lda,
                    T *b,
                    std::int64_t ldb,
                    const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event trsm(sycl::queue &queue,
                    oneapi::mkl::side left_right,
                    oneapi::mkl::uplo upper_lower,
                    oneapi::mkl::transpose transa,
                    oneapi::mkl::diag unit_diag,
                    std::int64_t m,
                    std::int64_t n,
                    value_or_pointer<T> alpha,
                    const T *a,
                    std::int64_t lda,
                    T *b,
                    std::int64_t ldb,
                    const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
: The queue where the routine should be executed.

left_right
: Specifies whether `A` multiplies `X` on the left (`side::left`) or on the right (`side::right`). See *oneMKL defined datatypes* for more details.

upper_lower
: Specifies whether the matrix `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

transa
: Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_diag
: Specifies whether `A` is assumed to be unit triangular (all diagonal elements are 1). See *oneMKL defined datatypes* for more details.

m
: Specifies the number of rows of `B`. The value of `m` must be at least zero.

n
: Specifies the number of columns of `B`. The value of `n` must be at least zero.

alpha
: Scaling factor for the solution. See *Scalar Arguments in BLAS* for more details.

a
Pointer to input matrix `A`. Must have size at least `lda*m` if `left_right = side::left`, or `lda*n` if `left_right = side::right`. See *Matrix Storage* for more details.

lda
Leading dimension of `A`. Must be at least `m` if `left_right = side::left`, and at least `n` if `left_right = side::right`. Must be positive.

b
Pointer to input/output matrix `B`. Must have size at least `ldb*n` if column major layout is used to store matrices or at least `ldb*m` if row major layout is used to store matrices. See *Matrix Storage* for more details.

ldb
Leading dimension of `B`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if row major layout is used to store matrices.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

b
Pointer to the output matrix. Overwritten by the solution matrix `X`.

Notes

If `alpha` = 0, matrix `B` is set to zero, and `A` and `B` do not need to be initialized at entry.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Out-of-place API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void trsm(sycl::queue &queue,
              oneapi::mkl::side left_right,
              oneapi::mkl::uplo upper_lower,
              oneapi::mkl::transpose trans,
              oneapi::mkl::diag unit_diag,
              std::int64_t m,
              std::int64_t n,
              value_or_pointer<T> alpha,
              const T *a,
              std::int64_t lda,
              const T *b,
              std::int64_t ldb,
              value_or_pointer<T> beta,
              T *c,
              std::int64_t ldc,
              const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    void trsm(sycl::queue &queue,
              oneapi::mkl::side left_right,
              oneapi::mkl::uplo upper_lower,
              oneapi::mkl::transpose trans,
              oneapi::mkl::diag unit_diag,
              std::int64_t m,
              std::int64_t n,
              value_or_pointer<T> alpha,
              const T *a,
              std::int64_t lda,
              const T *b,
              std::int64_t ldb,
              value_or_pointer<T> beta,
              T *c,
              std::int64_t ldc,
              const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
: The queue where the routine should be executed.

left_right
: Specifies whether `A` is on the left side of the matrix solve (`side::left`) or on the right side (`side::right`). See *oneMKL defined datatypes* for more details.

upper_lower
: Specifies whether the matrix `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans
Specifies op(`A`), the transposition operation applied to matrix `A`. See *oneMKL defined datatypes* for more details.

unit_diag
Specifies whether `A` is assumed to be unit triangular (all diagonal elements are 1). See *oneMKL defined datatypes* for more details.

m
Specifices the number of rows of `B`. The value of `m` must be at least zero.

n
Specifies the Number of columns of `B`. The value of `n` must be at least zero.

alpha
Scaling factor for the solution. See *Scalar Arguments in BLAS* for more details.

a
Pointer to input matrix `A`. Must have size at least `lda*m` if `left_right = side::left` or `lda*n` if `left_right = side::right`. See *Matrix Storage* for more details.

lda
Leading dimension of `A`. Must be at least `m` if `left_right = side::left` or at least `n` if `left_right = side::right`. Must be positive.

b
Pointer to input matrix `B`. Must have size at least `ldb*n` if column major layout or at least `ldb*m` if row major layout is used. See *Matrix Storage* for more details.

ldb
Leading dimension of matrix `B`. It must be positive and at least `m` if column major layout or at least `n` if row major layout is used.

beta
Scaling factor for matrix `C`. See *Scalar Arguments in BLAS* for more details.

c
Pointer to input/output matrix `C`. Must have size at least `ldc*n` if column major layout or at least `ldc*m` if row major layout is used. See *Matrix Storage* for more details.

ldc
Leading dimension of matrix `C`. Must be at least `m` if column major layout or at least `n` if row major layout is used. Must be positive.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

c
Pointer to the output matrix, overwritten by the solution matrix `X + beta*C`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 3 Routines*

Parent topic: *BLAS Routines*

BLAS-like Extensions

oneAPI Math Kernel Library DPC++ provides additional routines to extend the functionality of the BLAS routines. These include routines to compute many independent vector-vector and matrix-matrix operations.

The following table lists the BLAS-like extensions with their descriptions.

Routines	Description
axpy_batch	Computes groups of vector-scalar products added to a vector.
gemm_batch	Computes groups of matrix-matrix products with general matrices.
trsm_batch	Solves a triangular matrix equation for a group of matrices.
gemmt	Computes a matrix-matrix product with general matrices, but updates only the upper or lower triangular part of the result matrix.
gemm_bias	Computes a matrix-matrix product using general integer matrices with bias
imatcopy	Computes an in-place matrix transposition or copy.
omatcopy	Computes an out-of-place matrix transposition or copy.
omatcopy2	Computes a two-strided out-of-place matrix transposition or copy.
omatadd	Computes scaled matrix addition with possibly transposed arguments.
imatcopy_batch	Computes groups of in-place matrix transposition or copy operations.
omatcopy_batch	Computes groups of out-of-place matrix transposition or copy operations.
omatadd_batc	Computes groups of scaled matrix additions.

axpy_batch

Computes a group of `axpy` operations.

Description

The `axpy_batch` routines are batched versions of *axpy*, performing multiple `axpy` operations in a single call. Each `axpy` operation adds a scalar-vector product to a vector.

`axpy_batch` supports the following precisions for data.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

axpy_batch (Buffer Version)

Description

The buffer version of `axpy_batch` supports only the strided API.

The strided API operation is defined as:

```
for i = 0 ... batch_size - 1
   X and Y are vectors at offset i * stridex, i * stridey in x and y
   Y := alpha * X + Y
end for
```

where:

`alpha` is scalar,

`X` and `Y` are vectors.

Strided API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void axpy_batch(sycl::queue &queue,
                    std::int64_t n,
                    T alpha,
                    sycl::buffer<T,
                    1> &x,
                    std::int64_t incx,
                    std::int64_t stridex,
                    sycl::buffer<T,
                    1> &y,
                    std::int64_t incy,
                    std::int64_t stridey,
                    std::int64_t batch_size)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void axpy_batch(sycl::queue &queue,
                    std::int64_t n,
                    T alpha,
                    sycl::buffer<T,
                    1> &x,
                    std::int64_t incx,
                    std::int64_t stridex,
                    sycl::buffer<T,
                    1> &y,
                    std::int64_t incy,
                    std::int64_t stridey,
                    std::int64_t batch_size)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

n
: Number of elements in `X` and `Y`.

alpha
: Specifies the scalar `alpha`.

x
: Buffer holding input vectors `X` with size `stridex` * `batch_size`.

incx
: Stride of vector `X`. Must not be zero.

stridex
: Stride between different `X` vectors. Must be at least zero.

y
: Buffer holding input/output vectors `Y` with size `stridey` * `batch_size`.

incy
: Stride of vector `Y`. Must not be zero.

stridey
: Stride between different `Y` vectors. Must be at least (1 + (`n`-1)*abs(`incy`)).

batch_size
: Specifies the number of `axpy` operations to perform.

Output Parameters

y
Output buffer, overwritten by `batch_size` `axpy` operations of the form `alpha` * `X` + `Y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

axpy_batch (USM Version)

Description

The USM version of `axpy_batch` supports the group API and strided API.

The group API operation is defined as

```
idx = 0
for i = 0 ... group_count - 1
    for j = 0 ... group_size - 1
        X and Y are vectors in x[idx] and y[idx]
        Y := alpha[i] * X + Y
        idx := idx + 1
    end for
end for
```

The strided API operation is defined as

```
for i = 0 ... batch_size - 1
   X and Y are vectors at offset i * stridex, i * stridey in x and y
   Y := alpha * X + Y
end for
```

where:

`alpha` is scalar,

`X` and `Y` are vectors.

For group API, `x` and `y` arrays contain the pointers for all the input vectors. The total number of vectors in `x` and `y` are given by:

$$total\_batch\_count = \sum_{i=0}^{group\_count-1} group\_size[i]$$

For strided API, `x` and `y` arrays contain all the input vectors. The total number of vectors in `x` and `y` are given by the `batch_size` parameter.

Group API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event axpy_batch(sycl::queue &queue,
                           const std::int64_t *n,
                           const T *alpha,
                           const T **x,
                           const std::int64_t *incx,
                           T **y,
                           const std::int64_t *incy,
                           std::int64_t group_count,
                           const std::int64_t *group_size,
                           const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event axpy_batch(sycl::queue &queue,
                           const std::int64_t *n,
                           const T *alpha,
                           const T **x,
                           const std::int64_t *incx,
                           T **y,
                           const std::int64_t *incy,
                           std::int64_t group_count,
                           const std::int64_t *group_size,
                           const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
: The queue where the routine should be executed.

n
: Array of `group_count` integers. `n[i]` specifies the number of elements in vectors `X` and `Y` for every vector in group `i`.

alpha
: Array of `group_count` scalar elements. `alpha[i]` specifies the scaling factor for vector `X` in group `i`.

x
: Array of pointers to input vectors `X` with size `total_batch_count`. The size of array allocated for the `X` vector of the group `i` must be at least (1 + (`n[i]` – 1)*abs(`incx[i]`)). See *Matrix Storage* for more details.

incx
: Array of `group_count` integers. `incx[i]` specifies the stride of vector `X` in group `i`. Must not be zero.

y
: Array of pointers to input/output vectors `Y` with size `total_batch_count`. The size of array allocated for the `Y` vector of the group `i` must be at least (1 + (`n[i]` – 1)*abs(`incy[i]`)). See *Matrix Storage* for more details.

incy
Array of `group_count` integers. `incy[i]` specifies the stride of vector `Y` in group `i`. Must not be zero.

group_count
Number of groups. Must be at least 0.

group_size
Array of `group_count` integers. `group_size[i]` specifies the number of `axpy` operations in group `i`. Each element in `group_size` must be at least 0.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

y
Array of pointers holding the `Y` vectors, overwritten by `total_batch_count` `axpy` operations of the form `alpha * X + Y`.

Return Values

Output event to wait on to ensure computation is complete.

Strided API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event axpy_batch(sycl::queue &queue,
                           std::int64_t n,
                           value_or_pointer<T> alpha,
                           const T *x,
                           std::int64_t incx,
                           std::int64_t stridex,
                           T *y,
                           std::int64_t incy,
                           std::int64_t stridey,
                           std::int64_t batch_size,
                           const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event axpy_batch(sycl::queue &queue,
                           std::int64_t n,
                           value_or_pointer<T> alpha,
                           const T *x,
                           std::int64_t incx,
                           std::int64_t stridex,
                           T *y,
                           std::int64_t incy,
                           std::int64_t stridey,
                           std::int64_t batch_size,
```

(continues on next page)

(continued from previous page)

```
                                const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

n
Number of elements in `X` and `Y`.

alpha
Specifies the scalar `alpha`. See *Scalar Arguments in BLAS* for more details.

x
Pointer to input vectors `X` with size `stridex` * `batch_size`.

incx
Stride of vector `X`. Must not be zero.

stridex
Stride between different `X` vectors. Must be at least zero.

y
Pointer to input/output vectors `Y` with size `stridey` * `batch_size`.

incy
Stride of vector `Y`. Must not be zero.

stridey
Stride between different `Y` vectors. Must be at least (1 + (`n`-1)*abs(`incy`)).

batch_size
Specifies the number of `axpy` operations to perform.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

y
Output vectors, overwritten by `batch_size` `axpy` operations of the form `alpha` * `X` + `Y`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic:*BLAS-like Extensions*

axpby

Computes a vector-scalar product added to a scaled-vector.

Description

The `axpby` routines compute two scalar-vector product and add them:

$$y \leftarrow beta * y + alpha * x$$

where `x` and `y` are vectors of `n` elements and `alpha` and `beta` are scalars.

`axpby` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

axpby (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void axpby(sycl::queue &queue,
               std::int64_t n,
               T alpha,
               sycl::buffer<T,1> &x, std::int64_t incx,
               T beta,
               sycl::buffer<T,1> &y, std::int64_t incy)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void axpby(sycl::queue &queue,
               std::int64_t n,
               T alpha,
               sycl::buffer<T,1> &x, std::int64_t incx,
               T beta,
               sycl::buffer<T,1> &y, std::int64_t incy)
}
```

Input Parameters

queue
The queue where the routine should be executed.

n
Number of elements in vector `x` and `y`.

alpha
Specifies the scalar `alpha`.

x
Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` – 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
Stride between two consecutive elements of the `x` vector.

beta
Specifies the scalar `beta`.

y
Buffer holding input vector `y`. The buffer must be of size at least (1 + (`n` – 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
Stride between two consecutive elements of the `y` vector.

Output Parameters

y
Buffer holding the updated vector `y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

axpby (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event axpby(sycl::queue &queue,
                      std::int64_t n,
                      value_or_pointer<T> alpha,
                      const T *x, std::int64_t incx,
                      value_or_pointer<T> beta,
                      T *y, std::int64_t incy,
                      const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event axpby(sycl::queue &queue,
                      std::int64_t n,
                      value_or_pointer<T> alpha,
                      const T *x, std::int64_t incx,
                      value_or_pointer<T> beta,
                      T *y, std::int64_t incy,
                      const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
: The queue where the routine should be executed.

n
: Number of elements in vector `x` and `y`.

alpha
: Specifies the scalar alpha. See *Scalar Arguments in BLAS* for more details.

beta
: Specifies the scalar beta. See *Scalar Arguments in BLAS* for more details.

x
: Pointer to the input vector `x`. The allocated memory must be of size at least (1 + (`n` – 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx
: Stride between consecutive elements of the `x` vector.

y
: Pointer to the input vector `y`. The allocated memory must be of size at least (1 + (`n` – 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy
: Stride between consecutive elements of the `y` vector.

dependencies
: List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

y
Array holding the updated vector `y`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS-like Extensions*

copy_batch

Computes a group of `copy` operations.

Description

The `copy_batch` routines are batched versions of *copy*, performing multiple `copy` operations in a single call. Each `copy` operation copies one vector to another.

`copy_batch` supports the following precisions for data.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

copy_batch (Buffer Version)

Description

The buffer version of `copy_batch` supports only the strided API.

The strided API operation is defined as:

```
for i = 0 ... batch_size - 1
   X and Y are vectors at offset i * stridex, i * stridey in x and y
   Y := X
end for
```

where:

`X` and `Y` are vectors.

Strided API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void copy_batch(sycl::queue &queue,
                    std::int64_t n,
                    sycl::buffer<T,
                    1> &x,
                    std::int64_t incx,
                    std::int64_t stridex,
                    sycl::buffer<T,
                    1> &y,
                    std::int64_t incy,
                    std::int64_t stridey,
                    std::int64_t batch_size)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void copy_batch(sycl::queue &queue,
                    std::int64_t n,
                    sycl::buffer<T,
                    1> &x,
                    std::int64_t incx,
                    std::int64_t stridex,
                    sycl::buffer<T,
                    1> &y,
                    std::int64_t incy,
                    std::int64_t stridey,
                    std::int64_t batch_size)
}
```

Input Parameters

queue
The queue where the routine should be executed.

n
Number of elements in `X` and `Y`.

x
Buffer holding input vectors `X` with size `stridex` * `batch_size`.

incx
Stride of vector `X`. Must not be zero.

stridex
Stride between different `X` vectors. Must be at least zero.

y
Buffer holding input/output vectors `Y` with size `stridey` * `batch_size`.

incy
Stride of vector `Y`. Must not be zero.

stridey
Stride between different `Y` vectors. Must be at least (1 + (`n`-1)*abs(`incy`)).

batch_size
Specifies the number of `copy` operations to perform.

Output Parameters

y
Output buffer, overwritten by `batch_size` `copy` operations.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

copy_batch (USM Version)

Description

The USM version of `copy_batch` supports the group API and strided API.

The group API operation is defined as

```
idx = 0
for i = 0 ... group_count - 1
    for j = 0 ... group_size - 1
        X and Y are vectors in x[idx] and y[idx]
        Y := X
        idx := idx + 1
    end for
end for
```

The strided API operation is defined as

```
for i = 0 ... batch_size - 1
   X and Y are vectors at offset i * stridex, i * stridey in x and y
   Y := X
end for
```

where:

`X` and `Y` are vectors.

For group API, `x` and `y` arrays contain the pointers for all the input vectors. The total number of vectors in `x` and `y` are given by:

$$total\_batch\_count = \sum_{i=0}^{group\_count-1} group\_size[i]$$

For strided API, `x` and `y` arrays contain all the input vectors. The total number of vectors in `x` and `y` are given by the `batch_size` parameter.

Group API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event copy_batch(sycl::queue &queue,
                           const std::int64_t *n,
                           const T **x,
                           const std::int64_t *incx,
                           T **y,
                           const std::int64_t *incy,
                           std::int64_t group_count,
                           const std::int64_t *group_size,
                           const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event copy_batch(sycl::queue &queue,
                           const std::int64_t *n,
                           const T **x,
                           const std::int64_t *incx,
                           T **y,
                           const std::int64_t *incy,
                           std::int64_t group_count,
                           const std::int64_t *group_size,
                           const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

n
Array of `group_count` integers. `n[i]` specifies the number of elements in vectors `X` and `Y` for every vector in group `i`.

x
Array of pointers to input vectors `X` with size `total_batch_count`. The size of array allocated for the `X` vector of the group `i` must be at least (1 + (`n[i]` – 1)*abs(`incx[i]`)). See *Matrix Storage* for more details.

incx
Array of `group_count` integers. `incx[i]` specifies the stride of vector `X` in group `i`. Must not be zero.

y
Array of pointers to input/output vectors `Y` with size `total_batch_count`. The size of array allocated for the `Y` vector of the group `i` must be at least (1 + (`n[i]` – 1)*abs(`incy[i]`)). See *Matrix Storage* for more details.

incy
Array of `group_count` integers. `incy[i]` specifies the stride of vector `Y` in group `i`. Must not be zero.

group_count
Number of groups. Must be at least 0.

group_size
Array of `group_count` integers. `group_size[i]` specifies the number of `copy` operations in group `i`. Each element in `group_size` must be at least 0.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

y
Array of pointers holding the `Y` vectors, overwritten by `total_batch_count` `copy` operations.

Return Values

Output event to wait on to ensure computation is complete.

Strided API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event copy_batch(sycl::queue &queue,
                           std::int64_t n,
                           const T *x,
                           std::int64_t incx,
                           std::int64_t stridex,
                           T *y,
                           std::int64_t incy,
                           std::int64_t stridey,
                           std::int64_t batch_size,
                           const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event copy_batch(sycl::queue &queue,
                           std::int64_t n,
                           const T *x,
                           std::int64_t incx,
                           std::int64_t stridex,
                           T *y,
                           std::int64_t incy,
                           std::int64_t stridey,
                           std::int64_t batch_size,
                           const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
: The queue where the routine should be executed.

n
: Number of elements in `X` and `Y`.

x
: Pointer to input vectors `X` with size `stridex` * `batch_size`.

incx
: Stride of vector `X`. Must not be zero.

stridex
: Stride between different `X` vectors. Must be at least zero.

y
: Pointer to input/output vectors `Y` with size `stridey` * `batch_size`.

incy
Stride of vector `Y`. Must not be zero.

stridey
Stride between different `Y` vectors. Must be at least (1 + (`n`-1)*abs(`incy`)).

batch_size
Specifies the number of `copy` operations to perform.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

y
Output vectors, overwritten by `batch_size` `copy` operations

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic:*BLAS-like Extensions*

dgmm_batch

Computes a group of `dgmm` operations.

Description

The `dgmm_batch` routines perform multiple diagonal matrix-matrix product operations in a single call.

`dgmm_batch` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

dgmm_batch (Buffer Version)

Description

The buffer version of `dgmm_batch` supports only the strided API.

The strided API operation is defined as:

```
for i = 0 ... batch_size - 1
    A and C are matrices at offset i * stridea in a, i * stridec in c.
    X is a vector at offset i * stridex in x
    C := diag(X) * A or  C = A * diag(X)
end for
```

where:

`A` is a matrix,

`X` is a diagonal matrix stored as a vector

The `a` and `x` buffers contain all the input matrices. The stride between matrices is given by the stride parameter. The total number of matrices in `a` and `x` buffers is given by the `batch_size` parameter.

Strided API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void dgmm_batch(sycl::queue &queue,
                    oneapi::mkl::side left_right,
                    std::int64_t m,
                    std::int64_t n,
                    sycl::buffer<T,1> &a,
                    std::int64_t lda,
                    std::int64_t stridea,
                    sycl::buffer<T,1> &x,
                    std::int64_t incx,
                    std::int64_t stridex,
                    sycl::buffer<T,1> &c,
                    std::int64_t ldc,
                    std::int64_t stridec,
                    std::int64_t batch_size)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void dgmm_batch(sycl::queue &queue,
                    oneapi::mkl::side left_right,
                    std::int64_t m,
                    std::int64_t n,
                    sycl::buffer<T,1> &a,
                    std::int64_t lda,
                    std::int64_t stridea,
                    sycl::buffer<T,1> &x,
                    std::int64_t incx,
```

(continues on next page)

(continued from previous page)

```
                    std::int64_t stridex,
                    sycl::buffer<T,1> &c,
                    std::int64_t ldc,
                    std::int64_t stridec,
                    std::int64_t batch_size)
}
```

Input Parameters

queue
The queue where the routine should be executed.

left_right
Specifies the position of the diagonal matrix in the product. See *oneMKL defined datatypes* for more details.

m
Number of rows of matrices `A` and `C`. Must be at least zero.

n
Number of columns of matrices `A` and `C`. Must be at least zero.

a

Buffer holding the input matrices `A` with size `stridea` * `batch_size`. Must be of at least `lda` * `j` + `stridea` * (`batch_size` - 1) where j is n if column major layout is used or m if major layout is used.

lda
The leading dimension of the matrices `A`. It must be positive and at least `m` if column major layout is used or at least `n` if row major layout is used.

stridea
Stride between different `A` matrices.

x
Buffer holding the input matrices `X` with size `stridex` * `batch_size`. Must be of size at least (1 + (`len` - 1)*abs(`incx`)) + `stridex` * (`batch_size` - 1) where `len` is `n` if the diagonal matrix is on the right of the product or `m` otherwise.

incx
Stride between two consecutive elements of the `x` vectors.

stridex
Stride between different `X` vectors, must be at least 0.

c
Buffer holding input/output matrices `C` with size `stridec` * `batch_size`.

ldc
The leading dimension of the matrices `C`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if column major layout is used to store matrices.

stridec
Stride between different `C` matrices. Must be at least `ldc` * `n` if column major layout is used or `ldc` * `m` if row major layout is used.

batch_size
Specifies the number of diagonal matrix-matrix product operations to perform.

Output Parameters

c
Output overwritten by `batch_size` diagonal matrix-matrix product operations.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

dgmm_batch (USM Version)

Description

The USM version of `dgmm_batch` supports the group API and strided API.

The group API operation is defined as:

```
idx = 0
for i = 0 ... group_count - 1
    for j = 0 ... group_size - 1
        a and c are matrices of size mxn at position idx in a_array and c_array
        x is a vector of size m or n depending on left_right, at position idx in x_array
        if (left_right == oneapi::mkl::side::left)
            c := diag(x) * a
        else
            c := a * diag(x)
        idx := idx + 1
    end for
end for
```

The strided API operation is defined as

```
for i = 0 ... batch_size - 1
    A and C are matrices at offset i * stridea in a, i * stridec in c.
    X is a vector at offset i * stridex in x
    C := diag(X) * A or  C = A * diag(X)
end for
```

where:

`A` is a matrix,

`X` is a diagonal matrix stored as a vector

The `a` and `x` buffers contain all the input matrices. The stride between matrices is given by the stride parameter. The total number of matrices in `a` and `x` buffers is given by the `batch_size` parameter.

For group API, `a` and `x` arrays contain the pointers for all the input matrices. The total number of matrices in `a` and `x` are given by:

$$total\_batch\_count = \sum_{i=0}^{group\_count-1} group\_size[i]$$

For strided API, `a` and `x` arrays contain all the input matrices. The total number of matrices in `a` and `x` are given by the `batch_size` parameter.

Group API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event dgmm_batch(sycl::queue &queue,
                           const oneapi::mkl::side *left_right,
                           const std::int64_t *m,
                           const std::int64_t *n,
                           const T **a,
                           const std::int64_t *lda,
                           const T **x,
                           const std::int64_t *incx,
                           T **c,
                           const std::int64_t *ldc,
                           std::int64_t group_count,
                           const std::int64_t *group_size,
                           const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event dgmm_batch(sycl::queue &queue,
                           const oneapi::mkl::side *left_right,
                           const std::int64_t *m,
                           const std::int64_t *n,
                           const T **a,
                           const std::int64_t *lda,
                           const T **x,
                           const std::int64_t *incx,
                           T **c,
                           const std::int64_t *ldc,
                           std::int64_t group_count,
                           const std::int64_t *group_size,
                           const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

left_right
Specifies the position of the diagonal matrix in the product. See *oneMKL defined datatypes* for more details.

m
Array of `group_count` integers. `m[i]` specifies the number of rows of `A` for every matrix in group `i`. All entries must be at least zero.

n
Array of `group_count` integers. `n[i]` specifies the number of columns of `A` for every matrix in group `i`. All entries must be at least zero.

a
Array of pointers to input matrices `A` with size `total_batch_count`. Must be of size at least `lda[i]` * `n[i]` if column major layout is used or at least `lda[i]` * `m[i]` if row major layout is used. See *Matrix Storage* for more details.

lda
Array of `group_count` integers. `lda[i]` specifies the leading dimension of `A` for every matrix in group `i`. All entries must be positive and at least `m[i]` if column major layout is used or at least `n[i]` if row major layout is used.

x
Array of pointers to input vectors `X` with size `total_batch_count`. Must be of size at least (1 + `len[i]` – 1)*abs(`incx[i]`)) where `len[i]` is `n[i]` if the diagonal matrix is on the right of the product or `m[i]` otherwise. See *Matrix Storage* for more details.

incx
Array of `group_count` integers. `incx[i]` specifies the stride of `x` for every vector in group `i`. All entries must be positive.

c
Array of pointers to input/output matrices `C` with size `total_batch_count`. Must be of size at least `ldc[i]` * `n[i]` if column major layout is used or at least `ldc[i]` * `m[i]` if row major layout is used. See *Matrix Storage* for more details.

ldc
Array of `group_count` integers. `ldc[i]` specifies the leading dimension of `C` for every matrix in group `i`. All entries must be positive and `ldc[i]` must be at least `m[i]` if column major layout is used to store matrices or at least `n[i]` if row major layout is used to store matrices.

group_count
Specifies the number of groups. Must be at least 0.

group_size
Array of `group_count` integers. `group_size[i]` specifies the number of diagonal matrix-matrix product operations in group `i`. All entries must be at least 0.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

c
Output overwritten by `batch_size` diagonal matrix-matrix product operations.

Return Values

Output event to wait on to ensure computation is complete.

Strided API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event dgmm_batch(sycl::queue &queue,
                           oneapi::mkl::side left_right,
                           std::int64_t m,
                           std::int64_t n,
                           const T *a,
                           std::int64_t lda,
                           std::int64_t stridea,
                           const T *b,
                           std::int64_t incx,
                           std::int64_t stridex,
                           T *c,
                           std::int64_t ldc,
                           std::int64_t stridec,
                           std::int64_t batch_size,
                           const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event dgmm_batch(sycl::queue &queue,
                           oneapi::mkl::side left_right,
                           std::int64_t m,
                           std::int64_t n,
                           const T *a,
                           std::int64_t lda,
                           std::int64_t stridea,
                           const T *b,
                           std::int64_t incx,
                           std::int64_t stridex,
                           T *c,
                           std::int64_t ldc,
                           std::int64_t stridec,
                           std::int64_t batch_size,
                           const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

left_right
Specifies the position of the diagonal matrix in the product. See *oneMKL defined datatypes* for more details.

m
Number of rows of `A`. Must be at least zero.

n
Number of columns of `A`. Must be at least zero.

a
Pointer to input matrices `A` with size `stridea` * `batch_size`. Must be of size at least `lda` * `k` + `stridea` * (`batch_size` - 1) where `k` is `n` if column major layout is used or `m` if row major layout is used.

lda
The leading dimension of the matrices `A`. It must be positive and at least `m`. Must be positive and at least `m` if column major layout is used or at least `n` if row major layout is used.

stridea
Stride between different `A` matrices.

x
Pointer to input matrices `X` with size `stridex` * `batch_size`. Must be of size at least (1 + (`len` - 1)*abs(`incx`)) + `stridex` * (`batch_size` - 1) where `len` is `n` if the diagonal matrix is on the right of the product or `m` otherwise.

incx
Stride between two consecutive elements of the `x` vector.

stridex
Stride between different `X` vectors, must be at least 0.

c
Pointer to input/output matrices `C` with size `stridec` * `batch_size`.

ldc
The leading dimension of the matrices `C`. It must be positive and at least `ldc` * `m` if column major layout is used to store matrices or at least `ldc` * `n` if column major layout is used to store matrices.

stridec
Stride between different `C` matrices. Must be at least `ldc` * `n` if column major layout is used or `ldc` * `m` if row major layout is used.

batch_size
Specifies the number of diagonal matrix-matrix product operations to perform.

Output Parameters

c
Output overwritten by `batch_size` diagonal matrix-matrix product operations.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS-like Extensions*

gemm_batch

Computes a group of `gemm` operations.

Description

The `gemm_batch` routines are batched versions of *gemm*, performing multiple `gemm` operations in a single call. Each `gemm` operation perform a matrix-matrix product with general matrices.

`gemm_batch` supports the following precisions.

Ta (A matrix)	Tb (B matrix)	Tc (C matrix)	Ts (alpha/beta)
`std::int8_t`	`std::int8_t`	`std::int32_t`	`float`
`std::int8_t`	`std::int8_t`	`float`	`float`
`half`	`half`	`float`	`float`
`half`	`half`	`half`	`half`
`bfloat16`	`bfloat16`	`float`	`float`
`bfloat16`	`bfloat16`	`bfloat16`	`float`
`float`	`float`	`float`	`float`
`double`	`double`	`double`	`double`
`std::complex<float>`	`std::complex<float>`	`std::complex<float>`	`std::complex<float>`
`std::complex<double`	`std::complex<double`	`std::complex<double`	`std::complex<double>`

gemm_batch (Buffer Version)

Description

The buffer version of `gemm_batch` supports only the strided API.

The strided API operation is defined as:

```
for i = 0 ... batch_size - 1
    A, B and C are matrices at offset i * stridea, i * strideb, i * stridec in a, b and␣
↪c.
    C := alpha * op(A) * op(B) + beta * C
end for
```

where:

op(X) is one of op(X) = X, or op(X) = X^T, or op(X) = X^H,

`alpha` and `beta` are scalars,

`A`, `B`, and `C` are matrices,

op(`A`) is `m` x `k`, op(`B`) is `k` x `n`, and `C` is `m` x `n`.

The `a`, `b` and `c` buffers contain all the input matrices. The stride between matrices is given by the stride parameter. The total number of matrices in `a`, `b` and `c` buffers is given by the `batch_size` parameter.

Strided API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void gemm_batch(sycl::queue &queue,
                    oneapi::mkl::transpose transa,
                    oneapi::mkl::transpose transb,
                    std::int64_t m,
                    std::int64_t n,
                    std::int64_t k,
                    Ts alpha,
                    sycl::buffer<Ta,1> &a,
                    std::int64_t lda,
                    std::int64_t stridea,
                    sycl::buffer<Tb,1> &b,
                    std::int64_t ldb,
                    std::int64_t strideb,
                    Ts beta,
                    sycl::buffer<Tc,1> &c,
                    std::int64_t ldc,
                    std::int64_t stridec,
                    std::int64_t batch_size)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void gemm_batch(sycl::queue &queue,
                    oneapi::mkl::transpose transa,
```

(continues on next page)

(continued from previous page)

```
                    oneapi::mkl::transpose transb,
                    std::int64_t m,
                    std::int64_t n,
                    std::int64_t k,
                    Ts alpha,
                    sycl::buffer<Ta,1> &a,
                    std::int64_t lda,
                    std::int64_t stridea,
                    sycl::buffer<Tb,1> &b,
                    std::int64_t ldb,
                    std::int64_t strideb,
                    Ts beta,
                    sycl::buffer<Tc,1> &c,
                    std::int64_t ldc,
                    std::int64_t stridec,
                    std::int64_t batch_size)
}
```

Input Parameters

queue
The queue where the routine should be executed.

transa
Specifies op(`A`) the transposition operation applied to the matrices `A`. See *oneMKL defined datatypes* for more details.

transb
Specifies op(`B`) the transposition operation applied to the matrices `B`. See *oneMKL defined datatypes* for more details.

m
Number of rows of op(`A`) and `C`. Must be at least zero.

n
Number of columns of op(`B`) and `C`. Must be at least zero.

k
Number of columns of op(`A`) and rows of op(`B`). Must be at least zero.

alpha
Scaling factor for the matrix-matrix products.

a
Buffer holding the input matrices `A` with size `stridea` * `batch_size`.

lda
The leading dimension of the matrices `A`. It must be positive.

	`A` not transposed	`A` transposed
Column major	`lda` must be at least `m`.	`lda` must be at least `k`.
Row major	`lda` must be at least `k`.	`lda` must be at least `m`.

stridea
Stride between different `A` matrices.

b
Buffer holding the input matrices `B` with size `strideb` * `batch_size`.

ldb
The leading dimension of the matrices``B``. It must be positive.

	`B` not transposed	`B` transposed
Column major	`ldb` must be at least `k`.	`ldb` must be at least `n`.
Row major	`ldb` must be at least `n`.	`ldb` must be at least `k`.

strideb
Stride between different `B` matrices.

beta
Scaling factor for the matrices `C`.

c
Buffer holding input/output matrices `C` with size `stridec` * `batch_size`.

ldc
The leading dimension of the matrices `C`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if row major layout is used to store matrices.

stridec
Stride between different `C` matrices. Must be at least `ldc` * `n`.

batch_size
Specifies the number of matrix multiply operations to perform.

Output Parameters

c
Output buffer, overwritten by `batch_size` matrix multiply operations of the form `alpha` * op(`A`)*op(`B`) + `beta` * `C`.

Notes

If `beta` = 0, matrix `C` does not need to be initialized before calling `gemm_batch`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

gemm_batch (USM Version)

Description

The USM version of `gemm_batch` supports the group API and the strided API. The group API supports pointer and span inputs.

The group API operation is defined as:

```
idx = 0
for i = 0 ... group_count - 1
    for j = 0 ... group_size - 1
        A, B, and C are matrices in a[idx], b[idx] and c[idx]
        C := alpha[i] * op(A) * op(B) + beta[i] * C
        idx = idx + 1
    end for
end for
```

The advantage of using span instead of pointer is that the sizes of the array can vary and the size of the span can be queried at runtime. For each GEMM parameter, except the output matrices, the span can be of size 1, the number of groups or the total batch size. For the output matrices, to ensure all computation are independent, the size of the span must be the total batch size.

Depending on the size of the spans, each parameter for the GEMM computation is used as follows:

- If the span has size 1, the parameter is reused for all GEMM computation.
- If the span has size group_count, the parameter is reused for all GEMM within a group, but each group will have a different value for this parameter. This is like the gemm_batch group API with pointers.
- If the span has size equal to the total batch size, each GEMM computation will use a different value for this parameter.

The strided API operation is defined as

```
for i = 0 ... batch_size - 1
    A, B and C are matrices at offset i * stridea, i * strideb, i * stridec in a, b and␣
↪c.
    C := alpha * op(A) * op(B) + beta * C
end for
```

where:

op(X) is one of op(X) = X, or op(X) = X^T, or op(X) = X^H,

`alpha` and `beta` are scalars,

`A`, `B`, and `C` are matrices,

op(`A`) is `m` x `k`, op(`B`) is `k` x `n`, and `C` is `m` x `n`.

For group API, `a`, `b` and `c` arrays contain the pointers for all the input matrices. The total number of matrices in `a`, `b` and `c` are given by:

$$total\_batch\_count = \sum_{i=0}^{group\_count-1} group\_size[i]$$

For strided API, `a`, `b`, `c` arrays contain all the input matrices. The total number of matrices in `a`, `b` and `c` are given by the `batch_size` parameter.

Group API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event gemm_batch(sycl::queue &queue,
                           const oneapi::mkl::transpose *transa,
                           const oneapi::mkl::transpose *transb,
                           const std::int64_t *m,
                           const std::int64_t *n,
                           const std::int64_t *k,
                           const Ts *alpha,
                           const Ta **a,
                           const std::int64_t *lda,
                           const Tb **b,
                           const std::int64_t *ldb,
                           const Ts *beta,
                           Tc **c,
                           const std::int64_t *ldc,
                           std::int64_t group_count,
                           const std::int64_t *group_size,
                           const std::vector<sycl::event> &dependencies = {})

    sycl::event gemm_batch(sycl::queue &queue,
                           const sycl::span<oneapi::mkl::transpose> &transa,
                           const sycl::span<oneapi::mkl::transpose> &transb,
                           const sycl::span<std::int64_t> &m,
                           const sycl::span<std::int64_t> &n,
                           const sycl::span<std::int64_t> &k,
                           const sycl::span<Ts> &alpha,
                           const sycl::span<const Ta*> &a,
                           const sycl::span<std::int64_t> &lda,
                           const sycl::span<const Tb*> &b,
                           const sycl::span<std::int64_t> &ldb,
                           const sycl::span<Ts> &beta,
                           sycl::span<Tc*> &c,
                           const sycl::span<std::int64_t> &ldc,
                           size_t group_count,
                           const sycl::span<size_t> &group_sizes,
                           const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event gemm_batch(sycl::queue &queue,
                           const oneapi::mkl::transpose *transa,
                           const oneapi::mkl::transpose *transb,
                           const std::int64_t *m,
                           const std::int64_t *n,
                           const std::int64_t *k,
                           const Ts *alpha,
                           const Ta **a,
                           const std::int64_t *lda,
                           const Tb **b,
                           const std::int64_t *ldb,
                           const Ts *beta,
```

(continues on next page)

(continued from previous page)

```
                            Tc **c,
                            const std::int64_t *ldc,
                            std::int64_t group_count,
                            const std::int64_t *group_size,
                            const std::vector<sycl::event> &dependencies = {})

    sycl::event gemm_batch(sycl::queue &queue,
                            const sycl::span<oneapi::mkl::transpose> &transa,
                            const sycl::span<oneapi::mkl::transpose> &transb,
                            const sycl::span<std::int64_t> &m,
                            const sycl::span<std::int64_t> &n,
                            const sycl::span<std::int64_t> &k,
                            const sycl::span<Ts> &alpha,
                            const sycl::span<const Ta*> &a,
                            const sycl::span<std::int64_t> &lda,
                            const sycl::span<const Tb*> &b,
                            const sycl::span<std::int64_t> &ldb,
                            const sycl::span<Ts> &beta,
                            sycl::span<Tc*> &c,
                            const sycl::span<std::int64_t> &ldc,
                            size_t group_count,
                            const sycl::span<size_t> &group_sizes,
                            const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
: The queue where the routine should be executed.

transa
: Array or span of `group_count` `oneapi::mkl::transpose` values. `transa[i]` specifies the form of op(`A`) used in the matrix multiplication in group `i`. See *oneMKL defined datatypes* for more details.

transb
: Array or span of `group_count` `oneapi::mkl::transpose` values. `transb[i]` specifies the form of op(`B`) used in the matrix multiplication in group `i`. See *oneMKL defined datatypes* for more details.

m
: Array or span of `group_count` integers. `m[i]` specifies the number of rows of op(`A`) and `C` for every matrix in group `i`. All entries must be at least zero.

n
: Array or span of `group_count` integers. `n[i]` specifies the number of columns of op(`B`) and `C` for every matrix in group `i`. All entries must be at least zero.

k
: Array or span of `group_count` integers. `k[i]` specifies the number of columns of op(`A`) and rows of op(`B`) for every matrix in group `i`. All entries must be at least zero.

alpha
: Array or span of `group_count` scalar elements. `alpha[i]` specifies the scaling factor for every matrix-matrix product in group `i`.

a

Array of pointers or span of input matrices `A` with size `total_batch_count`.

See *Matrix Storage* for more details.

lda

Array or span of `group_count` integers. `lda[i]` specifies the leading dimension of `A` for every matrix in group `i`. All entries must be positive.

	`A` not transposed	`A` transposed
Column major	`lda[i]` must be at least `m[i]`.	`lda[i]` must be at least `k[i]`.
Row major	`lda[i]` must be at least `k[i]`.	`lda[i]` must be at least `m[i]`.

b

Array of pointers or span of input matrices `B` with size `total_batch_count`.

See *Matrix Storage* for more details.

ldb

Array or span of `group_count` integers. `ldb[i]` specifies the leading dimension of `B` for every matrix in group `i`. All entries must be positive.

	`B` not transposed	`B` transposed
Column major	`ldb[i]` must be at least `k[i]`.	`ldb[i]` must be at least `n[i]`.
Row major	`ldb[i]` must be at least `n[i]`.	`ldb[i]` must be at least `k[i]`.

beta

Array or span of `group_count` scalar elements. `beta[i]` specifies the scaling factor for matrix `C` for every matrix in group `i`.

c

Array of pointers or span of input/output matrices `C` with size `total_batch_count`.

See *Matrix Storage* for more details.

ldc

Array or span of `group_count` integers. `ldc[i]` specifies the leading dimension of `C` for every matrix in group `i`. All entries must be positive and `ldc[i]` must be at least `m[i]` if column major layout is used to store matrices or at least `n[i]` if row major layout is used to store matrices.

group_count

Specifies the number of groups. Must be at least 0.

group_size

Array or span of `group_count` integers. `group_size[i]` specifies the number of matrix multiply products in group `i`. All entries must be at least 0.

dependencies

List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

c
Overwritten by the `m[i]`-by-`n[i]` matrix calculated by (`alpha[i]` * op(`A`)*op(`B`) + `beta[i]` * `C`) for group `i`.

Notes

If `beta` = 0, matrix `C` does not need to be initialized before calling `gemm_batch`.

Return Values

Output event to wait on to ensure computation is complete.

Output Parameters

c
Overwritten by the `m[i]`-by-`n[i]` matrix calculated by (`alpha[i]` * op(`A`)*op(`B`) + `beta[i]` * `C`) for group `i`.

Notes

If `beta` = 0, matrix `C` does not need to be initialized before calling `gemm_batch`.

Return Values

Output event to wait on to ensure computation is complete.

Strided API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event gemm_batch(sycl::queue &queue,
                           oneapi::mkl::transpose transa,
                           oneapi::mkl::transpose transb,
                           std::int64_t m,
                           std::int64_t n,
                           std::int64_t k,
                           value_or_pointer<Ts> alpha,
                           const Ta *a,
                           std::int64_t lda,
                           std::int64_t stridea,
                           const Tb *b,
                           std::int64_t ldb,
                           std::int64_t strideb,
                           value_or_pointer<Ts> beta,
                           Tc *c,
                           std::int64_t ldc,
                           std::int64_t stridec,
                           std::int64_t batch_size,
```

(continues on next page)

(continued from previous page)

```
                            const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event gemm_batch(sycl::queue &queue,
                            oneapi::mkl::transpose transa,
                            oneapi::mkl::transpose transb,
                            std::int64_t m,
                            std::int64_t n,
                            std::int64_t k,
                            value_or_pointer<Ts> alpha,
                            const Ta *a,
                            std::int64_t lda,
                            std::int64_t stridea,
                            const Tb *b,
                            std::int64_t ldb,
                            std::int64_t strideb,
                            value_or_pointer<Ts> beta,
                            Tc *c,
                            std::int64_t ldc,
                            std::int64_t stridec,
                            std::int64_t batch_size,
                            const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

transa
Specifies op(`A`) the transposition operation applied to the matrices `A`. See *oneMKL defined datatypes* for more details.

transb
Specifies op(`B`) the transposition operation applied to the matrices `B`. See *oneMKL defined datatypes* for more details.

m
Number of rows of op(`A`) and `C`. Must be at least zero.

n
Number of columns of op(`B`) and `C`. Must be at least zero.

k
Number of columns of op(`A`) and rows of op(`B`). Must be at least zero.

alpha
Scaling factor for the matrix-matrix products. See *Scalar Arguments in BLAS* for more details.

a
Pointer to input matrices `A` with size `stridea` * `batch_size`.

lda
The leading dimension of the matrices `A`. It must be positive.

	`A` not transposed	`A` transposed
Column major	`lda` must be at least `m`.	`lda` must be at least `k`.
Row major	`lda` must be at least `k`.	`lda` must be at least `m`.

stridea
Stride between different `A` matrices.

b
Pointer to input matrices `B` with size `strideb` * `batch_size`.

ldb
The leading dimension of the matrices``B``. It must be positive.

	`B` not transposed	`B` transposed
Column major	`ldb` must be at least `k`.	`ldb` must be at least `n`.
Row major	`ldb` must be at least `n`.	`ldb` must be at least `k`.

strideb
Stride between different `B` matrices.

beta
Scaling factor for the matrices `C`. See *Scalar Arguments in BLAS* for more details.

c
Pointer to input/output matrices `C` with size `stridec` * `batch_size`.

ldc
The leading dimension of the matrices `C`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if row major layout is used to store matrices.

stridec
Stride between different `C` matrices.

batch_size
Specifies the number of matrix multiply operations to perform.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

c
Output matrices, overwritten by `batch_size` matrix multiply operations of the form `alpha` * op(`A`)*op(`B`) + `beta` * `C`.

Notes

If `beta` = 0, matrix `C` does not need to be initialized before calling `gemm_batch`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS-like Extensions*

gemv_batch

Computes a group of `gemv` operations.

Description

The `gemv_batch` routines are batched versions of *gemv*, performing multiple `gemv` operations in a single call. Each `gemv` operations perform a scalar-matrix-vector product and add the result to a scalar-vector product.

`gemv_batch` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

gemv_batch (Buffer Version)

Description

The buffer version of `gemv_batch` supports only the strided API.

The strided API operation is defined as:

```
for i = 0 ... batch_size - 1
    A is a matrix at offset i * stridea in a.
    X and Y are matrices at offset i * stridex, i * stridey, in x and y.
    Y := alpha * op(A) * X + beta * Y
end for
```

where:

op(A) is one of op(A) = A, or op(A) = A^T, or op(A) = A^H,

`alpha` and `beta` are scalars,

`A` is a matrix and `X` and `Y` are vectors,

The `x` and `y` buffers contain all the input matrices. The stride between vectors is given by the stride parameter. The total number of vectors in `x` and `y` buffers is given by the `batch_size` parameter.

Strided API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void gemv_batch(sycl::queue &queue,
                    oneapi::mkl::transpose trans,
                    std::int64_t m,
                    std::int64_t n,
                    T alpha,
                    sycl::buffer<T,1> &a,
                    std::int64_t lda,
                    std::int64_t stridea,
                    sycl::buffer<T,1> &x,
                    std::int64_t incx,
                    std::int64_t stridex,
                    T beta,
                    sycl::buffer<T,1> &y,
                    std::int64_t incy,
                    std::int64_t stridey,
                    std::int64_t batch_size)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void gemv_batch(sycl::queue &queue,
                    oneapi::mkl::transpose trans,
                    std::int64_t m,
                    std::int64_t n,
                    T alpha,
                    sycl::buffer<T,1> &a,
                    std::int64_t lda,
                    std::int64_t stridea,
                    sycl::buffer<T,1> &x,
                    std::int64_t incx,
                    std::int64_t stridex,
                    T beta,
                    sycl::buffer<T,1> &y,
```

(continues on next page)

(continued from previous page)

```
                    std::int64_t incy,
                    std::int64_t stridey,
                    std::int64_t batch_size)
}
```

Input Parameters

queue
The queue where the routine should be executed.

trans
Specifies op(`A`) the transposition operation applied to the matrices `A`. See *oneMKL defined datatypes* for more details.

m
Number of rows of `A`. Must be at least zero.

n
Number of columns of `A`. Must be at least zero.

alpha
Scaling factor for the matrix-vector products.

a
Buffer holding the input matrices `A` with size `stridea` * `batch_size`.

lda
The leading dimension of the matrices `A`. It must be positive and at least `m` if column major layout is used or at least `n` if row major layout is used.

stridea
Stride between different `A` matrices. Must be at least zero.

x
Buffer holding the input vectors `X` with size `stridex` * `batch_size`.

incx
The stride of the vector `X`. Must not be zero.

stridex
Stride between different consecutive `X` vectors, must be at least 0.

beta
Scaling factor for the vector `Y`.

y
Buffer holding input/output vectors `Y` with size `stridey` * `batch_size`.

incy
Stride between two consecutive elements of the `Y` vectors. Must not be zero.

stridey
Stride between two consecutive `Y` vectors. Must be at least (1 + (`m` - 1)*abs(`incy`)) if layout is column major or (1 + (`n` - 1)*abs(`incy`)) if row major layout is used.

batch_size
Specifies the number of matrix-vector operations to perform.

Output Parameters

y

Output overwritten by `batch_size` matrix-vector product operations of the form `alpha` * op(`A`) * `X` + `beta` * `Y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

gemv_batch (USM Version)

Description

The USM version of `gemv_batch` supports the group API and strided API.

The group API operation is defined as:

```
idx = 0
for i = 0 ... group_count - 1
    for j = 0 ... group_size - 1
        A is an m x n matrix in a[idx]
        X and Y are vectors in x[idx] and y[idx]
        Y := alpha[i] * op(A) * X + beta[i] * Y
        idx = idx + 1
    end for
end for
```

The strided API operation is defined as

```
for i = 0 ... batch_size - 1
    A is a matrix at offset i * stridea in a.
    X and Y are vectors at offset i * stridex, i * stridey in x and y.
    Y := alpha * op(A) * X + beta * Y
end for
```

where:

op(A) is one of op(A) = A, or op(A) = A^T, or op(A) = A^H,

`alpha` and `beta` are scalars,

`A` is a matrix and `X` and `Y` are vectors,

For group API, `x` and `y` arrays contain the pointers for all the input vectors. `A` array contains the pointers to all input matrices. The total number of vectors in `x` and `y` and matrices in `A` are given by:

$$total\_batch\_count = \sum_{i=0}^{group\_count-1} group\_size[i]$$

For strided API, `x` and `y` arrays contain all the input vectors. `A` array contains the pointers to all input matrices. The total number of vectors in `x` and `y` and matrices in `A` are given by the `batch_size` parameter.

Group API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event gemv_batch(sycl::queue &queue,
                           const oneapi::mkl::transpose *trans,
                           const std::int64_t *m,
                           const std::int64_t *n,
                           const T *alpha,
                           const T **a,
                           const std::int64_t *lda,
                           const T **x,
                           const std::int64_t *incx,
                           const T *beta,
                           T **y,
                           const std::int64_t *incy,
                           std::int64_t group_count,
                           const std::int64_t *group_size,
                           const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event gemv_batch(sycl::queue &queue,
                           const oneapi::mkl::transpose *trans,
                           const std::int64_t *m,
                           const std::int64_t *n,
                           const T *alpha,
                           const T **a,
                           const std::int64_t *lda,
                           const T **x,
                           const std::int64_t *incx,
                           const T *beta,
                           T **y,
                           const std::int64_t *incy,
                           std::int64_t group_count,
                           const std::int64_t *group_size,
                           const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

trans
Array of `group_count` `oneapi::mkl::transpose` values. `trans[i]` specifies the form of op(`A`) used in the matrix-vector product in group `i`. See *oneMKL defined datatypes* for more details.

m
Array of `group_count` integers. `m[i]` specifies the number of rows of `A` for every matrix in group `i`. All entries must be at least zero.

n
Array of `group_count` integers. `n[i]` specifies the number of columns of `A` for every matrix in group `i`. All entries must be at least zero.

alpha
Array of `group_count` scalar elements. `alpha[i]` specifies the scaling factor for every matrix-vector product in group `i`.

a
Array of pointers to input matrices `A` with size `total_batch_count`.

See *Matrix Storage* for more details.

lda
Array of `group_count` integers. `lda[i]` specifies the leading dimension of `A` for every matrix in group `i`. All entries must be positive and at least `m` if column major layout is used or at least `n` if row major layout is used.

x
Array of pointers to input vectors `X` with size `total_batch_count`.

See *Matrix Storage* for more details.

incx
Array of `group_count` integers. `incx[i]` specifies the stride of `X` for every vector in group `i`. Must not be zero.

beta
Array of `group_count` scalar elements. `beta[i]` specifies the scaling factor for vector `Y` for every vector in group `i`.

y
Array of pointers to input/output vectors `Y` with size `total_batch_count`.

See *Matrix Storage* for more details.

incy
Array of `group_count` integers. `incy[i]` specifies the leading dimension of `Y` for every vector in group `i`. Must not be zero.

group_count
Specifies the number of groups. Must be at least 0.

group_size
Array of `group_count` integers. `group_size[i]` specifies the number of matrix-vector products in group `i`. All entries must be at least 0.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

y
Overwritten by vector calculated by (`alpha[i]` * op(`A`) * `X` + `beta[i]` * `Y`) for group `i`.

Return Values

Output event to wait on to ensure computation is complete.

Strided API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event gemv_batch(sycl::queue &queue,
                           oneapi::mkl::transpose trans,
                           std::int64_t m,
                           std::int64_t n,
                           value_or_pointer<T> alpha,
                           const T *a,
                           std::int64_t lda,
                           std::int64_t stridea,
                           const T *x,
                           std::int64_t incx,
                           std::int64_t stridex,
                           value_or_pointer<T> beta,
                           T *y,
                           std::int64_t incy,
                           std::int64_t stridey,
                           std::int64_t batch_size,
                           const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event gemv_batch(sycl::queue &queue,
                           oneapi::mkl::transpose trans,
                           std::int64_t m,
                           std::int64_t n,
                           value_or_pointer<T> alpha,
                           const T *a,
                           std::int64_t lda,
                           std::int64_t stridea,
                           const T *x,
                           std::int64_t incx,
                           std::int64_t stridex,
                           value_or_pointer<T> beta,
                           T *y,
                           std::int64_t incy,
                           std::int64_t stridey,
                           std::int64_t batch_size,
                           const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

trans
Specifies op(`A`) the transposition operation applied to the matrices `A`. See *oneMKL defined datatypes* for more details.

m
Number of rows of `A`. Must be at least zero.

n
Number of columns of `A`. Must be at least zero.

alpha
Scaling factor for the matrix-vector products. See *Scalar Arguments in BLAS* for more details.

a
Pointer to the input matrices `A` with size `stridea` * `batch_size`.

lda
The leading dimension of the matrices `A`. It must be positive and at least `m` if column major layout is used or at least `n` if row major layout is used.

stridea
Stride between different `A` matrices. Must be at least zero.

x
Pointer to the input vectors `X` with size `stridex` * `batch_size`.

incx
Stride of the vector `X`. Must not be zero.

stridex
Stride between different consecutive `X` vectors, must be at least 0.

beta
Scaling factor for the vector `Y`. See *Scalar Arguments in BLAS* for more details.

y
Pointer to the input/output vectors `Y` with size `stridey` * `batch_size`.

incy
Stride between two consecutive elements of the `y` vectors. Must not be zero.

stridey
Stride between two consecutive `Y` vectors. Must be at least (1 + (`m` - 1)*abs(`incy`)) if layout is column major or (1 + (`n` - 1)*abs(`incy`)) if row major layout is used.

batch_size
Specifies the number of matrix-vector operations to perform.

Output Parameters

y

Output overwritten by `batch_size` matrix-vector product operations of the form `alpha` * op(`A`) * `X` + `beta` * `Y`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS-like Extensions*

syrk_batch

Computes a group of `syrk` operations.

Description

The `syrk_batch` routines are batched versions of *syrk*, performing multiple `syrk` operations in a single call. Each `syrk` operation perform a rank-k update with general matrices.

`syrk_batch` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

syrk_batch (Buffer Version)

Description

The buffer version of `syrk_batch` supports only the strided API.

The strided API operation is defined as:

```
for i = 0 ... batch_size - 1
    A and C are matrices at offset i * stridea, i * stridec in a and c.
    C := alpha * op(A) * op(A)^T + beta * C
end for
```

where:

op(X) is one of op(X) = X, or op(X) = X^T, or op(X) = X^H,

`alpha` and `beta` are scalars,

`A` and `C` are matrices,

op(`A`) is `n` x `k` and `C` is `n` x `n`.

The `a` and `c` buffers contain all the input matrices. The stride between matrices is given by the stride parameter. The total number of matrices in `a` and `c` buffers is given by the `batch_size` parameter.

Strided API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void syrk_batch(sycl::queue &queue,
                    oneapi::mkl::uplo upper_lower,
                    oneapi::mkl::transpose trans,
                    std::int64_t n,
                    std::int64_t k,
                    T alpha,
                    sycl::buffer<T,1> &a,
                    std::int64_t lda,
                    std::int64_t stridea,
                    T beta,
                    sycl::buffer<T,1> &c,
                    std::int64_t ldc,
                    std::int64_t stridec,
                    std::int64_t batch_size)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void syrk_batch(sycl::queue &queue,
                    oneapi::mkl::uplo upper_lower,
                    oneapi::mkl::transpose trans,
                    std::int64_t n,
                    std::int64_t k,
                    T alpha,
                    sycl::buffer<T,1> &a,
```

(continues on next page)

(continued from previous page)

```
                    std::int64_t lda,
                    std::int64_t stridea,
                    T beta,
                    sycl::buffer<T,1> &c,
                    std::int64_t ldc,
                    std::int64_t stridec,
                    std::int64_t batch_size)
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether data in `C` is stored in its upper or lower triangle. For more details, see *oneMKL defined datatypes*.

trans
Specifies op(`A`) the transposition operation applied to the matrix `A`. Conjugation is never performed, even if trans = transpose::conjtrans. See *oneMKL defined datatypes* for more details.

n
Number of rows and columns of `C`. Must be at least zero.

k
Number of columns of op(`A`). Must be at least zero.

alpha
Scaling factor for the rank-k update.

a
Buffer holding the input matrices `A` with size `stridea` * `batch_size`.

lda
The leading dimension of the matrices `A`. It must be positive.

	`A` not transposed	`A` transposed
Column major	`lda` must be at least `n`.	`lda` must be at least `k`.
Row major	`lda` must be at least `k`.	`lda` must be at least `n`.

stridea
Stride between different `A` matrices.

beta
Scaling factor for the matrices `C`.

c
Buffer holding input/output matrices `C` with size `stridec` * `batch_size`.

ldc
The leading dimension of the matrices `C`. It must be positive and at least `n`.

stridec
Stride between different `C` matrices. Must be at least `ldc` * `n`.

batch_size
Specifies the number of rank-k update operations to perform.

Output Parameters

c
Output buffer, overwritten by `batch_size` rank-k update operations of the form `alpha` * op(`A`)*op(`A`)^T + `beta` * `C`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

syrk_batch (USM Version)

Description

The USM version of `syrk_batch` supports the group API and strided API.

The group API operation is defined as:

```
idx = 0
for i = 0 ... group_count - 1
    for j = 0 ... group_size - 1
        A, B, and C are matrices in a[idx] and c[idx]
        C := alpha[i] * op(A) * op(A)^T + beta[i] * C
        idx = idx + 1
    end for
end for
```

The strided API operation is defined as

```
for i = 0 ... batch_size - 1
    A, B and C are matrices at offset i * stridea, i * stridec in a and c.
    C := alpha * op(A) * op(A)^T + beta * C
end for
```

where:

op(X) is one of op(X) = X, or op(X) = X^T, or op(X) = X^H,

`alpha` and `beta` are scalars,

`A` and `C` are matrices,

op(`A`) is `n` x `k` and `C` is `n` x `n`.

For group API, a and c arrays contain the pointers for all the input matrices. The total number of matrices in a and c are given by:

$$total\_batch\_count = \sum_{i=0}^{group\_count-1} group\_size[i]$$

For strided API, a and c arrays contain all the input matrices. The total number of matrices in a and c are given by the batch_size parameter.

Group API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event syrk_batch(sycl::queue &queue,
                           const uplo *upper_lower,
                           const transpose *trans,
                           const std::int64_t *n,
                           const std::int64_t *k,
                           const T *alpha,
                           const T **a,
                           const std::int64_t *lda,
                           const T *beta,
                           T **c,
                           const std::int64_t *ldc,
                           std::int64_t group_count,
                           const std::int64_t *group_size,
                           const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event syrk_batch(sycl::queue &queue,
                           const uplo *upper_lower,
                           const transpose *trans,
                           const std::int64_t *n,
                           const std::int64_t *k,
                           const T *alpha,
                           const T **a,
                           const std::int64_t *lda,
                           const T *beta,
                           T **c,
                           const std::int64_t *ldc,
                           std::int64_t group_count,
                           const std::int64_t *group_size,
                           const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Array of `group_count` `oneapi::mkl::upper_lower` values. `upper_lower[i]` specifies whether data in `C` for every matrix in group `i` is in upper or lower triangle.

trans
Array of `group_count` `oneapi::mkl::transpose` values. `trans[i]` specifies the form of op(`A`) used in the rank-k update in group `i`. See *oneMKL defined datatypes* for more details.

n
Array of `group_count` integers. `n[i]` specifies the number of rows and columns of `C` for every matrix in group `i`. All entries must be at least zero.

k
Array of `group_count` integers. `k[i]` specifies the number of columns of op(`A`) for every matrix in group `i`. All entries must be at least zero.

alpha
Array of `group_count` scalar elements. `alpha[i]` specifies the scaling factor for every rank-k update in group `i`.

a
Array of pointers to input matrices `A` with size `total_batch_count`.

See *Matrix Storage* for more details.

lda
Array of `group_count` integers. `lda[i]` specifies the leading dimension of `A` for every matrix in group `i`. All entries must be positive.

	`A` not transposed	`A` transposed
Column major	`lda[i]` must be at least `n[i]`.	`lda[i]` must be at least `k[i]`.
Row major	`lda[i]` must be at least `k[i]`.	`lda[i]` must be at least `n[i]`.

beta
Array of `group_count` scalar elements. `beta[i]` specifies the scaling factor for matrix `C` for every matrix in group `i`.

c
Array of pointers to input/output matrices `C` with size `total_batch_count`.

See *Matrix Storage* for more details.

ldc
Array of `group_count` integers. `ldc[i]` specifies the leading dimension of `C` for every matrix in group `i`. All entries must be positive and `ldc[i]` must be at least `n[i]`.

group_count
Specifies the number of groups. Must be at least 0.

group_size
Array of `group_count` integers. `group_size[i]` specifies the number of rank-k update products in group `i`. All entries must be at least 0.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

c

Overwritten by the `n[i]`-by-`n[i]` matrix calculated by (`alpha[i]` * op(`A`)*op(`A`)^T + `beta[i]` * `C`) for group `i`.

Return Values

Output event to wait on to ensure computation is complete.

Strided API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event syrk_batch(sycl::queue &queue,
                           uplo upper_lower,
                           transpose trans,
                           std::int64_t n,
                           std::int64_t k,
                           value_or_pointer<T> alpha,
                           const T *a,
                           std::int64_t lda,
                           std::int64_t stride_a,
                           value_or_pointer<T> beta,
                           T *c,
                           std::int64_t ldc,
                           std::int64_t stride_c,
                           std::int64_t batch_size,
                           const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event syrk_batch(sycl::queue &queue,
                           uplo upper_lower,
                           transpose trans,
                           std::int64_t n,
                           std::int64_t k,
                           value_or_pointer<T> alpha,
                           const T *a,
                           std::int64_t lda,
                           std::int64_t stride_a,
                           value_or_pointer<T> beta,
                           T *c,
                           std::int64_t ldc,
                           std::int64_t stride_c,
                           std::int64_t batch_size,
                           const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether data in `C` is stored in its upper or lower triangle. For more details, see *oneMKL defined datatypes*.

trans
Specifies op(`A`) the transposition operation applied to the matrices `A`. Conjugation is never performed, even if trans = transpose::conjtrans. See *oneMKL defined datatypes* for more details.

n
Number of rows and columns of `C`. Must be at least zero.

k
Number of columns of op(`A`). Must be at least zero.

alpha
Scaling factor for the rank-k updates. See *Scalar Arguments in BLAS* for more details.

a
Pointer to input matrices `A` with size `stridea` * `batch_size`.

lda
The leading dimension of the matrices `A`. It must be positive.

	`A` not transposed	`A` transposed
Column major	`lda` must be at least `n`.	`lda` must be at least `k`.
Row major	`lda` must be at least `k`.	`lda` must be at least `n`.

stridea
Stride between different `A` matrices.

beta
Scaling factor for the matrices `C`. See *Scalar Arguments in BLAS* for more details.

c
Pointer to input/output matrices `C` with size `stridec` * `batch_size`.

ldc
The leading dimension of the matrices `C`. It must be positive and at least `n`.

stridec
Stride between different `C` matrices.

batch_size
Specifies the number of rank-k update operations to perform.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

c

Output matrices, overwritten by `batch_size` rank-k update operations of the form `alpha` * op(`A`)*op(`A`)^T + `beta` * `C`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS-like Extensions*

trsm_batch

Computes a group of `trsm` operations.

Description

The `trsm_batch` routines are batched versions of *trsm*, performing multiple `trsm` operations in a single call. Each `trsm` solves an equation of the form op(A) * X = alpha * B or X * op(A) = alpha * B.

`trsm_batch` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

trsm_batch (Buffer Version)

Description

The buffer version of `trsm_batch` supports only the strided API.

The strided API operation is defined as:

```
for i = 0 ... batch_size - 1
    A and B are matrices at offset i * stridea and i * strideb in a and b.
    if (left_right == oneapi::mkl::side::left) then
        compute X such that op(A) * X = alpha * B
    else
        compute X such that X * op(A) = alpha * B
    end if
    B := X
end for
```

where:

op(`A`) is one of op(`A`) = `A`, or op(A) = A^T, or op(`A`) = A^H,

`alpha` is a scalar,

`A` is a triangular matrix,

`B` and `X` are `m` x `n` general matrices,

`A` is either `m` x `m` or `n` x `n`,depending on whether it multiplies `X` on the left or right. On return, the matrix `B` is overwritten by the solution matrix `X`.

The `a` and `b` buffers contain all the input matrices. The stride between matrices is given by the stride parameter. The total number of matrices in `a` and `b` buffers are given by the `batch_size` parameter.

Strided API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void trsm_batch(sycl::queue &queue,
                    oneapi::mkl::side left_right,
                    oneapi::mkl::uplo upper_lower,
                    oneapi::mkl::transpose trans,
                    oneapi::mkl::diag unit_diag,
                    std::int64_t m,
                    std::int64_t n,
                    T alpha,
                    sycl::buffer<T,1> &a,
                    std::int64_t lda,
                    std::int64_t stridea,
                    sycl::buffer<T,1> &b,
                    std::int64_t ldb,
                    std::int64_t strideb,
                    std::int64_t batch_size)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void trsm_batch(sycl::queue &queue,
                    oneapi::mkl::side left_right,
                    oneapi::mkl::uplo upper_lower,
                    oneapi::mkl::transpose trans,
                    oneapi::mkl::diag unit_diag,
                    std::int64_t m,
                    std::int64_t n,
                    T alpha,
                    sycl::buffer<T,1> &a,
                    std::int64_t lda,
                    std::int64_t stridea,
                    sycl::buffer<T,1> &b,
                    std::int64_t ldb,
                    std::int64_t strideb,
                    std::int64_t batch_size)
}
```

Input Parameters

queue
The queue where the routine should be executed.

left_right
Specifies whether the matrices `A` multiply `X` on the left (`side::left`) or on the right (`side::right`). See *oneMKL defined datatypes* for more details.

upper_lower
Specifies whether the matrices `A` are upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans
Specifies op(`A`), the transposition operation applied to the matrices `A`. See *oneMKL defined datatypes* for more details.

unit_diag
Specifies whether the matrices `A` are assumed to be unit triangular (all diagonal elements are 1). See *oneMKL defined datatypes* for more details.

m
Number of rows of the `B` matrices. Must be at least zero.

n
Number of columns of the `B` matrices. Must be at least zero.

alpha
Scaling factor for the solutions.

a
Buffer holding the input matrices `A` with size `stridea` * `batch_size`.

lda
Leading dimension of the matrices `A`. Must be at least `m` if `left_right` = `side::left`, and at least `n` if `left_right` = `side::right`. Must be positive.

stridea
Stride between different `A` matrices.

b
Buffer holding the input matrices `B` with size `strideb` * `batch_size`.

ldb
Leading dimension of the matrices `B`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if row major layout is used to store matrices.

strideb
Stride between different `B` matrices.

batch_size
Specifies the number of triangular linear systems to solve.

Output Parameters

b
Output buffer, overwritten by `batch_size` solution matrices `X`.

Notes

If `alpha` = 0, matrix `B` is set to zero and the matrices `A` and `B` do not need to be initialized before calling `trsm_batch`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

trsm_batch (USM Version)

Description

The USM version of `trsm_batch` supports the group API and strided API.

The group API operation is defined as:

```
idx = 0
for i = 0 ... group_count - 1
    for j = 0 ... group_size - 1
        A and B are matrices in a[idx] and b[idx]
        if (left_right == oneapi::mkl::side::left) then
            compute X such that op(A) * X = alpha[i] * B
        else
            compute X such that X * op(A) = alpha[i] * B
        end if
        B := X
```

(continues on next page)

(continued from previous page)

```
        idx = idx + 1
    end for
end for
```

The strided API operation is defined as:

```
for i = 0 ... batch_size - 1
    A and B are matrices at offset i * stridea and i * strideb in a and b.
    if (left_right == oneapi::mkl::side::left) then
        compute X such that op(A) * X = alpha * B
    else
        compute X such that X * op(A) = alpha * B
    end if
    B := X
end for

where:
```

op(A) is one of op(A) = A, or op(A) = A^T, or op(A) = A^H,

alpha is a scalar,

A is a triangular matrix,

B and X are m x n general matrices,

A is either m x m or n x n,depending on whether it multiplies X on the left or right. On return, the matrix B is overwritten by the solution matrix X.

For group API, a and b arrays contain the pointers for all the input matrices. The total number of matrices in a and b are given by:

$$total\_batch\_count = \sum_{i=0}^{group\_count-1} group\_size[i]$$

For strided API, a and b arrays contain all the input matrices. The total number of matrices in a and b are given by the batch_size parameter.

Group API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event trsm_batch(sycl::queue &queue,
                           const oneapi::mkl::side *left_right,
                           const oneapi::mkl::uplo *upper_lower,
                           const oneapi::mkl::transpose *trans,
                           const oneapi::mkl::diag *unit_diag,
                           const std::int64_t *m,
                           const std::int64_t *n,
                           const T *alpha,
                           const T **a,
                           const std::int64_t *lda,
                           T **b,
```

(continues on next page)

(continued from previous page)

```
                            const std::int64_t *ldb,
                            std::int64_t group_count,
                            const std::int64_t *group_size,
                            const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event trsm_batch(sycl::queue &queue,
                            const oneapi::mkl::side *left_right,
                            const oneapi::mkl::uplo *upper_lower,
                            const oneapi::mkl::transpose *trans,
                            const oneapi::mkl::diag *unit_diag,
                            const std::int64_t *m,
                            const std::int64_t *n,
                            const T *alpha,
                            const T **a,
                            const std::int64_t *lda,
                            T **b,
                            const std::int64_t *ldb,
                            std::int64_t group_count,
                            const std::int64_t *group_size,
                            const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
: The queue where the routine should be executed.

left_right
: Array of `group_count` `oneapi::mkl::side` values. `left_right[i]` specifies whether `A` multiplies `X` on the left (`side::left`) or on the right (`side::right`) for every `trsm` operation in group `i`. See *oneMKL defined datatypes* for more details.

upper_lower
: Array of `group_count` `oneapi::mkl::uplo` values. `upper_lower[i]` specifies whether `A` is upper or lower triangular for every matrix in group `i`. See *oneMKL defined datatypes* for more details.

trans
: Array of `group_count` `oneapi::mkl::transpose` values. `trans[i]` specifies the form of op(`A`) used for every `trsm` operation in group `i`. See *oneMKL defined datatypes* for more details.

unit_diag
: Array of `group_count` `oneapi::mkl::diag` values. `unit_diag[i]` specifies whether `A` is assumed to be unit triangular (all diagonal elements are 1) for every matrix in group `i`. See *oneMKL defined datatypes* for more details.

m
: Array of `group_count` integers. `m[i]` specifies the number of rows of `B` for every matrix in group `i`. All entries must be at least zero.

n
: Array of `group_count` integers. `n[i]` specifies the number of columns of `B` for every matrix in group `i`. All entries must be at least zero.

alpha
Array of `group_count` scalar elements. `alpha[i]` specifies the scaling factor in group `i`.

a
Array of pointers to input matrices `A` with size `total_batch_count`. See *Matrix Storage* for more details.

lda
Array of `group_count` integers. `lda[i]` specifies the leading dimension of `A` for every matrix in group `i`. All entries must be at least `m` if `left_right` is `side::left`, and at least `n` if `left_right` is `side::right`. All entries must be positive.

b
Array of pointers to input matrices `B` with size `total_batch_count`. See *Matrix Storage* for more details.

ldb
Array of `group_count` integers. `ldb[i]` specifies the leading dimension of `B` for every matrix in group `i`. All entries must be positive and at least `m` and positive if column major layout is used to store matrices or at least `n` if row major layout is used to store matrices.

group_count
Specifies the number of groups. Must be at least 0.

group_size
Array of `group_count` integers. `group_size[i]` specifies the number of `trsm` operations in group `i`. All entries must be at least 0.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

b
Output buffer, overwritten by the `total_batch_count` solution matrices `X`.

Notes

If `alpha` = 0, matrix `B` is set to zero and the matrices `A` and `B` do not need to be initialized before calling `trsm_batch`.

Return Values

Output event to wait on to ensure computation is complete.

Strided API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event trsm_batch(sycl::queue &queue,
                           oneapi::mkl::side left_right,
                           oneapi::mkl::uplo upper_lower,
                           oneapi::mkl::transpose trans,
                           oneapi::mkl::diag unit_diag,
                           std::int64_t m,
                           std::int64_t n,
```

(continues on next page)

(continued from previous page)

```
                            value_or_pointer<T> alpha,
                            const T *a,
                            std::int64_t lda,
                            std::int64_t stridea,
                            T *b,
                            std::int64_t ldb,
                            std::int64_t strideb,
                            std::int64_t batch_size,
                            const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event trsm_batch(sycl::queue &queue,
                            oneapi::mkl::side left_right,
                            oneapi::mkl::uplo upper_lower,
                            oneapi::mkl::transpose trans,
                            oneapi::mkl::diag unit_diag,
                            std::int64_t m,
                            std::int64_t n,
                            value_or_pointer<T> alpha,
                            const T *a,
                            std::int64_t lda,
                            std::int64_t stridea,
                            T *b,
                            std::int64_t ldb,
                            std::int64_t strideb,
                            std::int64_t batch_size,
                            const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

left_right
Specifies whether the matrices `A` multiply `X` on the left (`side::left`) or on the right (`side::right`). See *oneMKL defined datatypes* for more details.

upper_lower
Specifies whether the matrices `A` are upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans
Specifies op(`A`), the transposition operation applied to the matrices `A`. See *oneMKL defined datatypes* for more details.

unit_diag
Specifies whether the matrices `A` are assumed to be unit triangular (all diagonal elements are 1). See *oneMKL defined datatypes* for more details.

m
Number of rows of the `B` matrices. Must be at least zero.

n
Number of columns of the `B` matrices. Must be at least zero.

alpha
Scaling factor for the solutions. See *Scalar Arguments in BLAS* for more details.

a
Pointer to input matrices `A` with size `stridea` * `batch_size`.

lda
Leading dimension of the matrices `A`. Must be at least `m` if `left_right` = `side::left`, and at least `n` if `left_right` = `side::right`. Must be positive.

stridea
Stride between different `A` matrices.

b
Pointer to input matrices `B` with size `strideb` * `batch_size`.

ldb
Leading dimension of the matrices `B`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if row major layout is used to store matrices.

strideb
Stride between different `B` matrices.

batch_size
Specifies the number of triangular linear systems to solve.

Output Parameters

b
Output matrices, overwritten by `batch_size` solution matrices `X`.

Notes

If `alpha` = 0, matrix `B` is set to zero and the matrices `A` and `B` do not need to be initialized before calling `trsm_batch`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS-like Extensions*

gemmt

Computes a matrix-matrix product with general matrices, but updates only the upper or lower triangular part of the result matrix.

Description

The gemmt routines compute a scalar-matrix-matrix product and add the result to the upper or lower part of a scalar-matrix product, with general matrices. The operation is defined as:

$$C \leftarrow alpha * op(A) * op(B) + beta * C$$

where:

op(`X`) is one of op(`X`) = `X`, or op(`X`) = $\mathtt{X}^{\mathrm{T}}$, or op(`X`) = $\mathtt{X}^{\mathrm{H}}$,

`alpha` and `beta` are scalars

`A`, `B`, and `C` are matrices

op(`A`) is `m` x `k`, op(`B`) is `k` x `n`, and `C` is `m` x `n`.

`gemmt` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

gemmt (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void gemmt(sycl::queue &queue,
               oneapi::mkl::uplo upper_lower,
               oneapi::mkl::transpose transa,
               oneapi::mkl::transpose transb,
               std::int64_t n,
               std::int64_t k,
               T alpha,
               sycl::buffer<T,1> &a,
               std::int64_t lda,
               sycl::buffer<T,1> &b,
               std::int64_t ldb,
               T beta,
               sycl::buffer<T,1> &c,
               std::int64_t ldc)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void gemmt(sycl::queue &queue,
               oneapi::mkl::uplo upper_lower,
               oneapi::mkl::transpose transa,
               oneapi::mkl::transpose transb,
               std::int64_t n,
               std::int64_t k,
               T alpha,
               sycl::buffer<T,1> &a,
               std::int64_t lda,
               sycl::buffer<T,1> &b,
               std::int64_t ldb,
               T beta,
               sycl::buffer<T,1> &c,
               std::int64_t ldc)
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether `C`'s data is stored in its upper or lower triangle. See *oneMKL defined datatypes* for more details.

transa
Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

transb
Specifies op(`B`), the transposition operation applied to `B`. See *oneMKL defined datatypes* for more details.

n
Number of rows of op(`A`), columns of op(`B`), and columns and rows of `C`. Must be at least zero.

k
Number of columns of op(`A`) and rows of op(`B`). Must be at least zero.

alpha
Scaling factor for the matrix-matrix product.

a
Buffer holding the input matrix `A`.

	`A` not transposed	`A` transposed
Column major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*k`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*n`
Row major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*n`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*k`.

See *Matrix Storage* for more details.

lda
The leading dimension of `A`. It must be positive.

	A not transposed	A transposed
Column major	lda must be at least n.	lda must be at least k.
Row major	lda must be at least k.	lda must be at least n.

b

Buffer holding the input matrix B.

	B not transposed	B transposed
Column major	B is an k-by-n matrix so the array b must have size at least ldb*n.	B is an n-by-k matrix so the array b must have size at least ldb*k
Row major	B is an k-by-n matrix so the array b must have size at least ldb*k.	B is an n-by-k matrix so the array b must have size at least ldb*n.

See *Matrix Storage* for more details.

ldb

The leading dimension of B. It must be positive.

	B not transposed	B transposed
Column major	ldb must be at least k.	ldb must be at least n.
Row major	ldb must be at least n.	ldb must be at least k.

beta

Scaling factor for matrix C.

c

Buffer holding the input/output matrix C. Must have size at least ldc * n. See *Matrix Storage* for more details.

ldc

Leading dimension of C. Must be positive and at least m.

Output Parameters

c

Output buffer, overwritten by the upper or lower triangular part of alpha * op(A)*op(B) + beta * C.

Notes

If beta = 0, matrix C does not need to be initialized before calling gemmt.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

gemmt (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event gemmt(sycl::queue &queue,
                      oneapi::mkl::uplo upper_lower,
                      oneapi::mkl::transpose transa,
                      oneapi::mkl::transpose transb,
                      std::int64_t n,
                      std::int64_t k,
                      value_or_pointer<T> alpha,
                      const T *a,
                      std::int64_t lda,
                      const T *b,
                      std::int64_t ldb,
                      value_or_pointer<T> beta,
                      T *c,
                      std::int64_t ldc,
                      const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event gemmt(sycl::queue &queue,
                      oneapi::mkl::uplo upper_lower,
                      oneapi::mkl::transpose transa,
                      oneapi::mkl::transpose transb,
                      std::int64_t n,
                      std::int64_t k,
                      value_or_pointer<T> alpha,
                      const T *a,
                      std::int64_t lda,
                      const T *b,
                      std::int64_t ldb,
                      value_or_pointer<T> beta,
                      T *c,
                      std::int64_t ldc,
                      const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Specifies whether C's data is stored in its upper or lower triangle. See *oneMKL defined datatypes* for more details.

transa
Specifies op(A), the transposition operation applied to A. See *oneMKL defined datatypes* for more details.

transb
Specifies op(B), the transposition operation applied to B. See *oneMKL defined datatypes* for more details.

n
Number of columns of op(A), columns of op(B), and columns ofC. Must be at least zero.

k
Number of columns of op(A) and rows of op(B). Must be at least zero.

alpha
Scaling factor for the matrix-matrix product. See *Scalar Arguments in BLAS* for more details.

a
Pointer to input matrix A.

	A not transposed	A transposed
Column major	A is an n-by-k matrix so the array a must have size at least lda*k.	A is an k-by-n matrix so the array a must have size at least lda*n
Row major	A is an n-by-k matrix so the array a must have size at least lda*n.	A is an k-by-n matrix so the array a must have size at least lda*k

See *Matrix Storage* for more details.

lda
The leading dimension of A. It must be positive.

	A not transposed	A transposed
Column major	lda must be at least n.	lda must be at least k.
Row major	lda must be at least k.	lda must be at least n.

b
Pointer to input matrix B.

	B not transposed	B transposed
Column major	B is an k-by-n matrix so the array b must have size at least ldb*n.	B is an n-by-k matrix so the array b must have size at least ldb*k
Row major	B is an k-by-n matrix so the array b must have size at least ldb*k.	B is an n-by-k matrix so the array b must have size at least ldb*n

See *Matrix Storage* for more details.

ldb
The leading dimension of B. It must be positive.

	B not transposed	B transposed
Column major	ldb must be at least k.	ldb must be at least n.
Row major	ldb must be at least n.	ldb must be at least k.

beta
Scaling factor for matrix C. See *Scalar Arguments in BLAS* for more details.

c
Pointer to input/output matrix C. Must have size at least ldc * n. See *Matrix Storage* for more details.

ldc
Leading dimension of C. Must be positive and at least m.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

c
Pointer to the output matrix, overwritten by the upper or lower triangular part of alpha * op(A)*op(B) + beta * C.

Notes

If beta = 0, matrix C does not need to be initialized before calling gemmt.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS-like Extensions*

gemm_bias

Computes a matrix-matrix product using general integer matrices with bias.

Description

The gemm_bias routines compute a scalar-matrix-matrix product and add the result to a scalar-matrix product, using general integer matrices with biases/offsets. The operation is defined as:

$$C \leftarrow alpha*(op(A)-A\_offset)*(op(B)-B\_offset)+beta*C+C\_offset$$

where:

op(`X`) is one of op(`X`) = `X`, or op(`X`) = $\mathtt{X}^{\mathrm{T}}$, or op(`X`) = $\mathtt{X}^{\mathrm{H}}$,

`alpha` and `beta` are scalars,

`A_offset` is an `m`-by-`k` matrix with every element equal to the value ao,

`B_offset` is a `k`-by-`n` matrix with every element equal to the value bo,

`C_offset` is an `m`-by-`n` matrix defined by the co buffer as described below,

`A`, `B`, and `C` are matrices,

op(`A`) is `m` x `k`, op(`B`) is `k` x `n`, and `C` is `m` x `n`.

`gemm_bias` supports the following precisions.

Ta	Tb
std::uint8_t	std::uint8_t
std::int8_t	std::uint8_t
std::uint8_t	std::int8_t
std::int8_t	std::int8_t

gemm_bias (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void gemm_bias(sycl::queue &queue,
                   oneapi::mkl::transpose transa,
                   oneapi::mkl::transpose transb,
                   oneapi::mkl::offset offset_type,
                   std::int64_t m,
                   std::int64_t n,
                   std::int64_t k,
                   float alpha,
                   sycl::buffer<Ta,1> &a,
                   std::int64_t lda,
                   Ta ao,
                   sycl::buffer<Tb,1> &b,
                   std::int64_t ldb,
                   Tb bo,
```

(continues on next page)

(continued from previous page)

```
                    float beta,
                    sycl::buffer<std::int32_t,1> &c,
                    std::int64_t ldc,
                    sycl::buffer<std::int32_t,1> &co)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void gemm_bias(sycl::queue &queue,
                    oneapi::mkl::transpose transa,
                    oneapi::mkl::transpose transb,
                    oneapi::mkl::offset offset_type,
                    std::int64_t m,
                    std::int64_t n,
                    std::int64_t k,
                    float alpha,
                    sycl::buffer<Ta,1> &a,
                    std::int64_t lda,
                    Ta ao,
                    sycl::buffer<Tb,1> &b,
                    std::int64_t ldb,
                    Tb bo,
                    float beta,
                    sycl::buffer<std::int32_t,1> &c,
                    std::int64_t ldc,
                    sycl::buffer<std::int32_t,1> &co)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

transa
: Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

transb
: Specifies op(`B`), the transposition operation applied to `B`. See *oneMKL defined datatypes* for more details.

offset_type
: Specifies the form of `C_offset` used in the matrix multiplication. See *oneMKL defined datatypes* for more details.

m
: Number of rows of op(`A`) and `C`. Must be at least zero.

n
: Number of columns of op(`B`) and `C`. Must be at least zero.

k
: Number of columns of op(`A`) and rows of op(`B`). Must be at least zero.

alpha
: Scaling factor for the matrix-matrix product.

a

The buffer holding the input matrix `A`.

	`A` not transposed	`A` transposed
Column major	`A` is an `m`-by-`k` matrix so the array `a` must have size at least `lda*k`.	`A` is an `k`-by-`m` matrix so the array `a` must have size at least `lda*m`
Row major	`A` is an `m`-by-`k` matrix so the array `a` must have size at least `lda*m`.	`A` is an `k`-by-`m` matrix so the array `a` must have size at least `lda*k`

See *Matrix Storage* for more details.

lda

The leading dimension of `A`. It must be positive.

	`A` not transposed	`A` transposed
Column major	`lda` must be at least `m`.	`lda` must be at least `k`.
Row major	`lda` must be at least `k`.	`lda` must be at least `m`.

ao

Specifies the scalar offset value for matrix `A`.

b

Buffer holding the input matrix `B`.

	`B` not transposed	`B` transposed
Column major	`B` is an `k`-by-`n` matrix so the array `b` must have size at least `ldb*n`.	`B` is an `n`-by-`k` matrix so the array `b` must have size at least `ldb*k`
Row major	`B` is an `k`-by-`n` matrix so the array `b` must have size at least `ldb*k`.	`B` is an `n`-by-`k` matrix so the array `b` must have size at least `ldb*n`

See *Matrix Storage* for more details.

ldb

The leading dimension of `B`. It must be positive.

	`B` not transposed	`B` transposed
Column major	`ldb` must be at least `k`.	`ldb` must be at least `n`.
Row major	`ldb` must be at least `n`.	`ldb` must be at least `k`.

bo

Specifies the scalar offset value for matrix `B`.

beta

Scaling factor for matrix `C`.

c

Buffer holding the input/output matrix `C`. It must have a size of at least `ldc*n` if column major layout is used to store matrices or at least `ldc*m` if row major layout is used to store matrices . See *Matrix Storage* for more details.

ldc

The leading dimension of `C`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if row major layout is used to store matrices.

co

Buffer holding the offset values for matrix `C`.

If `offset_type` = `offset::fix`, the `co` array must have size at least 1.

If `offset_type` = `offset::col`, the `co` array must have size at least `max(1,m)`.

If `offset_type` = `offset::row`, the `co` array must have size at least `max(1,n)`.

Output Parameters

c

Output buffer, overwritten by `alpha` * (op(`A`) - `A_offset`)*(op(`B`) - `B_offset`) + `beta` * `C` + `C_offset`.

Notes

If `beta` = 0, matrix `C` does not need to be initialized before calling `gemm_bias`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

gemm_bias (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event gemm_bias(sycl::queue &queue,
                          oneapi::mkl::transpose transa,
                          oneapi::mkl::transpose transb,
                          oneapi::mkl::offset offset_type,
                          std::int64_t m,
                          std::int64_t n,
                          std::int64_t k,
                          value_or_pointer<float> alpha,
                          const Ta *a,
                          std::int64_t lda,
                          Ta ao,
                          const Tb *b,
```

(continues on next page)

(continued from previous page)

```
                          std::int64_t ldb,
                          Tb bo,
                          value_or_pointer<float> beta,
                          std::int32_t *c,
                          std::int64_t ldc,
                          const std::int32_t *co,
                          const std::vector<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event gemm_bias(sycl::queue &queue,
                          oneapi::mkl::transpose transa,
                          oneapi::mkl::transpose transb,
                          oneapi::mkl::offset offset_type,
                          std::int64_t m,
                          std::int64_t n,
                          std::int64_t k,
                          value_or_pointer<float> alpha,
                          const Ta *a,
                          std::int64_t lda,
                          Ta ao,
                          const Tb *b,
                          std::int64_t ldb,
                          Tb bo,
                          value_or_pointer<float> beta,
                          std::int32_t *c,
                          std::int64_t ldc,
                          const std::int32_t *co,
                          const std::vector<sycl::event> &dependencies = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

transa
Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

transb
Specifies op(`B`), the transposition operation applied to `B`. See *oneMKL defined datatypes* for more details.

offset_type
Specifies the form of `C_offset` used in the matrix multiplication. See *oneMKL defined datatypes* for more details.

m
Number of rows of op(`A`) and `C`. Must be at least zero.

n
Number of columns of op(`B`) and `C`. Must be at least zero.

k
Number of columns of op(`A`) and rows of op(`B`). Must be at least zero.

alpha
Scaling factor for the matrix-matrix product. See *Scalar Arguments in BLAS* for more details.

a
Pointer to input matrix `A`.

	`A` not transposed	`A` transposed
Column major	`A` is an `m`-by-`k` matrix so the array `a` must have size at least `lda*k`.	`A` is an `k`-by-`m` matrix so the array `a` must have size at least `lda*m`
Row major	`A` is an `m`-by-`k` matrix so the array `a` must have size at least `lda*m`.	`A` is an `k`-by-`m` matrix so the array `a` must have size at least `lda*k`

See *Matrix Storage* for more details.

lda
The leading dimension of `A`. It must be positive.

	`A` not transposed	`A` transposed
Column major	`lda` must be at least `m`.	`lda` must be at least `k`.
Row major	`lda` must be at least `k`.	`lda` must be at least `m`.

ao
Specifies the scalar offset value for matrix `A`.

b
Pointer to input matrix `B`.

	`B` not transposed	`B` transposed
Column major	`B` is an `k`-by-`n` matrix so the array `b` must have size at least `ldb*n`.	`B` is an `n`-by-`k` matrix so the array `b` must have size at least `ldb*k`
Row major	`B` is an `k`-by-`n` matrix so the array `b` must have size at least `ldb*k`.	`B` is an `n`-by-`k` matrix so the array `b` must have size at least `ldb*n`

See *Matrix Storage* for more details.

ldb
The leading dimension of `B`. It must be positive.

	`B` not transposed	`B` transposed
Column major	`ldb` must be at least `k`.	`ldb` must be at least `n`.
Row major	`ldb` must be at least `n`.	`ldb` must be at least `k`.

bo
Specifies the scalar offset value for matrix `B`.

beta
Scaling factor for matrix `C`. See *Scalar Arguments in BLAS* for more details.

c
Pointer to input/output matrix `C`. It must have a size of at least `ldc*n` if column major layout is used to store matrices or at least `ldc*m` if row major layout is used to store matrices . See *Matrix Storage* for more details.

ldc
The leading dimension of `C`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if row major layout is used to store matrices.

co
Pointer to offset values for matrix `C`.

If `offset_type` = `offset::fix`, the `co` array must have size at least 1.

If `offset_type` = `offset::col`, the `co` array must have size at least `max(1,m)`.

If `offset_type` = `offset::row`, the `co` array must have size at least `max(1,n)`.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

c
Pointer to the output matrix, overwritten by `alpha` * (op(`A`) - `A_offset`)*(op(`B`) - `B_offset`) + `beta` * `C` + `C_offset`.

Notes

If `beta` = 0, matrix `C` does not need to be initialized before calling `gemm_bias`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS-like Extensions*

imatcopy

Computes an in-place scaled matrix transpose or copy operation using a general dense matrix.

Description

The `imatcopy` routine performs an in-place scaled matrix copy or transposition.

The operation is defined as:

$$C \leftarrow \alpha * op(C)$$

where:

op(X) is one of op(X) = X, or op(X) = X^T, or op(X) = X^H,

`alpha` is a scalar,

`C` is a matrix to be transformed in place,

and `C` is `m` x `n` on input.

`imatcopy` supports the following precisions:

T
float
double
std::complex<float>
std::complex<double>

imatcopy (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void imatcopy(sycl::queue &queue,
                  oneapi::mkl::transpose trans,
                  std::int64_t m,
                  std::int64_t n,
                  T alpha,
                  sycl::buffer<T, 1> &matrix_in_out,
                  std::int64_t ld_in,
                  std::int64_t ld_out);
}
```

```
namespace oneapi::mkl::blas::row_major {
    void imatcopy(sycl::queue &queue,
                  oneapi::mkl::transpose trans,
                  std::int64_t m,
                  std::int64_t n,
                  T alpha,
                  sycl::buffer<T, 1> &matrix_in_out,
```

(continues on next page)

(continued from previous page)

```
                    std::int64_t ld_in,
                    std::int64_t ld_out);
}
```

Input Parameters

queue
The queue where the routine should be executed.

trans
Specifies op(C), the transposition operation applied to the matrix C. See *oneMKL defined datatypes* for more details.

m
Number of rows of C on input. Must be at least zero.

n
Number of columns of C on input. Must be at least zero.

alpha
Scaling factor for the matrix transposition or copy.

matrix_in_out
Buffer holding the input/output matrix C. Must have size as follows:

	trans = transpose::nontrans	trans = transpose::trans or trans = transpose::conjtrans
Column major	Size of array matrix_in_out must be at least max(ld_in, ld_out) * n	Size of array matrix_in_out must be at least max(ld_in``*``n, ld_out``*``m)
Row major	Size of array matrix_in_out must be at least max(ld_in, ld_out) * m	Size of array matrix_in_out must be at least max(ld_in``*``m, ld_out``*``n)

ld_in
The leading dimension of the matrix C on input. It must be positive, and must be at least m if column major layout is used, and at least n if row-major layout is used.

ld_out
The leading dimension of the matrix C on output. It must be positive.

	trans = transpose::nontrans	trans = transpose::trans or trans = transpose::conjtrans
Column major	ld_out must be at least m.	ld_out must be at least n.
Row major	ld_out must be at least n.	ld_out must be at least m.

Output Parameters

matrix_in_out
Output buffer, overwritten by `alpha` * op(`C`).

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

imatcopy (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event imatcopy(sycl::queue &queue,
                         oneapi::mkl::transpose trans,
                         std::int64_t m,
                         std::int64_t n,
                         value_or_pointer<T> alpha,
                         T *matrix_in_out,
                         std::int64_t ld_in,
                         std::int64_t ld_out,
                         const std::vector<sycl::event> &dependencies = {});
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event imatcopy(sycl::queue &queue,
                         oneapi::mkl::transpose trans,
                         std::int64_t m,
                         std::int64_t n,
                         value_or_pointer<T> alpha,
                         T *matrix_in_out,
                         std::int64_t ld_in,
                         std::int64_t ld_out,
                         const std::vector<sycl::event> &dependencies = {});
```

Input Parameters

queue
The queue where the routine will be executed.

trans
Specifies op(C), the transposition operation applied to the matrix C. See *oneMKL defined datatypes* for more details.

m
Number of rows for the matrix C on input. Must be at least zero.

n
Number of columns for the matrix C on input. Must be at least zero.

alpha
Scaling factor for the matrix transpose or copy operation. See *Scalar Arguments in BLAS* for more details.

matrix_in_out

Pointer to input/output matrix C. Must have size as follows:

	trans = transpose::nontrans	trans = transpose::trans or trans = transpose::conjtrans
Column major	Size of array matrix_in_out must be at least max(ld_in, ld_out) * n	Size of array matrix_in_out must be at least max(ld_in``*``n, ld_out``*``m)
Row major	Size of array matrix_in_out must be at least max(ld_in, ld_out) * m	Size of array matrix_in_out must be at least max(ld_in``*``m, ld_out``*``n)

ld_in
Leading dimension of the matrix C on input. If matrices are stored using column major layout, ld_in must be at least m. If matrices are stored using row major layout, ld_in must be at least n. Must be positive.

ld_out
Leading dimension of the matrix C on output. Must be positive.

	trans = transpose::nontrans	trans = transpose::trans or trans = transpose::conjtrans
Column major	ld_out must be at least m.	ld_out must be at least n.
Row major	ld_out must be at least n.	ld_out must be at least m.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

matrix_in_out
Pointer to output matrix `C` overwritten by `alpha` * op(`C`).

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS-like Extensions*

omatcopy

Computes an out-of-place scaled matrix transpose or copy operation using a general dense matrix.

Description

The `omatcopy` routine performs an out-of-place scaled matrix copy or transposition.

The operation is defined as:

$$B \leftarrow \alpha * op(A)$$

where:

op(`X`) is one of op(`X`) = `X`, or op(`X`) = $\mathtt{X}^{\mathrm{T}}$, or op(`X`) = $\mathtt{X}^{\mathrm{H}}$,

`alpha` is a scalar,

`A` and `B` are matrices,

`A` is `m` x `n` matrix,

`B` is `m` x `n` matrix if `op` is non-transpose and an `n` x `m` matrix otherwise.,

`omatcopy` supports the following precisions:

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

omatcopy (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void omatcopy(sycl::queue &queue,
                  oneapi::mkl::transpose trans,
                  std::int64_t m,
                  std::int64_t n,
                  T alpha,
                  sycl::buffer<T, 1> &a,
                  std::int64_t lda,
                  sycl::buffer<T, 1> &b,
                  std::int64_t ldb);
}
```

```
namespace oneapi::mkl::blas::row_major {
    void omatcopy(sycl::queue &queue,
                  oneapi::mkl::transpose trans,
                  std::int64_t m,
                  std::int64_t n,
                  T alpha,
                  sycl::buffer<T, 1> &a,
                  std::int64_t lda,
                  sycl::buffer<T, 1> &b,
                  std::int64_t ldb);
}
```

Input Parameters

queue
: The queue where the routine should be executed.

trans
: Specifies op(`A`), the transposition operation applied to the matrix `A`. See *oneMKL defined datatypes* for more details.

m
: Number of rows for the matrix `A`. Must be at least zero.

n
: Number of columns for the matrix `A`. Must be at least zero.

alpha
: Scaling factor for the matrix transposition or copy.

a
: Buffer holding the input matrix `A`. Must have size at least `lda` * `n` for column-major and at least `lda` * `m` for row-major.

lda
: Leading dimension of the matrix `A`. If matrices are stored using column major layout, `lda` must be at least `m`. If matrices are stored using row major layout, `lda` must be at least `n`. Must be positive.

b

Buffer holding the output matrix B.

	trans = transpose::nontrans	trans = transpose::trans or trans = transpose::conjtrans
Column major	B is m x n matrix. Size of array b must be at least ldb * n	B is n x m matrix. Size of array b must be at least ldb * m
Row major	B is m x n matrix. Size of array b must be at least ldb * m	B is n x m matrix. Size of array b must be at least ldb * n

ldb

The leading dimension of the matrix B. Must be positive.

	trans = transpose::nontrans	trans = transpose::trans or trans = transpose::conjtrans
Column major	ldb must be at least m.	ldb must be at least n.
Row major	ldb must be at least n.	ldb must be at least m.

Output Parameters

b

Output buffer, overwritten by alpha * op(A).

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

omatcopy (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event omatcopy(sycl::queue &queue,
                         oneapi::mkl::transpose trans,
                         std::int64_t m,
                         std::int64_t n,
                         value_or_pointer<T> alpha,
                         const T *a,
```

(continues on next page)

(continued from previous page)

```
                        std::int64_t lda,
                        T *b,
                        std::int64_t ldb,
                        const std::vector<sycl::event> &dependencies = {});
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event omatcopy(sycl::queue &queue,
                        oneapi::mkl::transpose trans,
                        std::int64_t m,
                        std::int64_t n,
                        value_or_pointer<T> alpha,
                        const T *a,
                        std::int64_t lda,
                        T *b,
                        std::int64_t ldb,
                        const std::vector<sycl::event> &dependencies = {});
}
```

Input Parameters

queue
The queue where the routine will be executed.

trans
Specifies op(`A`), the transposition operation applied to the matrix `A`. See *oneMKL defined datatypes* for more details.

m
Number of rows for the matrix `A`. Must be at least zero.

n
Number of columns for the matrix `A`. Must be at least zero.

alpha
Scaling factor for the matrix transposition or copy. See *Scalar Arguments in BLAS* for more details.

a
Pointer to input matrix `A`. Must have size at least `lda` * `n` for column-major and at least `lda` * `m` for row-major.

lda
Leading dimension of the matrix `A`. If matrices are stored using column major layout, `lda` must be at least `m`. If matrices are stored using row major layout, `lda` must be at least `n`. Must be positive.

b
Pointer to output matrix `B`.

	`trans = transpose::nontrans`	`trans = transpose::trans` or `trans = transpose::conjtrans`
Column major	`B` is `m` x `n` matrix. Size of array `b` must be at least `ldb` * `n`	`B` is `n` x `m` matrix. Size of array `b` must be at least `ldb` * `m`
Row major	`B` is `m` x `n` matrix. Size of array `b` must be at least `ldb` * `m`	`B` is `n` x `m` matrix. Size of array `b` must be at least `ldb` * `n`

ldb
Leading dimension of the matrix `B`. Must be positive.

	`trans = transpose::nontrans`	`trans = transpose::trans` or `trans = transpose::conjtrans`
Column major	Must be at least `m`	Must be at least `n`
Row major	Must be at least `n`	Must be at least `m`

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

b
Pointer to output matrix `B` overwritten by `alpha` * op(`A`).

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS-like Extensions*

omatcopy2

Computes two-strided scaling and out-of-place transposition or copying of general dense matrices.

Description

The `omatcopy2` routine performs a two-strided scaling and out-of-place transposition or copy of matrices. For complex matrices the transpose operation can be a conjugate transpose.

Normally, matrices in the BLAS or LAPACK are specified by a single stride index. For instance, in the column-major order, `A(2,1)` is stored in memory one element away from `A(1,1)`, but `A(1,2)` is a leading dimension away. The leading dimension in this case is at least the number of rows of the source matrix. If a matrix has two strides, then both `A(2,1)` and `A(1,2)` may be an arbitrary distance from `A(1,1)`.

The operation is defined as:

$$B \leftarrow \alpha * op(A)$$

where:

op(X) is one of op(X) = X, or op(X) = X^T, or op(X) = X^H,

alpha is a scalar,

A and B are matrices,

A is m x n matrix,

B is m x n matrix if op is non-transpose and an n x m matrix otherwise.,

omatcopy2 supports the following precisions:

T
float
double
std::complex<float>
std::complex<double>

omatcopy2 (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void omatcopy2(sycl::queue &queue,
                   oneapi::mkl::transpose trans,
                   std::int64_t m,
                   std::int64_t n,
                   T alpha,
                   sycl::buffer<T, 1> &a,
                   std::int64_t lda,
                   std::int64_t stridea,
                   sycl::buffer<T, 1> &b,
                   std::int64_t ldb,
                   std::int64_t strideb);
}
```

```
namespace oneapi::mkl::blas::row_major {
    void omatcopy2(sycl::queue &queue,
                   oneapi::mkl::transpose trans,
                   std::int64_t m,
                   std::int64_t n,
                   T alpha,
                   sycl::buffer<T, 1> &a,
                   std::int64_t lda,
                   std::int64_t stridea,
                   sycl::buffer<T, 1> &b,
                   std::int64_t ldb,
```

(continues on next page)

(continued from previous page)

```
                    std::int64_t strideb);
}
```

Input Parameters

queue
The queue where the routine should be executed.

trans
Specifies op(A), the transposition operation applied to the matrix A. See *oneMKL defined datatypes* for more details.

m
Number of rows for the matrix A. Must be at least zero.

n
Number of columns for the matrix A. Must be at least zero.

alpha
Scaling factor for the matrix transposition or copy.

a
Buffer holding the input matrix A. Must have size at least lda * n for column major ordering and at least lda * m for row major ordering.

lda
Leading dimension of the matrix A. If matrices are stored using column major layout, lda is the number of elements in the array between adjacent columns of the matrix, and must be at least stridea * (m-1) + 1. If using row major layout, lda is the number of elements between adjacent rows of the matrix and must be at least stridea * (n-1) + 1.

stridea
The second stride of the matrix A. For column major layout, stridea is the number of elements in the array between adjacent rows of the matrix. For row major layout stridea is the number of elements between adjacent columns of the matrix. In both cases stridea must be at least 1.

b
Buffer holding the output matrix B.

	trans = transpose::nontrans	trans = transpose::trans or trans = transpose::conjtrans
Column major	B is m x n matrix. Size of buffer b must be at least ldb * n	B is n x m matrix. Size of buffer b must be at least ldb * m
Row major	B is m x n matrix. Size of buffer b must be at least ldb * m	B is n x m matrix. Size of buffer b must be at least ldb * n

ldb
The leading dimension of the matrix B. Must be positive.

	trans = transpose::nontrans	trans = transpose::trans or trans = transpose::conjtrans
Column major	ldb must be at least strideb * (m-1) + 1.	ldb must be at least strideb * (n-1) + 1.
Row major	ldb must be at least strideb * (n-1) + 1.	ldb must be at least strideb * (m-1) + 1.

strideb
The second stride of the matrix B. For column major layout, strideb is the number of elements in the array between adjacent rows of the matrix. For row major layout, strideb is the number of elements between adjacent columns of the matrix. In both cases strideb must be at least 1.

Output Parameters

b
Output buffer, overwritten by alpha * op(A).

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

omatcopy2 (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event omatcopy2(sycl::queue &queue,
                          oneapi::mkl::transpose trans,
                          std::int64_t m,
                          std::int64_t n,
                          value_or_pointer<T> alpha,
                          const T *a,
                          std::int64_t lda,
                          std::int64_t stridea,
                          T *b,
                          std::int64_t ldb,
                          std::int64_t strideb,
                          const std::vector<sycl::event> &dependencies = {});
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event omatcopy2(sycl::queue &queue,
                          oneapi::mkl::transpose trans,
                          std::int64_t m,
                          std::int64_t n,
                          value_or_pointer<T> alpha,
                          const T *a,
                          std::int64_t lda,
                          std::int64_t stridea,
                          T *b,
                          std::int64_t ldb,
                          std::int64_t strideb,
                          const std::vector<sycl::event> &dependencies = {});
}
```

Input Parameters

queue
The queue where the routine will be executed.

trans
Specifies op(`A`), the transposition operation applied to matrix `A`. See *oneMKL defined datatypes* for more details.

m
Number of rows for the matrix `A`. Must be at least zero.

n
Number of columns for the matrix `A`. Must be at least zero.

alpha
Scaling factor for the matrix transposition or copy. See *Scalar Arguments in BLAS* for more details.

a
Pointer to input matrix `A`. Must have size at least `lda` * `n` for column-major and at least `lda` * `m` for row-major.

lda
Leading dimension of the matrix `A`. If matrices are stored using column major layout, `lda` is the number of elements in the array between adjacent columns of the matrix, and must be at least `stridea` * (`m`-1) + 1. If using row major layout, `lda` is the number of elements between adjacent rows of the matrix and must be at least `stridea` * (`n`-1) + 1.

stridea
The second stride of the matrix `A`. For column major layout, `stridea` is the number of elements in the array between adjacent rows of the matrix. For row major layout `stridea` is the number of elements between adjacent columns of the matrix. In both cases `stridea` must be at least 1.

b
Pointer to output matrix `B`.

	`trans` = `transpose::nontrans`	`trans` = `transpose::trans` or `trans` = `transpose::conjtrans`
Column major	`B` is `m` x `n` matrix. Size of array `b` must be at least `ldb` * `n`	`B` is `n` x `m` matrix. Size of array `b` must be at least `ldb` * `m`
Row major	`B` is `m` x `n` matrix. Size of array `b` must be at least `ldb` * `m`	`B` is `n` x `m` matrix. Size of array `b` must be at least `ldb` * `n`

ldb
The leading dimension of the matrix `B`. Must positive.

	`trans = transpose::nontrans`	`trans = transpose::trans` or `trans = transpose::conjtrans`
Column major	`ldb` must be at least `strideb` * (`m`-1) + 1.	`ldb` must be at least `strideb` * (`n`-1) + 1.
Row major	`ldb` must be at least `strideb` * (`n`-1) + 1.	`ldb` must be at least `strideb` * (`m`-1) + 1.

strideb
The second stride of the matrix `B`. For column major layout, `strideb` is the number of elements in the array between adjacent rows of the matrix. For row major layout, `strideb` is the number of elements between adjacent columns of the matrix. In both cases `strideb` must be at least 1.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

b
Pointer to output matrix `B` overwritten by `alpha` * op(`A`).

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS-like Extensions*

omatadd

Computes a sum of two general dense matrices, with optional transposes.

Description

The `omatadd` routine performs an out-of-place scaled matrix addition with optional transposes in the arguments. The operation is defined as:

$$C \leftarrow \alpha * op(A) + \beta * op(B)$$

where:

op(`X`) is one of op(`X`) = `X`, or op(`X`) = $\mathtt{X}^{T}$, or op(`X`) = $\mathtt{X}^{H}$

`alpha` and `beta` are scalars,

`A` and `B` are input matrices while `C` is an output matrix,

`C` is `m` x `n`,

`A` is `m` x `n` if the `op(A)` is not transposed or `n` by `m` if it is,

and `B` is `m` x `n` if the `op(B)` is not transposed or `n` by `m` if it is.

In general, `A`, `B`, and `C` should not overlap in memory, with the exception of the following in-place operations:

- `A` and `C` may point to the same memory if `op(A)` is non-transpose and `lda` = `ldc`;
- `B` and `C` may point to the same memory if `op(B)` is non-transpose and `ldb` = `ldc`.

`omatadd` supports the following precisions:

T
float
double
std::complex<float>
std::complex<double>

omatadd (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void omatadd(sycl::queue &queue,
                 oneapi::mkl::transpose transa,
                 oneapi::mkl::transpose transb,
                 std::int64_t m,
                 std::int64_t n,
                 T alpha,
                 sycl::buffer<T, 1> &a,
                 std::int64_t lda,
                 T beta,
                 sycl::buffer<T, 1> &b,
                 std::int64_t ldb,
```

(continues on next page)

(continued from previous page)

```
                sycl::buffer<T, 1> &c,
                std::int64_t ldc)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void omatadd(sycl::queue &queue,
                oneapi::mkl::transpose transa,
                oneapi::mkl::transpose transb,
                std::int64_t m,
                std::int64_t n,
                T alpha,
                sycl::buffer<T, 1> &a,
                std::int64_t lda,
                T beta,
                sycl::buffer<T, 1> &b,
                std::int64_t ldb,
                sycl::buffer<T, 1> &c,
                std::int64_t ldc)
}
```

Input Parameters

queue
The queue where the routine should be executed.

transa
Specifies op(`A`), the transposition operation applied to the matrix `A`. See *oneMKL defined datatypes* for more details.

transb
Specifies op(`B`), the transposition operation applied to the matrix `B`. See *oneMKL defined datatypes* for more details.

m
Number of rows for the result matrix `C`. Must be at least zero.

n
Number of columns for the result matrix `C`. Must be at least zero.

alpha
Scaling factor for the matrix `A`.

a
Buffer holding the input matrix `A`.

	`transa = transpose::nontrans`	`transa = transpose::trans` or `transa = transpose::conjtrans`
Column major	`A` is `m` x `n` matrix. Size of array `a` must be at least `lda * n`	`A` is `n` x `m` matrix. Size of array `a` must be at least `lda * m`
Row major	`A` is `m` x `n` matrix. Size of array `a` must be at least `lda * m`	`A` is `n` x `m` matrix. Size of array `a` must be at least `lda * n`

lda

The leading dimension of the matrix `A`. It must be positive.

	`transa` = `transpose::nontrans`	`transa` = `transpose::trans` or `transa` = `transpose::conjtrans`
Column major	`lda` must be at least `m`.	`lda` must be at least `n`.
Row major	`lda` must be at least `n`.	`lda` must be at least `m`.

beta

Scaling factor for the matrix `B`.

b

Buffer holding the input matrix `B`. Must have size at least:

	`transb` = `transpose::nontrans`	`transb` = `transpose::trans` or `transb` = `transpose::conjtrans`
Column major	`B` is `m` x `n` matrix. Size of array `b` must be at least `ldb` * `n`	`B` is `n` x `m` matrix. Size of array `b` must be at least `ldb` * `m`
Row major	`B` is `m` x `n` matrix. Size of array `b` must be at least `ldb` * `m`	`B` is `n` x `m` matrix. Size of array `b` must be at least `ldb` * `n`

ldb

The leading dimension of the `B` matrix. It must be positive.

	`transb` = `transpose::nontrans`	`transb` = `transpose::trans` or `transb` = `transpose::conjtrans`
Column major	`ldb` must be at least `m`.	`ldb` must be at least `n`.
Row major	`ldb` must be at least `n`.	`ldb` must be at least `m`.

c

Buffer holding the output matrix `C`.

Column major	`C` is `m` x `n` matrix. Size of array `c` must be at least `ldc` * `n`
Row major	`C` is `m` x `n` matrix. Size of array `c` must be at least `ldc` * `m`

ldc

Leading dimension of the C matrices. If matrices are stored using column major layout, `ldc` must be at least `m`. If matrices are stored using row major layout, `ldc` must be at least `n`. Must be positive.

Output Parameters

c
Output buffer overwritten by `alpha` * op(`A`) + `beta` * op(`B`).

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

omatadd (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event omatadd(sycl::queue &queue,
                        oneapi::mkl::transpose transa,
                        oneapi::mkl::transpose transb,
                        std::int64_t m,
                        std::int64_t n,
                        value_or_pointer<T> alpha,
                        const T *a,
                        std::int64_t lda,
                        value_or_pointer<T> beta,
                        const T *b,
                        std::int64_t ldb,
                        T *c,
                        std::int64_t ldc,
                        const std::vector<sycl::event> &dependencies = {});
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event omatadd(sycl::queue &queue,
                        oneapi::mkl::transpose transa,
                        oneapi::mkl::transpose transb,
                        std::int64_t m,
                        std::int64_t n,
                        value_or_pointer<T> alpha,
                        const T *a,
                        std::int64_t lda,
                        value_or_pointer<T> beta,
                        const T *b,
                        std::int64_t ldb,
```

(continues on next page)

(continued from previous page)

```
                        T *c,
                        std::int64_t ldc,
                        const std::vector<sycl::event> &dependencies = {});
}
```

Input Parameters

queue
The queue where the routine should be executed.

transa
Specifies op(A), the transposition operation applied to the matrix A. See *oneMKL defined datatypes* for more details.

transb
Specifies op(B), the transposition operation applied to the matrix B. See *oneMKL defined datatypes* for more details.

m
Number of rows for the result matrix C. Must be at least zero.

n
Number of columns for the result matrix C. Must be at least zero.

alpha
Scaling factor for the matrix A. See *Scalar Arguments in BLAS* for more details.

a
Array holding the input matrix A.

	transa = transpose::nontrans	transa = transpose::trans or transa = transpose::conjtrans
Column major	A is m x n matrix. Size of array a must be at least lda * n	A is n x m matrix. Size of array a must be at least lda * m
Row major	A is m x n matrix. Size of array a must be at least lda * m	A is n x m matrix. Size of array a must be at least lda * n

lda
The leading dimension of the matrix A. It must be positive.

	transa = transpose::nontrans	transa = transpose::trans or transa = transpose::conjtrans
Column major	lda must be at least m.	lda must be at least n.
Row major	lda must be at least n.	lda must be at least m.

beta
Scaling factor for the matrices B. See *Scalar Arguments in BLAS* for more details.

b
Array holding the input matrices B.

	transb = transpose::nontrans	transb = transpose::trans or transb = transpose::conjtrans
Column major	B is m x n matrix. Size of array b must be at least ldb * n	B is n x m matrix. Size of array b must be at least ldb * m
Row major	B is m x n matrix. Size of array b must be at least ldb * m	B is n x m matrix. Size of array b must be at least ldb * n

ldb

The leading dimension of the B matrix. It must be positive.

	transb = transpose::nontrans	transb = transpose::trans or transb = transpose::conjtrans
Column major	ldb must be at least m.	ldb must be at least n.
Row major	ldb must be at least n.	ldb must be at least m.

c

Array holding the output matrix C.

Column major	C is m x n matrix. Size of array c must be at least ldc * n
Row major	C is m x n matrix. Size of array c must be at least ldc * m

ldc

Leading dimension of the C matrix. If matrices are stored using column major layout, ldc must be at least m. If matrices are stored using row major layout, ldc must be at least n. Must be positive.

dependencies

List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

c

Output array, overwritten by alpha * op(A) + beta * op(B).

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS-like Extensions*

imatcopy_batch

Computes a group of in-place scaled matrix transpose or copy operations using general dense matrices.

Description

The `imatcopy_batch` routines perform a series of in-place scaled matrix copies or transpositions. They are batched versions of *imatcopy*, but the `imatcopy_batch` routines perform their operations with groups of matrices. Each group contains matrices with the same parameters.

There is a *strided API*, in which the matrices in a batch are a set distance away from each other in memory and in which all matrices share the same parameters (for example matrix size), and a more flexible *group API* where each group of matrices has the same parameters but the user may provide multiple groups that have different parameters. The group API argument structure is better suited to USM pointers than to `sycl::buffer` arguments, so we only specify it for USM inputs. The strided API works with both USM and buffer memory.

	strided API	group API
Buffer memory	supported	not supported
USM pointers	supported	supported

`imatcopy_batch` supports the following precisions:

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

imatcopy_batch (Buffer Version)

Description

The buffer version of `imatcopy_batch` supports only the strided API.

The operation for the strided API is defined as:

```
for i = 0 ... batch_size - 1
    C is a matrix at offset i * stride in matrix_array_in_out
    C := alpha * op(C)
end for
```

where:

op(X) is one of op(X) = X, or op(X) = X^T, or op(X) = X^H,

`alpha` is a scalar,

`C` is a matrix to be transformed in place,

and C is m x n.

The matrix_array_in_out buffer contains all the input matrices. The stride between matrices is given by the stride parameter. The total number of matrices in matrix_array_in_out is given by the batch_size parameter.

Strided API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void imatcopy_batch(sycl::queue &queue,
                        oneapi::mkl::transpose trans,
                        std::int64_t m,
                        std::int64_t n,
                        T alpha,
                        sycl::buffer<T, 1> &matrix_array_in_out,
                        std::int64_t ld_in,
                        std::int64_t ld_out,
                        std::int64_t stride,
                        std::int64_t batch_size);
}
```

```
namespace oneapi::mkl::blas::row_major {
    void imatcopy_batch(sycl::queue &queue,
                        oneapi::mkl::transpose trans,
                        std::int64_t m,
                        std::int64_t n,
                        T alpha,
                        sycl::buffer<T, 1> &matrix_array_in_out,
                        std::int64_t ld_in,
                        std::int64_t ld_out,
                        std::int64_t stride,
                        std::int64_t batch_size);
}
```

Input Parameters

queue
: The queue where the routine should be executed.

trans
: Specifies op(C), the transposition operation applied to the matrices C. See *oneMKL defined datatypes* for more details.

m
: Number of rows of each matrix C on input. Must be at least zero.

n
: Number of columns of each matrix C on input. Must be at least zero.

alpha
: Scaling factor for the matrix transpositions or copies.

matrix_array_in_out
: Buffer holding the input matrices C with size stride * batch_size.

ld_in
The leading dimension of the matrices `C` on input. It must be positive, and must be at least `m` if column major layout is used, and at least `n` if row-major layout is used.

ld_out
The leading dimension of the matrices `C` on output. It must be positive.

	`C` not transposed	`C` transposed
Column major	`ld_out` must be at least `m`.	`ld_out` must be at least `n`.
Row major	`ld_out` must be at least `n`.	`ld_out` must be at least `m`.

stride
Stride between different `C` matrices.

	`C` not transposed	`C` transposed
Column major	`stride` must be at least `max(ld_in*m, ld_out*m)`.	`stride` must be at least `max(ld_in*m, ld_out*n)`.
Row major	`stride` must be at least `max(ld_in*n, ld_out*n)`.	`stride` must be at least `max(ld_in*n, ld_out*m)`.

batch_size
Specifies the number of matrix transposition or copy operations to perform.

Output Parameters

matrix_array_in_out
Output buffer, overwritten by `batch_size` matrix copy or transposition operations of the form `alpha` * op(`C`).

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

imatcopy_batch (USM Version)

Description

The USM version of `imatcopy_batch` supports the group API and the strided API.

The operation for the group API is defined as:

```
idx = 0
for i = 0 ... group_count - 1
    m,n, alpha, ld_in, ld_out and group_size at position i in their respective arrays
    for j = 0 ... group_size - 1
        C is a matrix at position idx in matrix_array_in_out
        C := alpha * op(C)
        idx := idx + 1
    end for
end for
```

The operation for the strided API is defined as:

```
for i = 0 ... batch_size - 1
    C is a matrix at offset i * stride in matrix_array_in_out
    C := alpha * op(C)
end for
```

where:

op(X) is one of op(X) = X, or op(X) = X^T, or op(X) = X^H,

`alpha` is a scalar,

`C` is a matrix to be transformed in place,

and `C` is `m` x `n`.

For the group API, the matrices are given by arrays of pointers. `C` represents a matrix stored at the address pointed to by `matrix_array_in_out`. The number of entries in `matrix_array_in_out` is given by:

$$total\_batch\_count = \sum_{i=0}^{group\_count-1} group\_size[i]$$

For the strided API, the single array C contains all the matrices to be transformed in place. The locations of the individual matrices within the buffer or array are given by stride lengths, while the number of matrices is given by the `batch_size` parameter.

Group API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    event imatcopy_batch(sycl::queue &queue,
                         const oneapi::mkl::transpose *trans_array,
                         const std::int64_t *m_array,
                         const std::int64_t *n_array,
                         const T *alpha_array,
                         T **matrix_array_in_out,
```

(continues on next page)

(continued from previous page)

```
                          const std::int64_t *ld_in_array,
                          const std::int64_t *ld_out_array,
                          std::int64_t group_count,
                          const std::int64_t *groupsize,
                          const std::vector<sycl::event> &dependencies = {});
}
```

```
namespace oneapi::mkl::blas::row_major {
    event imatcopy_batch(sycl::queue &queue,
                          const oneapi::mkl::transpose *trans_array,
                          const std::int64_t *m_array,
                          const std::int64_t *n_array,
                          const T *alpha_array,
                          T **matrix_array_in_out,
                          const std::int64_t *ld_in_array,
                          const std::int64_t *ld_out_array,
                          std::int64_t group_count,
                          const std::int64_t *groupsize,
                          const std::vector<sycl::event> &dependencies = {});
}
```

Input Parameters

queue
The queue where the routine should be executed.

trans_array
Array of size `group_count`. Each element `i` in the array specifies `op(C)` the transposition operation applied to the matrices C.

m_array
Array of size `group_count` of number of rows of C on input. Each must be at least 0.

n_array
Array of size `group_count` of number of columns of C on input. Each must be at least 0.

alpha_array
Array of size `group_count` containing scaling factors for the matrix transpositions or copies.

matrix_array_in_out
Array of size `total_batch_count`, holding pointers to arrays used to store C matrices.

ld_in_array
Array of size `group_count`. The leading dimension of the matrix input `C`. If matrices are stored using column major layout, `ld_in_array[i]` must be at least `m_array[i]`. If matrices are stored using row major layout, `ld_in_array[i]` must be at least `n_array[i]`. Must be positive.

ld_out_array
Array of size `group_count`. The leading dimension of the output matrix `C`. Each entry `ld_out_array[i]` must be positive and at least:

- `m_array[i]` if column major layout is used and `C` is not transposed
- `m_array[i]` if row major layout is used and `C` is transposed
- `n_array[i]` otherwise

group_count
Number of groups. Must be at least 0.

group_size
Array of size `group_count`. The element `group_size[i]` is the number of matrices in the group `i`. Each element in `group_size` must be at least 0.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

matrix_array_in_out
Output array of pointers to `C` matrices, overwritten by `total_batch_count` matrix transpose or copy operations of the form `alpha*op(C)`.

Return Values

Output event to wait on to ensure computation is complete.

Strided API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event imatcopy_batch(sycl::queue &queue,
                               oneapi::mkl::transpose trans,
                               std::int64_t m,
                               std::int64_t n,
                               value_or_pointer<T> alpha,
                               const T *matrix_array_in_out,
                               std::int64_t ld_in,
                               std::int64_t ld_out,
                               std::int64_t stride,
                               std::int64_t batch_size,
                               const std::vector<sycl::event> &dependencies = {});
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event imatcopy_batch(sycl::queue &queue,
                               oneapi::mkl::transpose trans,
                               std::int64_t m,
                               std::int64_t n,
                               value_or_pointer<T> alpha,
                               const T *matrix_array_in_out,
                               std::int64_t ld_in,
                               std::int64_t ld_out,
                               std::int64_t stride,
                               std::int64_t batch_size,
                               const std::vector<sycl::event> &dependencies = {});
```

Input Parameters

queue
The queue where the routine should be executed.

trans
Specifies `op(C)`, the transposition operation applied to the matrices C.

m
Number of rows for each matrix `C` on input. Must be at least 0.

n
Number of columns for each matrix `C` on input. Must be at least 0.

alpha
Scaling factor for the matrix transpose or copy operation. See *Scalar Arguments in BLAS* for more details.

matrix_array_in_out
Array holding the matrices `C`. Must have size at least `stride*batch_size`.

ld_in
Leading dimension of the `C` matrices on input. If matrices are stored using column major layout, `ld_in` must be at least `m`. If matrices are stored using row major layout, `ld_in` must be at least `n`. Must be positive.

ld_out
Leading dimension of the `C` matrices on output. If matrices are stored using column major layout, `ld_out` must be at least `m` if `C` is not transposed or `n` if `C` is transposed. If matrices are stored using row major layout, `ld_out` must be at least `n` if `C` is not transposed or at least `m` if `C` is transposed. Must be positive.

stride
Stride between different `C` matrices within `matrix_array_in_out`.

	`C` not transposed	`C` transposed
Column major	`stride` must be at least `max(ld_in*m, ld_out*m)`.	`stride` must be at least `max(ld_in*m, ld_out*n)`.
Row major	`stride` must be at least `max(ld_in*n, ld_out*n)`.	`stride` must be at least `max(ld_in*n, ld_out*m)`.

batch_size
Specifies the number of matrices to transpose or copy.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

matrix_array_in_out
Output array, overwritten by `batch_size` matrix transposition or copy operations of the form `alpha*op(C)`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS-like Extensions*

omatcopy_batch

Computes a group of out-of-place scaled matrix transpose or copy operations using general dense matrices.

Description

The `omatcopy_batch` routines perform a series of out-of-place scaled matrix copies or transpositions. They are batched versions of *omatcopy*, but the `omatcopy_batch` routines perform their operations with groups of matrices. Each group contains matrices with the same parameters.

There is a *strided API*, in which the matrices in a batch are a set distance away from each other in memory and in which all matrices share the same parameters (for example matrix size), and a more flexible *group API* where each group of matrices has the same parameters but the user may provide multiple groups that have different parameters. The group API argument structure is better suited to USM pointers than to `sycl::buffer` arguments, so we only specify it for USM inputs. The strided API works with both USM and buffer memory.

	strided API	group API
Buffer memory	supported	not supported
USM pointers	supported	supported

`omatcopy_batch` supports the following precisions:

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

omatcopy_batch (Buffer Version)

Description

The buffer version of `omatadd_batch` supports only the strided API.

The operation for the strided API is defined as:

```
for i = 0 ... batch_size - 1
    A and B are matrices at offset i * stridea in a and i * strideb in b
    B := alpha * op(A)
end for
```

where:

op(X) is one of op(X) = X, or op(X) = X^T, or op(X) = X^H,

`alpha` is a scalar,

`A` and `B` are input and output matrices,

`A` is `m` x `n`,

and `B` is `m` x `n` if the matrix is not transposed or `n` by `m` if it is.

The `a` buffer contains all the input matrices while the `b` buffer contains all the output matrices. The locations of the individual matrices within the buffer are given by the `stride_a` and `stride_b` parameters, while the total number of matrices in each buffer is given by the `batch_size` parameter.

Strided API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void omatcopy_batch(sycl::queue &queue,
                        oneapi::mkl::transpose trans,
                        std::int64_t m,
                        std::int64_t n,
                        T alpha,
                        sycl::buffer<T, 1> &a,
                        std::int64_t lda,
                        std::int64_t stride_a,
                        sycl::buffer<T, 1> &b,
                        std::int64_t ldb,
                        std::int64_t stride_b,
                        std::int64_t batch_size);
}
```

```
namespace oneapi::mkl::blas::row_major {
    void omatcopy_batch(sycl::queue &queue,
                        oneapi::mkl::transpose trans,
                        std::int64_t m,
                        std::int64_t n,
                        T alpha,
                        sycl::buffer<T, 1> &a,
                        std::int64_t lda,
```

(continues on next page)

(continued from previous page)

```
                          std::int64_t stride_a,
                          sycl::buffer<T, 1> &b,
                          std::int64_t ldb,
                          std::int64_t stride_b,
                          std::int64_t batch_size);
}
```

Input Parameters

queue
The queue where the routine should be executed.

trans
Specifies op(`A`), the transposition operation applied to the matrices `A`. See *oneMKL defined datatypes* for more details.

m
Number of rows for each matrix `A`. Must be at least zero.

n
Number of columns for each matrix `A`. Must be at least zero.

alpha
Scaling factor for the matrix transposition or copy operations.

a
Buffer holding the input matrices `A` with size `stride_a` * `batch_size`.

lda
The leading dimension of the matrices `A`. It must be positive, and must be at least `m` if column major layout is used, and at least `n` if row-major layout is used.

stride_a
Stride between the different `A` matrices. If matrices are stored using column major layout, `stride_a` must be at least `lda*n`. If matrices are stored using row major layout, `stride_a` must be at least `lda*m`.

b
Buffer holding the output matrices `B` with size `stride_b` * `batch_size`.

ldb
The leading dimension of the matrices `B`. It must be positive.

	`B` not transposed	`B` transposed
Column major	`ldb` must be at least `m`.	`ldb` must be at least `n`.
Row major	`ldb` must be at least `n`.	`ldb` must be at least `m`.

stride_b
Stride between different `B` matrices.

	`B` not transposed	`B` transposed
Column major	`stride_b` must be at least `ldb` x `n`.	`stride_b` must be at least `ldb` x `m`.
Row major	`stride_b` must be at least `ldb` x `m`.	`stride_b` must be at least `ldb` x `n`.

batch_size
Specifies the number of matrix transposition or copy operations to perform.

Output Parameters

b
Output buffer, overwritten by `batch_size` matrix copy or transposition operations of the form `alpha` * op(`A`).

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

omatcopy_batch (USM Version)

Description

The USM version of `omatcopy_batch` supports the group API and the strided API.

The operation for the group API is defined as:

```
idx = 0
for i = 0 ... group_count - 1
    m, n, alpha, lda, ldb and group_size at position i in their respective arrays
    for j = 0 ... group_size - 1
        A and B are matrices at position idx in their respective arrays
        B := alpha * op(A)
        idx := idx + 1
    end for
end for
```

The operation for the strided API is defined as:

```
for i = 0 ... batch_size - 1
    A and B are matrices at offset i * stridea in a and i * strideb in b
    B := alpha * op(A)
end for
```

where:

op(X) is one of op(X) = X, or op(X) = X^T, or op(X) = X^H,

`alpha` is a scalar,

`A` and `B` are input and output matrices,

`A` is `m` x `n`,

and B is m x n if the matrix is not transposed or n by m if it is.

For the group API, the matrices are given by arrays of pointers. A and B represent matrices stored at addresses pointed to by a_array and b_array respectively. The number of entries in a_array and b_array is given by:

$$total\_batch\_count = \sum_{i=0}^{group\_count-1} group\_size[i]$$

For the strided API, the single input array contains all the input matrices, and the single output array contains all the output matrices. The locations of the individual matrices within the array are given by stride lengths, while the number of matrices is given by the batch_size parameter.

Group API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event omatcopy_batch(sycl::queue &queue,
                               const oneapi::mkl::transpose *trans_array,
                               const std::int64_t *m_array,
                               const std::int64_t *n_array,
                               const T *alpha_array,
                               const T **a_array,
                               const std::int64_t *lda_array,
                               T **b_array,
                               const std::int64_t *ldb_array,
                               std::int64_t group_count,
                               const std::int64_t *groupsize,
                               const std::vector<sycl::event> &dependencies = {});
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event omatcopy_batch(sycl::queue &queue,
                               const oneapi::mkl::transpose *trans_array,
                               const std::int64_t *m_array,
                               const std::int64_t *n_array,
                               const T *alpha_array,
                               const T **a_array,
                               const std::int64_t *lda_array,
                               T **b_array,
                               const std::int64_t *ldb_array,
                               std::int64_t group_count,
                               const std::int64_t *groupsize,
                               const std::vector<sycl::event> &dependencies = {});
}
```

Input Parameters

queue
The queue where the routine should be executed.

trans_array
Array of size `group_count`. Each element `i` in the array specifies `op(A)` the transposition operation applied to the matrices A.

m_array
Array of size `group_count` of number of rows of A. Each must be at least 0.

n_array
Array of size `group_count` of number of columns of A. Each must be at least 0.

alpha_array
Array of size `group_count` containing scaling factors for the matrix transpositions or copies.

a_array
Array of size `total_batch_count`, holding pointers to arrays used to store A matrices.

lda_array
Array of size `group_count` of leading dimension of the A matrices. If matrices are stored using column major layout, `lda_array[i]` must be at least `m_array[i]`. If matrices are stored using row major layout, `lda_array[i]` must be at least `n_array[i]`. Each must be positive.

b_array
Array of size `total_batch_count` of pointers used to store B matrices. The array allocated for each B matrix of the group `i` must be of size at least:

	`B` not transposed	`B` transposed
Column major	`ldb_array[i]` x `n_array[i]`	`ldb_array[i]` x `m_array[i]`
Row major	`ldb_array[i]` x `m_array[i]`	`ldb_array[i]` x `n_array[i]`

ldb_array
Array of size `group_count`. The leading dimension of the output matrix B. Each entry `ldb_array[i]` must be positive and at least:

	`B` not transposed	`B` transposed
Column major	`ldb[i]` must be at least `m_array[i]`.	`ldb[i]` must be at least `n_array[i]`.
Row major	`ldb[i]` must be at least `n_array[i]`.	`ldb[i]` must be at least `m_array[i]`.

group_count
Number of groups. Must be at least 0.

group_size
Array of size `group_count`. The element `group_size[i]` is the number of matrices in the group `i`. Each element in `group_size` must be at least 0.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

b_array
Output array of pointers to B matrices, overwritten by `total_batch_count` matrix transpose or copy operations of the form `alpha*op(A)`.

Return Values

Output event to wait on to ensure computation is complete.

Strided API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    event omatcopy_batch(queue &queue,
        transpose trans,
        std::int64_t m,
        std::int64_t n,
        value_or_pointer<T> alpha,
        const T *a,
        std::int64_t lda,
        std::int64_t stride_a,
        T *b,
        std::int64_t ldb,
        std::int64_t stride_b,
        std::int64_t batch_size,
        const std::vector<sycl::event> &dependencies = {});
}
```

```
namespace oneapi::mkl::blas::row_major {
    event omatcopy_batch(queue &queue,
        transpose trans,
        std::int64_t m,
        std::int64_t n,
        value_or_pointer<T> alpha,
        const T *a,
        std::int64_t lda,
        std::int64_t stride_a,
        T *b,
        std::int64_t ldb,
        std::int64_t stride_b,
        std::int64_t batch_size,
        const vector_class<event> &dependencies = {});
}
```

Input Parameters

queue
The queue where the routine will be executed.

trans
Specifies `op(A)`, the transposition operation applied to the matrices A.

m
Number of rows for each matrix A. Must be at least 0.

n
Number of columns for each matrix B. Must be at least 0.

alpha
Scaling factor for the matrix transpose or copy operation. See *Scalar Arguments in BLAS* for more details.

a
Array holding the matrices A. Must have size at least `stride_a*batch_size`.

lda
Leading dimension of the A matrices. If matrices are stored using column major layout, `lda` must be at least `m`. If matrices are stored using row major layout, `lda` must be at least `n`. Must be positive.

stride_a
Stride between the different A matrices. If matrices are stored using column major layout, `stride_a` must be at least `lda*n`. If matrices are stored using row major layout, `stride_a` must be at least `lda*m`.

b
Array holding the matrices B. Must have size at least `stride_b*batch_size`.

ldb
Leading dimension of the B matrices. Must be positive.

	`B` not transposed	`B` transposed
Column major	`ldb` must be at least `m`.	`ldb` must be at least `n`.
Row major	`ldb` must be at least `n`.	`ldb` must be at least `m`.

stride_b
Stride between different `B` matrices.

	`B` not transposed	`B` transposed
Column major	`stride_b` must be at least `ldb` x `n`.	`stride_b` must be at least `ldb` x `m`.
Row major	`stride_b` must be at least `ldb` x `m`.	`stride_b` must be at least `ldb` x `n`.

batch_size
Specifies the number of matrices to transpose or copy.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

b
Output array, overwritten by `batch_size` matrix transposition or copy operations of the form `alpha*op(A)`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS-like Extensions*

omatadd_batch

Computes a group of out-of-place scaled matrix additions using general dense matrices.

Description

The `omatadd_batch` routines perform a series of out-of-place scaled matrix additions. They are batched versions of *omatadd*, but the `omatadd_batch` routines perform their operations with groups of matrices. Each group contains matrices with the same parameters.

There is a *strided API*, in which the matrices in a batch are a set distance away from each other in memory and in which all matrices share the same parameters (for example matrix size), and a more flexible *group API* where each group of matrices has the same parameters but the user may provide multiple groups that have different parameters. The group API argument structure is better suited to USM pointers than to `sycl::buffer` arguments, so we only specify it for USM inputs. The strided API works with both USM and buffer memory.

	strided API	group API
Buffer memory	supported	not supported
USM pointers	supported	supported

`omatadd_batch` supports the following precisions:

T
float
double
std::complex<float>
std::complex<double>

omatadd_batch (Buffer Version)

Description

The buffer version of `omatcopy_batch` supports only the strided API.

The operation of `omatadd_batch` is defined as:

```
for i = 0 ... batch_size - 1
    A is a matrix at offset i * stridea in a
    B is a matrix at offset i * strideb in b
    C is a matrix at offset i * stridec in c
    C := alpha * op(A) + beta * op(B)
end for
```

where:

op(X) is one of op(X) = X, or op(X) = X^T, or op(X) = X^H,

`alpha` and `beta` are scalars,

`A` and `B` are input matrices while `C` is an output matrix,

`C` is `m` x `n`,

`A` is `m` x `n` if the `op(A)` is not transposed or `n` by `m` if it is,

and `B` is `m` x `n` if the `op(B)` is not transposed or `n` by `m` if it is.

The `a` and `b` buffers contain all the input matrices while the `c` buffer contains all the output matrices. The locations of the individual matrices within the buffer are given by the `stride_a`, `stride_b`, and `stride_c` parameters, while the total number of matrices in each buffer is given by the `batch_size` parameter.

In general, the `a`, `b`, and `c` buffers should not overlap in memory, with the exception of the following in-place operations:

- `a` and `c` may point to the same memory if `op(A)` is non-transpose and all the `A` matrices have the same parameters as all the respective `C` matrices;
- `b` and `c` may point to the same memory if `op(B)` is non-transpose and all the `B` matrices have the same parameters as all the respective `C` matrices.

Strided API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void omatadd_batch(sycl::queue &queue,
                       oneapi::mkl::transpose transa,
                       oneapi::mkl::transpose transb,
                       std::int64_t m,
                       std::int64_t n,
                       T alpha,
                       sycl::buffer<T, 1> &a,
                       std::int64_t lda,
                       std::int64_t stride_a,
                       T beta,
                       sycl::buffer<T, 1> &b,
                       std::int64_t ldb,
                       std::int64_t stride_b,
                       sycl::buffer<T, 1> &c,
                       std::int64_t ldc,
                       std::int64_t stride_c,
                       std::int64_t batch_size);
}
```

```
namespace oneapi::mkl::blas::row_major {
    void omatadd_batch(sycl::queue &queue,
                       oneapi::mkl::transpose transa,
                       oneapi::mkl::transpose transb,
                       std::int64_t m,
                       std::int64_t n,
                       T alpha,
                       sycl::buffer<T, 1> &a,
                       std::int64_t lda,
                       std::int64_t stride_a,
                       T beta,
                       sycl::buffer<T, 1> &b,
                       std::int64_t ldb,
                       std::int64_t stride_b,
                       sycl::buffer<T, 1> &c,
                       std::int64_t ldc,
                       std::int64_t stride_c,
                       std::int64_t batch_size);
}
```

Input Parameters

queue
: The queue where the routine should be executed.

transa
: Specifies op(`A`), the transposition operation applied to the matrices `A`. See *oneMKL defined datatypes* for more details.

transb
: Specifies op(`B`), the transposition operation applied to the matrices `B`. See *oneMKL defined datatypes* for more

details.

m
Number of rows for the result matrix `C`. Must be at least zero.

n
Number of columns for the result matrix `C`. Must be at least zero.

alpha
Scaling factor for the matrices `A`.

a
Buffer holding the input matrices `A`. Must have size at least `stride_a` * `batch_size`.

lda
The leading dimension of the matrices `A`. It must be positive.

	`A` not transposed	`A` transposed
Column major	`lda` must be at least `m`.	`lda` must be at least `n`.
Row major	`lda` must be at least `n`.	`lda` must be at least `m`.

stride_a
Stride between the different `A` matrices within the buffer.

	`A` not transposed	`A` transposed
Column major	`stride_a` must be at least `lda*n`.	`stride_a` must be at least `lda*m`.
Row major	`stride_a` must be at least `lda*m`.	`stride_a` must be at least `lda*n`.

beta
Scaling factor for the matrices `B`.

b
Buffer holding the input matrices `B`. Must have size at least `stride_b` * `batch_size`.

ldb
The leading dimension of the `B` matrices. It must be positive.

	`B` not transposed	`B` transposed
Column major	`ldb` must be at least `m`.	`ldb` must be at least `n`.
Row major	`ldb` must be at least `n`.	`ldb` must be at least `m`.

stride_b
Stride between different `B` matrices.

	`B` not transposed	`B` transposed
Column major	`stride_b` must be at least `ldb` x `n`.	`stride_b` must be at least `ldb` x `m`.
Row major	`stride_b` must be at least `ldb` x `m`.	`stride_b` must be at least `ldb` x `n`.

c
Buffer holding the output matrices `C`. Must have size at least `stride_c` * `batch_size`.

ldc
: Leading dimension of the `C` matrices. If matrices are stored using column major layout, `ldc` must be at least `m`. If matrices are stored using row major layout, `ldc` must be at least `n`. Must be positive.

stride_c
: Stride between the different `C` matrices. If matrices are stored using column major layout, `stride_c` must be at least `ldc*n`. If matrices are stored using row major layout, `stride_c` must be at least `ldc*m`.

batch_size
: Specifies the number of matrix transposition or copy operations to perform.

Output Parameters

c
: Output buffer, overwritten by `batch_size` matrix addition operations of the form `alpha*op(A) + beta*op(B)`. Must have size at least `stride_c*batch_size`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

omatadd_batch (USM Version)

Description

The USM version of `omatadd_batch` supports the group API and the strided API.

The operation for the group API is defined as:

```
idx = 0
for i = 0 ... group_count - 1
    m, n, alpha, beta, lda, ldb, ldc and group_size at position i in their respective␣
↪arrays
    for j = 0 ... group_size - 1
        A, B and C are matrices at position idx in their respective arrays
        C := alpha * op(A) + beta * op(B)
        idx := idx + 1
    end for
end for
```

The operation for the strided API is defined as:

```
for i = 0 ... batch_size - 1
    A is a matrix at offset i * stridea in a
    B is a matrix at offset i * strideb in b
    C is a matrix at offset i * stridec in c
    C := alpha * op(A) + beta * op(B)
end for
```

where:

op(X) is one of op(X) = X, or op(X) = X^T, or op(X) = X^H,

`alpha` and `beta` are scalars,

`A` and `B` are input matrices while `C` is an output matrix,

`C` is `m` x `n`,

`A` is `m` x `n` if the `op(A)` is not transposed or `n` by `m` if it is,

and `B` is `m` x `n` if the `op(B)` is not transposed or `n` by `m` if it is.

For the group API, the matrices are given by arrays of pointers. `A`, `B`, and `C` represent matrices stored at addresses pointed to by `a_array`, `b_array`, and `c_array` respectively. The number of entries in `a_array`, `b_array`, and `c_array` is given by:

$$total\_batch\_count = \sum_{i=0}^{group\_count-1} group\_size[i]$$

For the strided API, the `a` and `b` arrays contain all the input matrices while the `c` array contains all the output matrices. The locations of the individual matrices within the array are given by the `stride_a`, `stride_b`, and `stride_c` parameters, while the total number of matrices in each array is given by the `batch_size` parameter.

In general, the batches of matrices indicated by `a`, `b`, and `c` should not overlap in memory, with the exception of the the following in-place operations:

- `a` and `c` may point to the same memory if `op(A)` is non-transpose and all the `A` matrices have identical parameters as all the respective `C` matrices;
- `b` and `c` may point to the same memory if `op(B)` is non-transpose and all the the `B` matrices have identical parameters as all the respective `C` matrices.

Group API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event omatadd_batch(sycl::queue &queue,
                              const oneapi::mkl::transpose *transa_array,
                              const oneapi::mkl::transpose *transb_array,
                              const std::int64_t *m_array,
                              const std::int64_t *n_array,
                              const T *alpha_array,
                              const T **a_array,
                              const std::int64_t *lda_array,
                              const T *beta_array,
                              const T **b_array,
                              const std::int64_t *ldb_array,
```

(continues on next page)

(continued from previous page)

```
                                const T **c_array,
                                const std::int64_t *ldc_array,
                                std::int64_t group_count,
                                const std::int64_t *groupsize,
                                const std::vector<sycl::event> &dependencies = {});
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event omatadd_batch(sycl::queue &queue,
                                const oneapi::mkl::transpose *transa_array,
                                const oneapi::mkl::transpose *transb_array,
                                const std::int64_t *m_array,
                                const std::int64_t *n_array,
                                const T *alpha_array,
                                const T **a_array,
                                const std::int64_t *lda_array,
                                const T *beta_array,
                                const T **b_array,
                                const std::int64_t *ldb_array,
                                const T **c_array,
                                const std::int64_t *ldc_array,
                                std::int64_t group_count,
                                const std::int64_t *groupsize,
                                const std::vector<sycl::event> &dependencies = {});
}
```

Input Parameters

queue
: The queue where the routine should be executed.

transa_array
: Array of size `group_count`. Each element `i` in the array specifies `op(A)` the transposition operation applied to the matrices A.

transb_array
: Array of size `group_count`. Each element `i` in the array specifies `op(B)` the transposition operation applied to the matrices B.

m_array
: Array of size `group_count` of number of rows of `C`. Each must be at least 0.

n_array
: Array of size `group_count` of number of columns of `C`. Each must be at least 0.

alpha_array
: Array of size `group_count` containing scaling factors for the matrices `A`.

a_array
: Array of size `total_batch_count`, holding pointers to arrays used to store `A` matrices. The array allocated for each `A` matrix of the group `i` must be of size at least:

	transa[i] = transpose::nontrans		transa[i] = transpose::trans or transa[i] = transpose::conjtrans
Column major	lda_array[i]	*	lda_array[i] * m_array[i]
Row major	lda_array[i]	*	lda_array[i] * n_array[i]

lda_array
Array of size group_count of leading dimension of the A matrices. All must be positive and satisfy:

	transa[i] = transpose::nontrans	transa[i] = transpose::trans or transa = transpose::conjtrans
Column major	lda_array[i] must be at least m_array[i].	lda_array[i] must be at least n_array[i].
Row major	lda_array[i] must be at least n_array[i].	lda_array[i] must be at least m_array[i].

beta_array
Array of size group_count containing scaling factors for the matrices B.

b_array
Array of size total_batch_count of pointers used to store the B matrices. The array allocated for each B matrix of the group i must be of size at least:

	transb[i] = transpose::nontrans		transb[i] = transpose::trans or transb[i] = transpose::conjtrans
Column major	ldb_array[i]	*	ldb_array[i] * m_array[i]
Row major	ldb_array[i]	*	ldb_array[i] * n_array[i]

ldb_array
Array of size group_count. The leading dimension of B matrices. All must be positive and satisfy:

	transb[i] = transpose::nontrans	transb[i] = transpose::trans or transb[i] = transpose::conjtrans
Column major	ldb_array[i] must be at least m_array[i].	ldb_array[i] must be at least n_array[i].
Row major	ldb_array[i] must be at least n_array[i].	ldb_array[i] must be at least m_array[i].

c_array
Array of size total_batch_count of pointers used to store the C output matrices. The array allocated for each C matrix of the group i must be of size at least:

Column major	ldc_array[i] * n_array[i]
Row major	ldc_array[i] * m_array[i]

ldc_array
: Array of size `group_count`. The leading dimension of the `C` matrices. If matrices are stored using column major layout, `ldc_array[i]` must be at least `m_array[i]`. If matrices are stored using row major layout, `ldc_array[i]` must be at least `n_array[i]`. All entries must be positive.

group_count
: Number of groups. Must be at least 0.

group_size
: Array of size `group_count`. The element `group_size[i]` is the number of matrices in the group `i`. Each element in `group_size` must be at least 0.

dependencies
: List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

c_array
: Output array of pointers to C matrices, overwritten by `total_batch_count` matrix addition operations of the form `alpha*op(A) + beta*op(B)`.

Return Values

Output event to wait on to ensure computation is complete.

Strided API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event omatadd_batch(sycl::queue &queue,
                              oneapi::mkl::transpose transa,
                              oneapi::mkl::transpose transb,
                              std::int64_t m,
                              std::int64_t n,
                              value_or_pointer<T> alpha,
                              const T *a,
                              std::int64_t lda,
                              std::int64_t stride_a,
                              value_or_pointer<T> beta,
                              T *b,
                              std::int64_t ldb,
                              std::int64_t stride_b,
                              T *c,
                              std::int64_t ldc,
                              std::int64_t stride_c,
                              std::int64_t batch_size,
                              const std::vector<sycl::event> &dependencies = {});
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event omatadd_batch(sycl::queue &queue,
```

(continues on next page)

(continued from previous page)

```
                                oneapi::mkl::transpose transa,
                                oneapi::mkl::transpose transb,
                                std::int64_t m,
                                std::int64_t n,
                                value_or_pointer<T> alpha,
                                const T *a,
                                std::int64_t lda,
                                std::int64_t stride_a,
                                value_or_pointer<T> beta,
                                T *b,
                                std::int64_t ldb,
                                std::int64_t stride_b,
                                T *c,
                                std::int64_t ldc,
                                std::int64_t stride_c,
                                std::int64_t batch_size,
                                const std::vector<sycl::event> &dependencies = {});
}
```

Input Parameters

queue
The queue where the routine should be executed.

transa
Specifies op(`A`), the transposition operation applied to the matrices `A`. See *oneMKL defined datatypes* for more details.

transb
Specifies op(`B`), the transposition operation applied to the matrices `B`. See *oneMKL defined datatypes* for more details.

m
Number of rows for the result matrix `C`. Must be at least zero.

n
Number of columns for the result matrix `C`. Must be at least zero.

alpha
Scaling factor for the matrices `A`. See *Scalar Arguments in BLAS* for more details.

a
Array holding the input matrices `A`. Must have size at least `stride_a` * `batch_size`.

lda
The leading dimension of the matrices `A`. It must be positive.

	`A` not transposed	`A` transposed
Column major	`lda` must be at least `m`.	`lda` must be at least `n`.
Row major	`lda` must be at least `n`.	`lda` must be at least `m`.

stride_a
Stride between the different `A` matrices within the array.

	A not transposed	A transposed
Column major	stride_a must be at least lda*n.	stride_a must be at least lda*m.
Row major	stride_a must be at least lda*m.	stride_a must be at least lda*n.

beta
Scaling factor for the matrices B. See *Scalar Arguments in BLAS* for more details.

b
Array holding the input matrices B. Must have size at least stride_b * batch_size.

ldb
The leading dimension of the B matrices. It must be positive.

	B not transposed	B transposed
Column major	ldb must be at least m.	ldb must be at least n.
Row major	ldb must be at least n.	ldb must be at least m.

stride_b
Stride between different B matrices.

	B not transposed	B transposed
Column major	stride_b must be at least ldb x n.	stride_b must be at least ldb x m.
Row major	stride_b must be at least ldb x m.	stride_b must be at least ldb x n.

c
Array holding the output matrices C. Must have size at least stride_c * batch_size.

ldc
Leading dimension of the C matrices. If matrices are stored using column major layout, ldc must be at least m. If matrices are stored using row major layout, ldc must be at least n. Must be positive.

stride_c
Stride between the different C matrices. If matrices are stored using column major layout, stride_c must be at least ldc*n. If matrices are stored using row major layout, stride_c must be at least ldc*m.

batch_size
Specifies the number of matrix transposition or copy operations to perform.

dependencies
List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

c
Output array, overwritten by batch_size matrix addition operations of the form alpha*op(A) + beta*op(B). Must have size at least stride_c*batch_size.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS-like Extensions*

Parent topic: *BLAS Routines*

Parent topic: *Dense Linear Algebra*

LAPACK Routines

oneMKL provides a DPC++ interface to select routines from the Linear Algebra PACKage (LAPACK), as well as several LAPACK-like extension routines. LAPACK routines require column major layout of matrices.

LAPACK Linear Equation Routines

LAPACK Linear Equation routines are used for factoring a matrix, solving a system of linear equations, solving linear least squares problems, and inverting a matrix. The following table lists the LAPACK Linear Equation routine groups.

Routines	Scratchpad Size Routines	Description
geqrf	*geqrf_scratchpad_*	Computes the QR factorization of a general m-by-n matrix.
gerqf	*gerqf_scratchpad_*	Computes the RQ factorization of a general m-by-n matrix.
getrf	*getrf_scratchpad_*	Computes the LU factorization of a general m-by-n matrix.
getri	*getri_scratchpad_*	Computes the inverse of an LU-factored general matrix.
getrs	*getrs_scratchpad_*	Solves a system of linear equations with an LU-factored square coefficient matrix, with multiple right-hand sides.
hetrf	*hetrf_scratchpad_*	Computes the Bunch-Kaufman factorization of a complex Hermitian matrix.
orgqr	*orgqr_scratchpad*	Generates the real orthogonal matrix Q of the QR factorization formed by geqrf.
ormqr	*ormqr_scratchpad_*	Multiplies a real matrix by the orthogonal matrix Q of the QR factorization formed by geqrf.
ormrq	*ormrq_scratchpa*	Multiplies a real matrix by the orthogonal matrix Q of the RQ factorization formed by gerqf.
potrf	*potrf_scratchpad_*	Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite matrix.
potri	*potri_scratchpad_*	Computes the inverse of a Cholesky-factored symmetric (Hermitian) positive-definite matrix.
potrs	*potrs_scratchpad_*	Solves a system of linear equations with a Cholesky-factored symmetric (Hermitian) positive-definite coefficient matrix, with multiple right-hand sides.
sytrf	*sytrf_scratchpad_*	Computes the Bunch-Kaufman factorization of a symmetric matrix.
trtrs	*trtrs_scratchpad_*	Solves a system of linear equations with a triangular coefficient matrix, with multiple right-hand sides.
ungqr	*ungqr_scratchpa*	Generates the complex unitary matrix Q of the QR factorization formed by geqrf.
unmqr	*unmqr_scratchpad_*	Multiplies a complex matrix by the unitary matrix Q of the QR factorization formed by geqrf.
unmrq	*unmrq_scratchpad_*	Multiplies a complex matrix by the unitary matrix Q of the RQ factorization formed by gerqf.

geqrf

Computes the QR factorization of a general $m \times n$ matrix.

Description

`geqrf` supports the following precisions:

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

The routine forms the QR factorization of a general $m \times n$ matrix A. No pivoting is performed.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ elementary reflectors. Routines are provided to work with Q in this representation.

geqrf (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void geqrf(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, cl::sycl::buffer<T,
↪1> &a, std::int64_t lda, cl::sycl::buffer<T,1> &tau, cl::sycl::buffer<T,1> &scratchpad,
↪ std::int64_t scratchpad_size)
}
```

Input Parameters

queue
The queue where the routine should be executed.

m
The number of rows in the matrix A ($0 \leq m$).

n
The number of columns in A ($0 \leq n$).

a
Buffer holding input matrix A. Must have size at least lda $\cdot n$.

lda
The leading dimension of A; at least $\max(1, m)$.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *geqrf_scratchpad_size* function.

Output Parameters

a
Output buffer, overwritten by the factorization data as follows:

The elements on and above the diagonal of the array contain the $\min(m, n) \times n$ upper trapezoidal matrix R (R is upper triangular if $m \geq n$); the elements below the diagonal, with the array tau, represent the orthogonal matrix Q as a product of $\min(m, n)$ elementary reflectors.

tau
Output buffer, size at least $\max(1, \min(m, n))$. Contains scalars that define elementary reflectors for the matrix Q in its decomposition in a product of elementary reflectors.

scratchpad
Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If `info=-i`, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

geqrf (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event geqrf(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, T *a,␣
↪std::int64_t lda, T *tau, T *scratchpad, std::int64_t scratchpad_size, const␣
↪std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

m
The number of rows in the matrix A ($0 \leq m$).

n
The number of columns in A ($0 \leq n$).

a
Pointer to memory holding input matrix A. Must have size at least lda $\cdot n$.

lda
The leading dimension of A; at least $\max(1, m)$.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *geqrf_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a

Overwritten by the factorization data as follows:

The elements on and above the diagonal of the array contain the $\min(m, n) \times n$ upper trapezoidal matrix R (R is upper triangular if $m \geq n$); the elements below the diagonal, with the array tau, represent the orthogonal matrix Q as a product of $\min(m, n)$ elementary reflectors.

tau

Array, size at least $\max(1, \min(m, n))$. Contains scalars that define elementary reflectors for the matrix Q in its decomposition in a product of elementary reflectors.

scratchpad

Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If `info=-i`, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

geqrf_scratchpad_size

Computes size of scratchpad memory required for *geqrf* function.

Description

`geqrf_scratchpad_size` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

Computes the number of elements of type `T` the scratchpad memory to be passed to *geqrf* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t geqrf_scratchpad_size(cl::sycl::queue &queue, std::int64_t m, std::int64_
↪t n, std::int64_t lda)
}
```

Input Parameters

queue
: Device queue where calculations by *geqrf* function will be performed.

m
: The number of rows in the matrix A ($0 \leq m$).

n
: The number of columns in the matrix A ($0 \leq n$).

lda
: The leading dimension of `a`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

: Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *geqrf* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

gerqf

Computes the RQ factorization of a general $m \times n$ matrix.

Description

`gerqf` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

The routine forms the RQ factorization of a general $m \times n$ matrix A. No pivoting is performed. The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ elementary reflectors. Routines are provided to work with Q in this representation

gerqf (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void gerqf(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, cl::sycl::buffer<T>␣
↪&a, std::int64_t lda, cl::sycl::buffer<T> &tau, cl::sycl::buffer<T> &scratchpad,␣
↪std::int64_t scratchpad_size)
}
```

Input Parameters

queue
: Device queue where calculations will be performed.

m
: The number of rows in the matrix A ($0 \leq m$).

n
: The number of columns in the matrix A ($0 \leq n$).

a
: Buffer holding input matrix A. The second dimension of `a` must be at least $\max(1, n)$.

lda
: The leading dimension of `a`, at least $\max(1, m)$.

scratchpad
: Buffer holding scratchpad memory to be used by the routine for storing intermediate results.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by the *gerqf_scratchpad_size* function.

Output Parameters

a
Output buffer, overwritten by the factorization data as follows:

If $m \leq n$, the upper triangle of the subarray `a(1:m, n-m+1:n)` contains the $m \times m$ upper triangular matrix R; if $m \geq n$, the elements on and above the $(m-n)$-th subdiagonal contain the $m \times n$ upper trapezoidal matrix R

In both cases, the remaining elements, with the array `tau`, represent the orthogonal/unitary matrix Q as a product of $\min(m, n)$ elementary reflectors.

tau
Array, size at least $\min(m, n)$.

Contains scalars that define elementary reflectors for the matrix Q in its decomposition in a product of elementary reflectors.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -i`, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

gerqf (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event gerqf(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, T *a,␣
↪std::int64_t lda, T *tau, T *scratchpad, std::int64_t scratchpad_size, const␣
↪std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
Device queue where calculations will be performed.

m
The number of rows in the matrix A ($0 \leq m$).

n
The number of columns in the matrix A ($0 \leq n$).

a
Buffer holding input matrix A. The second dimension of `a` must be at least $\max(1, n)$.

lda
The leading dimension of `a`, at least $\max(1, m)$.

scratchpad
Buffer holding scratchpad memory to be used by the routine for storing intermediate results.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by the *gerqf_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a
Output buffer, overwritten by the factorization data as follows:

If $m \leq n$, the upper triangle of the subarray `a(1:m, n-m+1:n)` contains the $m \times m$ upper triangular matrix R; if $m \geq n$, the elements on and above the $(m-n)$-th subdiagonal contain the $m \times n$ upper trapezoidal matrix R

In both cases, the remaining elements, with the array `tau`, represent the orthogonal/unitary matrix Q as a product of $\min(m, n)$ elementary reflectors.

tau
Array, size at least $\min(m, n)$.

Contains scalars that define elementary reflectors for the matrix Q in its decomposition in a product of elementary reflectors.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -i`, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

gerqf_scratchpad_size

Computes size of scratchpad memory required for *gerqf* function.

Description

`gerqf_scratchpad_size` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

Computes the number of elements of type `T` the scratchpad memory to be passed to *gerqf* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

gerqf_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t gerqf_scratchpad_size(cl::sycl::queue &queue, std::int64_t m, std::int64_
↪t n, std::int64_t lda)
}
```

Input Parameters

queue
Device queue where calculations by the gerqf (buffer or USM version) function will be performed.

m
The number of rows in the matrix A ($0 \leq m$).

n
The number of columns in the matrix A ($0 \leq n$).

lda
The leading dimension of `a`; at least $\max(1, m)$.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *gerqf* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

getrf

Computes the LU factorization of a general $m \times n$ matrix.

Description

`getrf` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

The routine computes the LU factorization of a general $m \times n$ matrix A as $A = PLU$,

where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U is upper triangular (upper trapezoidal if $m < n$). The routine uses partial pivoting, with row interchanges.

getrf (BUFFER Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void getrf(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, cl::sycl::buffer<T,
↪1> &a, std::int64_t lda, cl::sycl::buffer<std::int64_t,1> &ipiv, cl::sycl::buffer<T,1>␣
↪&scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue
The queue where the routine should be executed.

m
The number of rows in the matrix A ($0 \leq m$).

n
The number of columns in A ($0 \leq n$).

a
Buffer holding input matrix A. The buffer a contains the matrix A. The second dimension of a must be at least $\max(1, n)$.

lda
The leading dimension of `a`.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *getrf_scratchpad_size* function.

Output Parameters

a
Overwritten by L and U. The unit diagonal elements of L are not stored.

ipiv
Array, size at least $\max(1, \min(m, n))$. Contains the pivot indices; for $1 \leq i \leq \min(m, n)$, row i was interchanged with row ipiv(i).

scratchpad
Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If info $= i$, u_{ii} is 0. The factorization has been completed, but U is exactly singular. Division by 0 will occur if you use the factor U for solving a system of linear equations.

If info equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

getrf (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event getrf(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, T *a,␣
↪std::int64_t lda, std::int64_t *ipiv, T *scratchpad, std::int64_t scratchpad_size,␣
↪const std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

m
The number of rows in the matrix A ($0 \leq m$).

n
The number of columns in A ($0 \leq n$).

a
Pointer to array holding input matrix A. The second dimension of `a` must be at least $\max(1, n)$.

lda
The leading dimension of `a`.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *getrf_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a
Overwritten by L and U. The unit diagonal elements of L are not stored.

ipiv
Array, size at least $\max(1, \min(m, n))$. Contains the pivot indices; for $1 \leq i \leq \min(m, n)$, row i was interchanged with row ipiv(i).

scratchpad
Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If info $= i$, u_{ii} is 0. The factorization has been completed, but U is exactly singular. Division by 0 will occur if you use the factor U for solving a system of linear equations.

If info equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

getrf_scratchpad_size

Computes size of scratchpad memory required for *getrf* function.

Description

`getrf_scratchpad_size` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *getrf* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

getrf_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t getrf_scratchpad_size(cl::sycl::queue &queue, std::int64_t m, std::int64_
↪t n, std::int64_t lda)
}
```

Input Parameters

queue
: Device queue where calculations by *getrf* function will be performed.

m
: The number of rows in the matrix A ($0 \leq m$).

n
: The number of columns in A ($0 \leq n$).

lda
: The leading dimension of `a` ($n \leq$ lda).

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

: Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *getrf* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

getri

Computes the inverse of an LU-factored general matrix determined by *getrf*.

Description

`getri` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

The routine computes the inverse A^{-1} of a general matrix A. Before calling this routine, call *getrf* to factorize A.

getri (BUFFER Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void getri(cl::sycl::queue &queue, std::int64_t n, cl::sycl::buffer<T,1> &a,␣
↪std::int64_t lda, cl::sycl::buffer<std::int64_t,1> &ipiv, cl::sycl::buffer<T,1> &
↪scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

n
: The order of the matrix A ($0 \leq n$).

a
: The buffer `a` as returned by *getrf*. Must be of size at least lda $\cdot \max(1, n)$.

lda
: The leading dimension of `a` ($n \leq$ lda).

ipiv
: The buffer as returned by *getrf*. The dimension of `ipiv` must be at least $\max(1, n)$.

scratchpad_size
: Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *getri_scratchpad_size* function.

Output Parameters

a
Overwritten by the $n \times n$ matrix A.

scratchpad
Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If info equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

getri (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event getri(cl::sycl::queue &queue, std::int64_t n, T *a, std::int64_t lda,␣
↪std::int64_t *ipiv, T *scratchpad, std::int64_t scratchpad_size, const std::vector
↪<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

n
The order of the matrix A ($0 \leq n$).

a
The array as returned by *getrf*. Must be of size at least lda $\cdot \max(1, n)$.

lda
The leading dimension of `a` ($n \leq$ lda).

ipiv
The array as returned by *getrf*. The dimension of `ipiv` must be at least $\max(1, n)$.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *getri_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a
Overwritten by the $n \times n$ matrix A.

scratchpad
Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If info equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

getri_scratchpad_size

Computes size of scratchpad memory required for *getri* function.

Description

`getri_scratchpad_size` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *getri* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

getri_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t getri_scratchpad_size(cl::sycl::queue &queue, std::int64_t n, std::int64_
↪t lda)
}
```

Input Parameters

queue
: Device queue where calculations by *getri* function will be performed.

n
: The order of the matrix A ($0 \leq n$).

lda
: The leading dimension of `a` ($n \leq$ lda).

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

: Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *getri* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

getrs

Solves a system of linear equations with an LU-factored square coefficient matrix, with multiple right-hand sides.

Description

`getrs` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

The routine solves for X the following systems of linear equations:

$AX = B$	if `trans=oneapi::mkl::transpose::nontrans`
$A^T X = B$	if `trans=oneapi::mkl::transpose::trans`
$A^H X = B$	if `trans=oneapi::mkl::transpose::conjtrans`

Before calling this routine, you must call *getrf* to compute the LU factorization of A.

getrs (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void getrs(cl::sycl::queue &queue, oneapi::mkl::transpose trans, std::int64_t n,␣
↪std::int64_t nrhs, cl::sycl::buffer<T,1> &a, std::int64_t lda, cl::sycl::buffer
↪<std::int64_t,1> &ipiv, cl::sycl::buffer<T,1> &b, std::int64_t ldb, cl::sycl::buffer<T,
↪1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

trans
: Indicates the form of the equations:

  If `trans=oneapi::mkl::transpose::nontrans`, then $AX = B$ is solved for X.

  If `trans=oneapi::mkl::transpose::trans`, then $A^T X = B$ is solved for X.

If `trans=oneapi::mkl::transpose::conjtrans`, then $A^H X = B$ is solved for X.

n
The order of the matrix A and the number of rows in matrix $B (0 \leq n)$.

nrhs
The number of right-hand sides ($0 \leq$ nrhs).

a
Buffer containing the factorization of the matrix A, as returned by *getrf*. The second dimension of `a` must be at least $\max(1, n)$.

lda
The leading dimension of `a`.

ipiv
Array, size at least $\max(1, n)$. The `ipiv` array, as returned by *getrf*.

b
The array `b` contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of `b` must be at least $\max(1, \text{nrhs})$.

ldb
The leading dimension of `b`.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *getrs_scratchpad_size* function.

Output Parameters

b
The buffer `b` is overwritten by the solution matrix X.

scratchpad
Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If `info=-i`, the i-th parameter had an illegal value.

If `info=i`, the i-th diagonal element of U is zero, and the solve could not be completed.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

getrs (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event getrs(cl::sycl::queue &queue, oneapi::mkl::transpose trans, std::int64_
↪t n, std::int64_t nrhs, T *a, std::int64_t lda, std::int64_t *ipiv, T *b, std::int64_t␣
↪ldb, T *scratchpad, std::int64_t scratchpad_size, const std::vector<cl::sycl::event> &
↪events = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

trans
Indicates the form of the equations:

If `trans=oneapi::mkl::transpose::nontrans`, then $AX = B$ is solved for X.

If `trans=oneapi::mkl::transpose::trans`, then $A^T X = B$ is solved for X.

If `trans=oneapi::mkl::transpose::conjtrans`, then $A^H X = B$ is solved for X.

n
The order of the matrix A and the number of rows in matrix $B (0 \leq n)$.

nrhs
The number of right-hand sides ($0 \leq$ nrhs).

a
Pointer to array containing the factorization of the matrix A, as returned by *getrf*. The second dimension of `a` must be at least $\max(1, n)$.

lda
The leading dimension of `a`.

ipiv
Array, size at least $\max(1, n)$. The `ipiv` array, as returned by *getrf*.

b
The array `b` contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of `b` must be at least $\max(1, \text{nrhs})$.

ldb
The leading dimension of `b`.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *getrs_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

b
The array `b` is overwritten by the solution matrix X.

scratchpad
Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If `info=-i`, the i-th parameter had an illegal value.

If `info=i`, the i-th diagonal element of U is zero, and the solve could not be completed.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

getrs_scratchpad_size

Computes size of scratchpad memory required for *getrs* function.

Description

`getrs_scratchpad_size` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type T the scratchpad memory to be passed to *getrs* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

getrs_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t getrs_scratchpad_size(cl::sycl::queue &queue, oneapi::mkl::transpose␣
↪trans, std::int64_t n, std::int64_t nrhs, std::int64_t lda, std::int64_t ldb)
}
```

Input Parameters

queue
: Device queue where calculations by *getrs* function will be performed.

trans
: Indicates the form of the equations:

  If `trans=oneapi::mkl::transpose::nontrans`, then $AX = B$ is solved for X.

  If `trans=oneapi::mkl::transpose::trans`, then $A^T X = B$ is solved for X.

  If `trans=oneapi::mkl::transpose::conjtrans`, then $A^H X = B$ is solved for X.

n
: The order of the matrix A $(0 \leq n)$ and the number of rows in matrix $B(0 \leq n)$.

nrhs
: The number of right-hand sides ($0 \leq$ nrhs).

lda
: The leading dimension of `a`.

ldb
: The leading dimension of `b`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

: Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *getrs* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

hetrf

Computes the Bunch-Kaufman factorization of a complex Hermitian matrix.

Description

`hetrf` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

The routine computes the factorization of a complex Hermitian matrix A using the Bunch-Kaufman diagonal pivoting method. The form of the factorization is:

- if `upper_lower=uplo::upper`, $A = UDU^H$
- if `upper_lower=uplo::lower`, $A = LDL^H$

where A is the input matrix, U and L are products of permutation and triangular matrices with unit diagonal (upper triangular for U and lower triangular for L), and D is a Hermitian block-diagonal matrix with 1×1 and 2×2 diagonal blocks. U and L have 2×2 unit diagonal blocks corresponding to the 2×2 blocks of D.

hetrf (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void hetrf(cl::sycl::queue &queue, oneapi::mkl::uplo upper_lower, std::int64_t n,␣
↪cl::sycl::buffer<T,1> &a, std::int64_t lda, cl::sycl::buffer<int_64,1> &ipiv,␣
↪cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If `upper_lower=uplo::upper`, the buffer `a` stores the upper triangular part of the matrix A, and A is factored as UDU^H.

If `upper_lower=uplo::lower`, the buffer `a` stores the lower triangular part of the matrix A, and A is factored as LDL^H.

n
The order of matrix A ($0 \leq n$).

a
The buffer `a`, size $\max(1, \text{lda} \cdot n)$. The buffer `a` contains either the upper or the lower triangular part of the matrix A (see `upper_lower`). The second dimension of `a` must be at least $\max(1, n)$.

lda
The leading dimension of `a`.

scratchpad
Buffer holding scratchpad memory to be used by the routine for storing intermediate results.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *hetrf_scratchpad_size* function.

Output Parameters

a
The upper or lower triangular part of a is overwritten by details of the block-diagonal matrix D and the multipliers used to obtain the factor U (or L).

ipiv
Buffer, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of D. If $\text{ipiv}(i) = k > 0$, then d_{ii} is a 1×1 block, and the i-th row and column of A was interchanged with the k-th row and column.

If `upper_lower=oneapi::mkl::uplo::upper` and $\text{ipiv}(i) = \text{ipiv}(i-1) = -m < 0$, then D has a 2×2 block in rows/columns i and i-1, and $(i-1)$-th row and column of A was interchanged with the m-th row and column.

If `upper_lower=oneapi::mkl::uplo::lower` and $\text{ipiv}(i) = \text{ipiv}(i+1) = -m < 0$, then D has a 2×2 block in rows/columns i and $i+1$, and $(i+1)$-th row and column of A was interchanged with the m-th row and column.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -i`, the i-th parameter had an illegal value.

If info $= i$, d_{ii} is 0. The factorization has been completed, but D is exactly singular. Division by 0 will occur if you use D for solving a system of linear equations.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

hetrf (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event hetrf(cl::sycl::queue &queue, oneapi::mkl::uplo upper_lower,␣
↪std::int64_t n, T *a, std::int64_t lda, int_64 *ipiv, T *scratchpad, std::int64_t␣
↪scratchpad_size, const std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If `upper_lower=uplo::upper`, the array `a` stores the upper triangular part of the matrix A, and A is factored as UDU^H.

If `upper_lower=uplo::lower`, the array `a` stores the lower triangular part of the matrix A, and A is factored as LDL^H.

n
The order of matrix A ($0 \leq n$).

a
The pointer to A, size $\max(1, \text{lda} \cdot n)$, containing either the upper or the lower triangular part of the matrix A (see `upper_lower`). The second dimension of `a` must be at least $\max(1, n)$.

lda
The leading dimension of `a`.

scratchpad
Pointer to scratchpad memory to be used by the routine for storing intermediate results.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *hetrf_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a

The upper or lower triangular part of a is overwritten by details of the block-diagonal matrix D and the multipliers used to obtain the factor U (or L).

ipiv

Pointer to array of size at least $\max(1, n)$. Contains details of the interchanges and the block structure of D. If $\text{ipiv}(i) = k > 0$, then d_{ii} is a 1×1 block, and the i-th row and column of A was interchanged with the k-th row and column.

If `upper_lower=oneapi::mkl::uplo::upper` and $\text{ipiv}(i) = \text{ipiv}(i-1) = -m < 0$, then D has a 2×2 block in rows/columns i and $i-1$, and $(i-1)$-th row and column of A was interchanged with the m-th row and column.

If `upper_lower=oneapi::mkl::uplo::lower` and $\text{ipiv}(i) = \text{ipiv}(i+1) = -m < 0$, then D has a 2×2 block in rows/columns i and $i+1$, and $(i+1)$-th row and column of A was interchanged with the m-th row and column.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -i`, the i-th parameter had an illegal value.

If $\text{info} = i$, d_{ii} is 0. The factorization has been completed, but D is exactly singular. Division by 0 will occur if you use D for solving a system of linear equations.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

hetrf_scratchpad_size

Computes size of scratchpad memory required for *hetrf* function.

Description

`hetrf_scratchpad_size` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *hetrf* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

hetrf_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t hetrf_scratchpad_size(cl::sycl::queue &queue, oneapi::mkl::uplo upper_
↪lower, std::int64_t n, std::int64_t lda)
}
```

Input Parameters

queue
: Device queue where calculations by *hetrf* function will be performed.

upper_lower
: Indicates whether the upper or lower triangular part of A is stored and how A is factored:

  If `upper_lower=uplo::upper`, the buffer `a` stores the upper triangular part of the matrix A, and A is factored as UDU^H.

  If `upper_lower=uplo::lower`, the buffer `a` stores the lower triangular part of the matrix A, and A is factored as LDL^H

n
: The order of the matrix A ($0 \leq n$).

lda
: The leading dimension of `a`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

: Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *hetrf* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

orgqr

Generates the real orthogonal matrix Q of the QR factorization formed by *geqrf*.

Description

`orgqr` supports the following precisions.

T
float
double

The routine generates the whole or part of $m \times m$ orthogonal matrix Q of the QR factorization formed by the routine *geqrf*.

Usually Q is determined from the QR factorization of an `m` by `p` matrix A with $m \geq p$. To compute the whole matrix Q, use:

```
oneapi::mkl::lapack::orgqr(queue, m, m, p, a, lda, tau, scratchpad, scratchpad_size)
```

To compute the leading p columns of Q (which form an orthonormal basis in the space spanned by the columns of A):

```
oneapi::mkl::lapack::orgqr(queue, m, p, p, a, lda, tau, scratchpad, scratchpad_size)
```

To compute the matrix Q^k of the QR factorization of leading k columns of the matrix A:

```
oneapi::mkl::lapack::orgqr(queue, m, m, k, a, lda, tau, scratchpad, scratchpad_size)
```

To compute the leading k columns of Q^k (which form an orthonormal basis in the space spanned by leading k columns of the matrix A):

```
oneapi::mkl::lapack::orgqr(queue, m, k, k, a, lda, tau, scratchpad, scratchpad_size)
```

orgqr (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void orgqr(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, std::int64_t k,␣
↪cl::sycl::buffer<T,1> &a, std::int64_t lda, cl::sycl::buffer<T,1> &tau,␣
↪cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue
The queue where the routine should be executed.

m
The number of rows in the matrix A ($0 \leq m$).

n
The number of columns in the matrix A ($0 \leq n \leq m$).

k
The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

a
The buffer `a` as returned by *geqrf*.

lda
The leading dimension of `a` (lda $\leq m$).

tau
The buffer `tau` as returned by *geqrf*.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *orgqr_scratchpad_size* function.

Output Parameters

a
Overwritten by n leading columns of the $m \times m$ orthogonal matrix Q.

scratchpad
Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

orgqr (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event orgqr(cl::sycl::queue &queue, std::int64_t m, std::int64_t n,␣
↪std::int64_t k, T *a, std::int64_t lda, T *tau, T *scratchpad, std::int64_t scratchpad_
↪size, const std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

m
The number of rows in the matrix A ($0 \leq m$).

n
The number of columns in the matrix A ($0 \leq n \leq m$).

k
The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

a
The pointer to `a` as returned by *geqrf*.

lda
The leading dimension of `a` (lda $\leq m$).

tau
The pointer to `tau` as returned by *geqrf*.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *orgqr_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a
Overwritten by n leading columns of the $m \times m$ orthogonal matrix Q.

scratchpad
Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

> Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:
>
> If info $= -i$, the i-th parameter had an illegal value.
>
> If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

orgqr_scratchpad_size

Computes size of scratchpad memory required for *orgqr* function.

Description

`orgqr_scratchpad_size` supports the following precisions.

T
`float`
`double`

Computes the number of elements of type `T` the scratchpad memory to be passed to *orgqr* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

orgqr_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t orgqr_scratchpad_size(cl::sycl::queue &queue, std::int64_t m, std::int64_
↪t n, std::int64_t k, std::int64_t lda)
}
```

Input Parameters

queue
: Device queue where calculations by *orgqr* function will be performed.

m
: The number of rows in the matrix A ($0 \leq m$).

n
: The number of columns in the matrix A ($0 \leq n \leq m$).

k
: The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

lda
: The leading dimension of `a`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

: Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *orgqr* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

ormqr

Multiplies a real matrix by the orthogonal matrix Q of the QR factorization formed by *geqrf*.

Description

`ormqr` supports the following precisions.

T
`float`
`double`

The routine multiplies a rectangular real $m \times n$ matrix C by Q or Q^T, where Q is the complex unitary matrix defined as a product of k elementary reflectors $H(i)$ of order n: $Q = H(1)^T H(2)^T ... H(k)^T$ as returned by the RQ factorization routine *gerqf*.

Depending on the parameters `side` and `trans`, the routine can form one of the matrix products QC, Q^TC, CQ, or CQ^T (overwriting the result over C).

ormqr (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void ormqr(cl::sycl::queue &queue, oneapi::mkl::side side, oneapi::mkl::transpose␣
↪trans, std::int64_t m, std::int64_t n, std::int64_t k, cl::sycl::buffer<T,1> &a,␣
↪std::int64_t lda, cl::sycl::buffer<T,1> &tau, cl::sycl::buffer<T,1> &c, std::int64_t␣
↪ldc, cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

side
: If `side = oneapi::mkl::side::left`, Q or Q^T is applied to C from the left.

: If `side = oneapi::mkl::side::right`, Q or Q^T is applied to C from the right.

trans
: If `trans = oneapi::mkl::transpose::nontrans`, the routine multiplies C by Q.

: If `trans = oneapi::mkl::transpose::trans`, the routine multiplies C by Q^T.

m
: The number of rows in the matrix C ($0 \leq m$).

n
: The number of columns in the matrix C ($0 \leq n$).

k
: The number of elementary reflectors whose product defines the matrix Q

: If `side = oneapi::mkl::side::left`, $0 \leq k \leq m$

: If `side = oneapi::mkl::side::right`, $0 \leq k \leq n$

a
The buffer `a` as returned by *geqrf*. The second dimension of `a` must be at least $\max(1, k)$.

lda
The leading dimension of `a`.

tau
The buffer `tau` as returned by *geqrf*.

c
The buffer `c` contains the matrix C. The second dimension of `c` must be at least $\max(1, n)$.

ldc
The leading dimension of `c`.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by the *ormqr_scratchpad_size* function.

Output Parameters

c
Overwritten by the product QC, Q^TC, CQ, or CQ^T (as specified by `side` and `trans`).

scratchpad
Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

ormqr (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event ormqr(cl::sycl::queue &queue, oneapi::mkl::side side,␣
↪oneapi::mkl::transpose trans, std::int64_t m, std::int64_t n, std::int64_t k, T *a,␣
↪std::int64_t lda, T *tau, T *c, std::int64_t ldc, T *scratchpad, std::int64_t␣
↪scratchpad_size, const std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

side
If `side = oneapi::mkl::side::left`, Q or Q^T is applied to C from the left.

If `side = oneapi::mkl::side::right`, Q or Q^T is applied to C from the right.

trans
If `trans = oneapi::mkl::transpose::nontrans`, the routine multiplies C by Q.

If `trans = oneapi::mkl::transpose::trans`, the routine multiplies C by Q^T.

m
The number of rows in the matrix C ($0 \leq m$).

n
The number of columns in the matrix C ($0 \leq n$).

k
The number of elementary reflectors whose product defines the matrix Q

If `side = oneapi::mkl::side::left`, $0 \leq k \leq m$

If `side = oneapi::mkl::side::right`, $0 \leq k \leq n$

a
The pointer to `a` as returned by *geqrf*. The second dimension of `a` must be at least $\max(1, k)$.

lda
The leading dimension of `a`.

tau
The pointer to `tau` as returned by *geqrf*.

c
The pointer `c` points to the matrix C. The second dimension of `c` must be at least $\max(1, n)$.

ldc
The leading dimension of `c`.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by the *ormqr_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

c
Overwritten by the product QC, Q^TC, CQ, or CQ^T (as specified by `side` and `trans`).

scratchpad
Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

ormqr_scratchpad_size

Computes size of scratchpad memory required for *ormqr* function.

Description

`ormqr_scratchpad_size` supports the following precisions.

T
float
double

Computes the number of elements of type `T` the scratchpad memory to be passed to *ormqr* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

ormqr_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t ormqr_scratchpad_size(cl::sycl::queue &queue, oneapi::mkl::side side,␣
↪oneapi::mkl::transpose trans, std::int64_t m, std::int64_t n, std::int64_t k,␣
↪std::int64_t lda, std::int64_t ldc, std::int64_t &scratchpad_size)
}
```

Input Parameters

queue
: Device queue where calculations by *ormqr* function will be performed.

side
: If `side=oneapi::mkl::side::left`, Q or Q^T is applied to C from the left.

: If `side=oneapi::mkl::side::right`, Q or Q^T is applied to C from the right.

trans
: If `trans=oneapi::mkl::transpose::nontrans`, the routine multiplies C by Q.

: If `trans=oneapi::mkl::transpose::trans`, the routine multiplies C by Q^T.

m
: The number of rows in the matrix C ($0 \leq m$).

n
: The number of columns in the matrix C ($0 \leq n \leq m$).

k
: The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

lda
: The leading dimension of `a`.

ldc
: The leading dimension of `c`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

: Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *ormqr* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

ormrq

Multiplies a real matrix by the orthogonal matrix Q of the RQ factorization formed by *gerqf*.

Description

`ormrq` supports the following precisions.

T
float
double

The routine multiplies a rectangular real $m \times n$ matrix C by Q or Q^T, where Q is the complex unitary matrix defined as a product of k elementary reflectors $H(i)$ of order n: $Q = H(1)^T H(2)^T ... H(k)^T$ as returned by the RQ factorization routine *gerqf*.

Depending on the parameters `side` and `trans`, the routine can form one of the matrix products QC, Q^TC, CQ, or CQ^T (overwriting the result over C).

ormrq (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void ormrq(cl::sycl::queue &queue, oneapi::mkl::side side, oneapi::mkl::transpose␣
↪trans, std::int64_t m, std::int64_t n, std::int64_t k, cl::sycl::buffer<T,1> &a,␣
↪std::int64_t lda, cl::sycl::buffer<T,1> &tau, cl::sycl::buffer<T,1> &c, std::int64_t␣
↪ldc, cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

side
: If `side = oneapi::mkl::side::left`, Q or Q^T is applied to C from the left.

: If `side = oneapi::mkl::side::right`, Q or Q^T is applied to C from the right.

trans
: If `trans = oneapi::mkl::transpose::nontrans`, the routine multiplies C by Q.

: If `trans = oneapi::mkl::transpose::trans`, the routine multiplies C by Q^T.

m
: The number of rows in the matrix C ($0 \leq m$).

n
The number of columns in the matrix C ($0 \leq n$).

k
The number of elementary reflectors whose product defines the matrix Q

If `side = oneapi::mkl::side::left`, $0 \leq k \leq m$

If `side = oneapi::mkl::side::right`, $0 \leq k \leq n$

a
The buffer `a` as returned by *gerqf*. The second dimension of `a` must be at least $\max(1, k)$.

lda
The leading dimension of `a`.

tau
The buffer `tau` as returned by *gerqf*.

c
The buffer `c` contains the matrix C. The second dimension of `c` must be at least $\max(1, n)$.

ldc
The leading dimension of `c`.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by the *ormrq_scratchpad_size* function.

Output Parameters

c
Overwritten by the product QC, Q^TC, CQ, or CQ^T (as specified by `side` and `trans`).

scratchpad
Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

ormrq (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event ormrq(cl::sycl::queue &queue, oneapi::mkl::side side,␣
↪oneapi::mkl::transpose trans, std::int64_t m, std::int64_t n, std::int64_t k, T *a,␣
↪std::int64_t lda, T *tau, T *c, std::int64_t ldc, T *scratchpad, std::int64_t␣
↪scratchpad_size, const std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

side
If `side = oneapi::mkl::side::left`, Q or Q^T is applied to C from the left.

If `side = oneapi::mkl::side::right`, Q or Q^T is applied to C from the right.

trans
If `trans = oneapi::mkl::transpose::nontrans`, the routine multiplies C by Q.

If `trans = oneapi::mkl::transpose::trans`, the routine multiplies C by Q^T.

m
The number of rows in the matrix C ($0 \leq m$).

n
The number of columns in the matrix C ($0 \leq n$).

k
The number of elementary reflectors whose product defines the matrix Q

If `side = oneapi::mkl::side::left`, $0 \leq k \leq m$

If `side = oneapi::mkl::side::right`, $0 \leq k \leq n$

a
The pointer to `a` as returned by *gerqf*. The second dimension of `a` must be at least $\max(1, k)$.

lda
The leading dimension of `a`.

tau
The pointer to `tau` as returned by *gerqf*.

c
The pointer `c` points to the matrix C. The second dimension of `c` must be at least $\max(1, n)$.

ldc
The leading dimension of `c`.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by the *ormrq_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

c
Overwritten by the product QC, Q^TC, CQ, or CQ^T (as specified by `side` and `trans`).

scratchpad
Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

ormrq_scratchpad_size

Computes size of scratchpad memory required for *ormrq* function.

Description

`ormrq_scratchpad_size` supports the following precisions.

T
float
double

Computes the number of elements of type `T` the scratchpad memory to be passed to *ormrq* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

ormrq_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t ormrq_scratchpad_size(cl::sycl::queue &queue, oneapi::mkl::side side,␣
↪oneapi::mkl::transpose trans, std::int64_t m, std::int64_t n, std::int64_t k,␣
↪std::int64_t lda, std::int64_t ldc);
}
```

Input Parameters

queue
: Device queue where calculations by the ormrq function will be performed.

side
: If `side = oneapi::mkl::side::left`, Q or Q^T is applied to C from the left.

: If `side = oneapi::mkl::side::right`, Q or Q^T is applied to C from the right.

trans
: If `trans=oneapi::mkl::transpose::nontrans`, the routine multiplies C by Q.

: If `trans=oneapi::mkl::transpose::trans`, the routine multiplies C by Q^T.

m
: The number of rows in the matrix C ($0 \leq m$).

n
: The number of columns in the matrix C ($0 \leq n \leq m$).

k
: The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

lda
: The leading dimension of `a`.

ldc
: The leading dimension of `c`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

: Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *ormrq* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

potrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite matrix.

Description

`potrf` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

The routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix A:

$A = U^T U$ for real data, $A = U^H U$ for complex data	if upper_lower=`oneapi::mkl::uplo::upper`
$A = LL^T$ for real data, $A = LL^H$ for complex data	if upper_lower=`oneapi::mkl::uplo::lower`

where L is a lower triangular matrix and U is upper triangular.

potrf (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void potrf(cl::sycl::queue &queue, oneapi::mkl::uplo upper_lower, std::int64_t n,␣
↪cl::sycl::buffer<T,1> &a, std::int64_t lda, cl::sycl::buffer<T,1> &scratchpad,␣
↪std::int64_t scratchpad_size)
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If upper_lower=`oneapi::mkl::uplo::upper`, the array `a` stores the upper triangular part of the matrix A, and the strictly lower triangular part of the matrix is not referenced.

If upper_lower=`oneapi::mkl::uplo::lower`, the array `a` stores the lower triangular part of the matrix A, and the strictly upper triangular part of the matrix is not referenced.

n
Specifies the order of the matrix A ($0 \leq n$).

a
Buffer holding input matrix A. The buffer `a` contains either the upper or the lower triangular part of the matrix A (see upper_lower). The second dimension of `a` must be at least $\max(1, n)$.

lda
The leading dimension of `a`.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *potrf_scratchpad_size* function.

Output Parameters

a
The buffer `a` is overwritten by the Cholesky factor U or L, as specified by `upper_lower`.

scratchpad
Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If info $= i$, and *detail()* returns 0, then the leading minor of order i (and therefore the matrix A itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix A.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

potrf (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event potrf(cl::sycl::queue &queue, oneapi::mkl::uplo upper_lower,␣
↪std::int64_t n, T *a, std::int64_t lda, T *scratchpad, std::int64_t scratchpad_size,␣
↪const std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If upper_lower=`oneapi::mkl::uplo::upper`, the array `a` stores the upper triangular part of the matrix A, and the strictly lower triangular part of the matrix is not referenced.

If upper_lower=`oneapi::mkl::uplo::lower`, the array `a` stores the lower triangular part of the matrix A, and the strictly upper triangular part of the matrix is not referenced.

n
Specifies the order of the matrix A ($0 \leq n$).

a
Pointer to input matrix A. The array `a` contains either the upper or the lower triangular part of the matrix A (see upper_lower). The second dimension of `a` must be at least $\max(1, n)$.

lda
The leading dimension of `a`.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *potrf_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a
The memory pointer to by pointer `a` is overwritten by the Cholesky factor U or L, as specified by `upper_lower`.

scratchpad
Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If info $= i$, and *detail()* returns 0, then the leading minor of order i (and therefore the matrix A itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix A.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

potrf_scratchpad_size

Computes size of scratchpad memory required for *potrf* function.

Description

`potrf_scratchpad_size` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type T the scratchpad memory to be passed to *potrf* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

potrf_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t potrf_scratchpad_size(cl::sycl::queue &queue, oneapi::mkl::uplo upper_
↪lower, std::int64_t n, std::int64_t lda)
}
```

Input Parameters

queue
Device queue where calculations by *potrf* function will be performed.

upper_lower
Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If `upper_lower = oneapi::mkl::uplo::upper`, the array `a` stores the upper triangular part of the matrix A, and the strictly lower triangular part of the matrix is not referenced.

If `upper_lower = oneapi::mkl::uplo::lower`, the array `a` stores the lower triangular part of the matrix A, and the strictly upper triangular part of the matrix is not referenced.

n
Specifies the order of the matrix A ($0 \leq n$).

lda
The leading dimension of `a`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *potrf* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

potri

Computes the inverse of a symmetric (Hermitian) positive-definite matrix using the Cholesky factorization.

Description

`potri` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

The routine computes the inverse A^{-1} of a symmetric positive definite or, for complex flavors, Hermitian positive-definite matrix A. Before calling this routine, call *potrf* to factorize A.

potri (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void potri(cl::sycl::queue &queue, oneapi::mkl::uplo upper_lower, std::int64_t n,␣
↪cl::sycl::buffer<T,1> &a, std::int64_t lda, cl::sycl::buffer<T,1> &scratchpad,␣
↪std::int64_t scratchpad_size)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

upper_lower
: Indicates how the input matrix A has been factored:

  If `upper_lower = oneapi::mkl::uplo::upper`, the upper triangle of A is stored.

  If `upper_lower = oneapi::mkl::uplo::lower`, the lower triangle of A is stored.

n
: Specifies the order of the matrix A ($0 \leq n$).

a
: Contains the factorization of the matrix A, as returned by *potrf*. The second dimension of `a` must be at least $\max(1, n)$.

lda
The leading dimension of `a`.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *potri_scratchpad_size* function.

Output Parameters

a
Overwritten by the upper or lower triangle of the inverse of A. Specified by `upper_lower`.

scratchpad
Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If info $= i$, the i-th diagonal element of the Cholesky factor (and therefore the factor itself) is zero, and the inversion could not be completed.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

potri (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event potri(cl::sycl::queue &queue, oneapi::mkl::uplo upper_lower,␣
↪std::int64_t n, T *a, std::int64_t lda, T *scratchpad, std::int64_t scratchpad_size,␣
↪const std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Indicates how the input matrix A has been factored:

If `upper_lower = oneapi::mkl::uplo::upper`, the upper triangle of A is stored.

If `upper_lower = oneapi::mkl::uplo::lower`, the lower triangle of A is stored.

n
Specifies the order of the matrix A ($0 \leq n$).

a
Contains the factorization of the matrix A, as returned by *potrf*. The second dimension of `a` must be at least $\max(1, n)$.

lda
The leading dimension of `a`.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *potri_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a
Overwritten by the upper or lower triangle of the inverse of A. Specified by `upper_lower`.

scratchpad
Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If info $= i$, the i-th diagonal element of the Cholesky factor (and therefore the factor itself) is zero, and the inversion could not be completed.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

potri_scratchpad_size

Computes size of scratchpad memory required for *potri* function.

Description

`potri_scratchpad_size` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *potri* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

potri_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t potri_scratchpad_size(cl::sycl::queue &queue, oneapi::mkl::uplo upper_
↪lower, std::int64_t n, std::int64_t lda)
}
```

Input Parameters

queue
Device queue where calculations by *potri* function will be performed.

upper_lower
Indicates how the input matrix A has been factored:

If `upper_lower = oneapi::mkl::uplo::upper`, the upper triangle of A is stored.

If `upper_lower = oneapi::mkl::uplo::lower`, the lower triangle of A is stored.

n
Specifies the order of the matrix A ($0 \leq n$).

lda
The leading dimension of `a`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *potri* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

potrs

Solves a system of linear equations with a Cholesky-factored symmetric (Hermitian) positive-definite coefficient matrix.

Description

`potrs` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

The routine solves for X the system of linear equations $AX = B$ with a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix A, given the Cholesky factorization of A:

$A = U^T U$ for real data, $A = U^H U$ for complex data	if `upper_lower=oneapi::mkl::uplo::upper`
$A = LL^T$ for real data, $A = LL^H$ for complex data	if `upper_lower=oneapi::mkl::uplo::lower`

where L is a lower triangular matrix and U is upper triangular. The system is solved with multiple right-hand sides stored in the columns of the matrix B.

Before calling this routine, you must call *potrf* to compute the Cholesky factorization of A.

potrs (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void potrs(cl::sycl::queue &queue, oneapi::mkl::uplo upper_lower, std::int64_t n,␣
↪std::int64_t nrhs, cl::sycl::buffer<T,1> &a, std::int64_t lda, cl::sycl::buffer<T,1> &
↪b, std::int64_t ldb, cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Indicates how the input matrix has been factored:

If `upper_lower = oneapi::mkl::uplo::upper`, the upper triangle U of A is stored, where $A = U^T$‘U for real data, $A = U^H U$ for complex data.

If `upper_lower = oneapi::mkl::uplo::lower`, the lower triangle L of A is stored, where $A = LL^T$ for real data, $A = LL^H$ for complex data.

n
The order of matrix A ($0 \leq n$).

nrhs
The number of right-hand sides ($0 \leq$ nrhs).

a
Buffer containing the factorization of the matrix A, as returned by *potrf*. The second dimension of `a` must be at least $\max(1, n)$.

lda
The leading dimension of `a`.

b
The array `b` contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of `b` must be at least $\max(1, \text{nrhs})$.

ldb
The leading dimension of `b`.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *potrs_scratchpad_size* function.

Output Parameters

b
Overwritten by the solution matrix X.

scratchpad
Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If info $= i$, the i-th diagonal element of the Cholesky factor is zero, and the solve could not be completed.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

potrs (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event potrs(cl::sycl::queue &queue, oneapi::mkl::uplo upper_lower,␣
↪std::int64_t n, std::int64_t nrhs, T *a, std::int64_t lda, T *b, std::int64_t ldb, T␣
↪*scratchpad, std::int64_t scratchpad_size, const std::vector<cl::sycl::event> &events␣
↪= {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Indicates how the input matrix has been factored:

If `upper_lower = oneapi::mkl::uplo::upper`, the upper triangle U of A is stored, where $A = U^T U$ for real data, $A = U^H U$ for complex data.

If `upper_lower = oneapi::mkl::uplo::lower`, the lower triangle L of A is stored, where $A = LL^T$ for real data, $A = LL^H$ for complex data.

n
The order of matrix A ($0 \leq n$).

nrhs
The number of right-hand sides ($0 \leq$ nrhs).

a
Pointer to array containing the factorization of the matrix A, as returned by *potrf*. The second dimension of `a` must be at least $\max(1, n)$.

lda
The leading dimension of `a`.

b
The array `b` contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of `b` must be at least $\max(1, \text{nrhs})$.

ldb
The leading dimension of `b`.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *potrs_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

b
Overwritten by the solution matrix X.

scratchpad
Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

> Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:
>
> If info $= -i$, the i-th parameter had an illegal value.
>
> If info $= i$, the i-th diagonal element of the Cholesky factor is zero, and the solve could not be completed.
>
> If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

potrs_scratchpad_size

Computes size of scratchpad memory required for *potrs* function.

Description

`potrs_scratchpad_size` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *potrs* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

potrs_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t potrs_scratchpad_size(cl::sycl::queue &queue, oneapi::mkl::uplo upper_
↪lower, std::int64_t n, std::int64_t nrhs, std::int64_t lda, std::int64_t ldb)
}
```

Input Parameters

queue
Device queue where calculations by *potrs* function will be performed.

upper_lower
Indicates how the input matrix has been factored:

If `upper_lower = oneapi::mkl::uplo::upper`, the upper triangle U of A is stored, where $A = U^T U$ for real data, $A = U^H U$ for complex data.

If `upper_lower = oneapi::mkl::uplo::lower`, the lower triangle L of A is stored, where $A = LL^T$ for real data, $A = LL^H$ for complex data.

n
The order of matrix A ($0 \leq n$).

nrhs
The number of right-hand sides ($0 \leq nrhs$).

lda
The leading dimension of `a`.

ldb
The leading dimension of `b`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *potrs* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

sytrf

Computes the Bunch-Kaufman factorization of a symmetric matrix.

Description

`sytrf` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

The routine computes the factorization of a real/complex symmetric matrix A using the Bunch-Kaufman diagonal pivoting method. The form of the factorization is:

- if `upper_lower=uplo::upper`, $A = UDU^T$
- if `upper_lower=uplo::lower`, $A = LDL^T$

where A is the input matrix, U and L are products of permutation and triangular matrices with unit diagonal (upper triangular for U and lower triangular for L), and D is a symmetric block-diagonal matrix with 1×1 and 2×2 diagonal blocks. U and L have 2×2 unit diagonal blocks corresponding to the 2×2 blocks of D.

sytrf (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void sytrf(cl::sycl::queue &queue, oneapi::mkl::uplo upper_lower, std::int64_t n,␣
↪cl::sycl::buffer<T,1> &a, std::int64_t lda, cl::sycl::buffer<int_64,1> &ipiv,␣
↪cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

upper_lower
: Indicates whether the upper or lower triangular part of A is stored and how A is factored:

  If `upper_lower=uplo::upper`, the buffer `a` stores the upper triangular part of the matrix A, and A is factored as UDU^T.

If `upper_lower=uplo::lower`, the buffer `a` stores the lower triangular part of the matrix A, and A is factored as LDL^T.

n

The order of matrix A ($0 \leq n$).

a

The buffer `a`, size $\max(1, lda \cdot n)$. The buffer `a` contains either the upper or the lower triangular part of the matrix A (see `upper_lower`). The second dimension of `a` must be at least $\max(1, n)$.

lda

The leading dimension of `a`.

scratchpad_size

Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *sytrf_scratchpad_size* function.

Output Parameters

a

The upper or lower triangular part of a is overwritten by details of the block-diagonal matrix D and the multipliers used to obtain the factor U (or L).

ipiv

Buffer, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of D. If $\text{ipiv}(i) = k > 0$, then d_{ii} is a 1×1 block, and the i-th row and column of A was interchanged with the k-th row and column.

If `upper_lower=oneapi::mkl::uplo::upper` and $\text{ipiv}(i) = \text{ipiv}(i-1) = -m < 0$, then D has a 2×2 block in rows/columns i and i-1, and $(i-1)$-th row and column of A was interchanged with the m-th row and column.

If `upper_lower=oneapi::mkl::uplo::lower` and $\text{ipiv}(i) = \text{ipiv}(i+1) = -m < 0$, then D has a 2×2 block in rows/columns i and $i+1$, and $(i+1)$-th row and column of A was interchanged with the m-th row and column.

scratchpad

Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If info $= i$, d_{ii} is 0. The factorization has been completed, but D is exactly singular. Division by 0 will occur if you use D for solving a system of linear equations.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

sytrf (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event sytrf(cl::sycl::queue &queue, oneapi::mkl::uplo upper_lower,␣
↪std::int64_t n, T *a, std::int64_t lda, int_64 *ipiv, T *scratchpad, std::int64_t␣
↪scratchpad_size, const std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If `upper_lower=uplo::upper`, the array `a` stores the upper triangular part of the matrix A, and A is factored as UDU^T.

If `upper_lower=uplo::lower`, the array `a` stores the lower triangular part of the matrix A, and A is factored as LDL^T.

n
The order of matrix A ($0 \leq n$).

a
The pointer to A, size $\max(1, \text{lda} \cdot n)$, containing either the upper or the lower triangular part of the matrix A (see `upper_lower`). The second dimension of `a` must be at least $\max(1, n)$.

lda
The leading dimension of `a`.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *sytrf_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a

The upper or lower triangular part of a is overwritten by details of the block-diagonal matrix D and the multipliers used to obtain the factor U (or L).

ipiv

Pointer to array of size at least $\max(1, n)$. Contains details of the interchanges and the block structure of D. If $\text{ipiv}(i) = k > 0$, then d_{ii} is a 1×1 block, and the i-th row and column of A was interchanged with the k-th row and column.

If `upper_lower=oneapi::mkl::uplo::upper` and $\text{ipiv}(i) = \text{ipiv}(i-1) = -m < 0$, then D has a 2×2 block in rows/columns i and $i-1$, and $(i-1)$-th row and column of A was interchanged with the m-th row and column.

If `upper_lower=oneapi::mkl::uplo::lower` and $\text{ipiv}(i) = \text{ipiv}(i+1) = -m < 0$, then D has a 2×2 block in rows/columns i and $i+1$, and $(i+1)$-th row and column of A was interchanged with the m-th row and column.

scratchpad

Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If info $= i$, d_{ii} is 0. The factorization has been completed, but D is exactly singular. Division by 0 will occur if you use D for solving a system of linear equations.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

sytrf_scratchpad_size

Computes size of scratchpad memory required for *sytrf* function.

Description

`sytrf_scratchpad_size` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

Computes the number of elements of type `T` the scratchpad memory to be passed to *sytrf* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

sytrf_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t sytrf_scratchpad_size(cl::sycl::queue &queue, oneapi::mkl::uplo upper_
↪lower, std::int64_t n, std::int64_t lda)
}
```

Input Parameters

queue
Device queue where calculations by *sytrf* function will be performed.

upper_lower
Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If `upper_lower=uplo::upper`, the buffer `a` stores the upper triangular part of the matrix A, and A is factored as UDU^T.

If `upper_lower=uplo::lower`, the buffer `a` stores the lower triangular part of the matrix A, and A is factored as LDL^T

n
The order of the matrix A ($0 \leq n$).

lda
The leading dimension of `a`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *sytrf* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

trtrs

Solves a system of linear equations with a triangular coefficient matrix, with multiple right-hand sides.

Description

`trtrs` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

The routine solves for X the following systems of linear equations with a triangular matrix A, with multiple right-hand sides stored in B:

$AX = B$	if `transa` =`transpose::nontrans`,
$A^T X = B$	if `transa` =`transpose::trans`,
$A^H X = B$	if `transa` =`transpose::conjtrans` (for complex matrices only).

trtrs (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void trtrs(cl::sycl::queue &queue, oneapi::mkl::uplo upper_lower,␣
↪oneapi::mkl::transpose transa, oneapi::mkl::diag unit_diag, std::int64_t n, std::int64_
↪t nrhs, cl::sycl::buffer<T,1> &a, std::int64_t lda, cl::sycl::buffer<T,1> &b,␣
↪std::int64_t ldb, cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Indicates whether A is upper or lower triangular:

If upper_lower = `uplo::upper`, then A is upper triangular.

If upper_lower = `uplo::lower`, then A is lower triangular.

transa
If transa = `transpose::nontrans`, then $AX = B$ is solved for X.

If transa = `transpose::trans`, then $A^T X = B$ is solved for X.

If transa = `transpose::conjtrans`, then $A^H X = B$ is solved for X.

unit_diag
If unit_diag = `diag::nonunit`, then A is not a unit triangular matrix.

If unit_diag = `diag::unit`, then A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array `a`.

n
The order of A; the number of rows in B; $n \geq 0$.

nrhs
The number of right-hand sides; nrhs ≥ 0.

a
Buffer containing the matrix A. The second dimension of `a` must be at least $\max(1, n)$.

lda
The leading dimension of `a`; lda $\geq \max(1, n)$.

b
Buffer containing the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of `b` at least $\max(1, \text{nrhs})$.

ldb
The leading dimension of `b`; ldb $\geq \max(1, n)$.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *trtrs_scratchpad_size* function.

Output Parameters

b
Overwritten by the solution matrix X.

scratchpad
Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

trtrs (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event trtrs(cl::sycl::queue &queue, oneapi::mkl::uplo upper_lower,␣
↪oneapi::mkl::transpose transa, oneapi::mkl::diag unit_diag, std::int64_t n, std::int64_
↪t nrhs, T *a, std::int64_t lda, T *b, std::int64_t ldb, T *scratchpad, std::int64_t␣
↪scratchpad_size, const std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Indicates whether A is upper or lower triangular:

If upper_lower = `uplo::upper`, then A is upper triangular.

If upper_lower = `uplo::lower`, then A is lower triangular.

transa
If transa = `transpose::nontrans`, then $AX = B$ is solved for X.

If transa = `transpose::trans`, then $A^T X = B$ is solved for X.

If transa = `transpose::conjtrans`, then $A^H X = B$ is solved for X.

unit_diag
If unit_diag = `diag::nonunit`, then A is not a unit triangular matrix.

If unit_diag = `diag::unit`, then A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array `a`.

n
The order of A; the number of rows in B; $n \geq 0$.

nrhs
The number of right-hand sides; nrhs ≥ 0.

a
Array containing the matrix A. The second dimension of `a` must be at least $\max(1, n)$.

lda
The leading dimension of `a`; lda $\geq \max(1, n)$.

b
Array containing the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of `b` at least $\max(1, \text{nrhs})$.

ldb
The leading dimension of `b`; ldb $\geq \max(1, n)$.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *trtrs_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

b
Overwritten by the solution matrix X.

scratchpad
Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

trtrs_scratchpad_size

Computes size of scratchpad memory required for *trtrs* function.

Description

`trtrs_scratchpad_size` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

Computes the number of elements of type `T` the scratchpad memory to be passed to *trtrs* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

trtrs_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t trtrs_scratchpad_size(cl::sycl::queue &queue, oneapi::mkl::uplo upper_
↪lower, oneapi::mkl::transpose trans, oneapi::mkl::diag diag, std::int64_t n,␣
↪std::int64_t nrhs, std::int64_t lda, std::int64_t ldb)
}
```

Input Parameters

queue
: Device queue where calculations by *trtrs* function will be performed.

upper_lower
: Indicates whether A is upper or lower triangular:

  If upper_lower = `uplo::upper`, then A is upper triangular.

  If upper_lower = `uplo::lower`, then A is lower triangular.

trans
: Indicates the form of the equations:

  If `trans=oneapi::mkl::transpose::nontrans`, then $AX = B$ is solved for X.

  If `trans=oneapi::mkl::transpose::trans`, then $A^T X = B$ is solved for X.

  If `trans=oneapi::mkl::transpose::conjtrans`, then $A^H X = B$ is solved for X.

diag
If diag = `oneapi::mkl::diag::nonunit`, then A is not a unit triangular matrix.

If unit_diag = `diag::unit`, then A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array `a`.

n
The order of A; the number of rows in B; $n \geq 0$.

nrhs
The number of right-hand sides ($0 \leq$ nrhs).

lda
The leading dimension of `a`; lda $\geq \max(1, n)$.

ldb
The leading dimension of `b`; ldb $\geq \max(1, n)$.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *trtrs* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

ungqr

Generates the complex unitary matrix Q of the QR factorization formed by *geqrf*.

Description

`ungqr` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

The routine generates the whole or part of $m \times m$ unitary matrix Q of the QR factorization formed by the routines *geqrf*.

Usually Q is determined from the QR factorization of an $m \times p$ matrix A with $m \geq p$. To compute the whole matrix Q, use:

```
oneapi::mkl::lapack::ungqr(queue, m, m, p, a, lda, tau, scratchpad, scratchpad_size)
```

To compute the leading p columns of Q (which form an orthonormal basis in the space spanned by the columns of A):

```
oneapi::mkl::lapack::ungqr(queue, m, p, p, a, lda, tau, scratchpad, scratchpad_size)
```

To compute the matrix Q^k of the QR factorization of the leading k columns of the matrix A:

```
oneapi::mkl::lapack::ungqr(queue, m, m, k, a, lda, tau, scratchpad, scratchpad_size)
```

To compute the leading k columns of Q^k (which form an orthonormal basis in the space spanned by the leading k columns of the matrix A):

```
oneapi::mkl::lapack::ungqr(queue, m, k, k, a, lda, tau, scratchpad, scratchpad_size)
```

ungqr (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void ungqr(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, std::int64_t k,␣
↪cl::sycl::buffer<T,1> &a, std::int64_t lda, cl::sycl::buffer<T,1> &tau,␣
↪cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

m
: The number of rows in the matrix A ($0 \leq m$).

n
: The number of columns in the matrix A ($0 \leq n$).

k
: The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

a
: The buffer `a` as returned by *geqrf*.

lda
: The leading dimension of `a` (lda $\leq m$).

tau
: The buffer `tau` as returned by *geqrf*.

scratchpad_size
: Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *ungqr_scratchpad_size* function.

Output Parameters

a
Overwritten by n leading columns of the $m \times m$ orthogonal matrix Q.

scratchpad
Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

ungqr (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event ungqr(cl::sycl::queue &queue, std::int64_t m, std::int64_t n,␣
↪std::int64_t k, T *a, std::int64_t lda, T *tau, T *scratchpad, std::int64_t scratchpad_
↪size, const std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

m
The number of rows in the matrix A ($0 \leq m$).

n
The number of columns in the matrix A ($0 \leq n$).

k
The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

a
The pointer to `a` as returned by *geqrf*.

lda
The leading dimension of `a` (lda $\leq m$).

tau
The pointer to `tau` as returned by *geqrf*.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *ungqr_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a
Overwritten by n leading columns of the $m \times m$ orthogonal matrix Q.

scratchpad
Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

ungqr_scratchpad_size

Computes size of scratchpad memory required for *ungqr* function.

Description

`ungqr_scratchpad_size` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *ungqr* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

ungqr_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t ungqr_scratchpad_size(cl::sycl::queue &queue, std::int64_t m, std::int64_
↪t n, std::int64_t k, std::int64_t lda)
}
```

Input Parameters

queue
: Device queue where calculations by *ungqr* function will be performed.

m
: The number of rows in the matrix A ($0 \leq m$).

n
: The number of columns the matrix A ($0 \leq n \leq m$).

k
: The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

lda
: The leading dimension of `a`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *ungqr* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

unmqr

Multiplies a complex matrix by the unitary matrix Q of the QR factorization formed by *geqrf*.

Description

`unmqr` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

The routine multiplies a rectangular complex $m \times n$ matrix C by Q or Q^H, where Q is the complex unitary matrix defined as a product of k elementary reflectors $H(i)$ of order n: $Q = H(1)^H H(2)^H ... H(k)^H$ as returned by the RQ factorization routine *gerqf*.

Depending on the parameters `side` and `trans`, the routine can form one of the matrix products QC, Q^HC, CQ, or CQ^H (overwriting the result over C).

unmqr (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void unmqr(cl::sycl::queue &queue, oneapi::mkl::side side, oneapi::mkl::transpose␣
↪trans, std::int64_t m, std::int64_t n, std::int64_t k, cl::sycl::buffer<T,1> &a,␣
↪std::int64_t lda, cl::sycl::buffer<T,1> &tau, cl::sycl::buffer<T,1> &c, std::int64_t␣
↪ldc, cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue
The queue where the routine should be executed.

side
If `side = oneapi::mkl::side::left`, Q or Q^H is applied to C from the left.

If `side = oneapi::mkl::side::right`, Q or Q^H is applied to C from the right.

trans
If `trans = oneapi::mkl::transpose::nontrans`, the routine multiplies C by Q.

If `trans = oneapi::mkl::transpose::conjtrans`, the routine multiplies C by Q^H.

m
The number of rows in the matrix C ($0 \leq m$).

n
The number of columns in the matrix C ($0 \leq n$).

k
The number of elementary reflectors whose product defines the matrix Q

If `side = oneapi::mkl::side::left`, $0 \leq k \leq m$

If `side = oneapi::mkl::side::right`, $0 \leq k \leq n$

a
The buffer `a` as returned by *geqrf*. The second dimension of `a` must be at least $\max(1, k)$.

lda
The leading dimension of `a`.

tau
The buffer `tau` as returned by *geqrf*.

c
The buffer `c` contains the matrix C. The second dimension of `c` must be at least $\max(1, n)$.

ldc
The leading dimension of `c`.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by the *unmqr_scratchpad_size* function.

Output Parameters

c
Overwritten by the product QC, Q^HC, CQ, or CQ^H (as specified by `side` and `trans`).

scratchpad
Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

unmqr (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event unmqr(cl::sycl::queue &queue, oneapi::mkl::side side,␣
↪oneapi::mkl::transpose trans, std::int64_t m, std::int64_t n, std::int64_t k, T *a,␣
↪std::int64_t lda, T *tau, T *c, std::int64_t ldc, T *scratchpad, std::int64_t␣
↪scratchpad_size, const std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

side
If `side = oneapi::mkl::side::left`, Q or Q^H is applied to C from the left.

If `side = oneapi::mkl::side::right`, Q or Q^H is applied to C from the right.

trans
If `trans = oneapi::mkl::transpose::nontrans`, the routine multiplies C by Q.

If `trans = oneapi::mkl::transpose::conjtrans`, the routine multiplies C by Q^H.

m
The number of rows in the matrix C ($0 \leq m$).

n
The number of columns in the matrix C ($0 \leq n$).

k
The number of elementary reflectors whose product defines the matrix Q

If `side = oneapi::mkl::side::left`, $0 \leq k \leq m$

If `side = oneapi::mkl::side::right`, $0 \leq k \leq n$

a
The pointer to `a` as returned by *geqrf*. The second dimension of `a` must be at least $\max(1, k)$.

lda
The leading dimension of `a`.

tau
The pointer to `tau` as returned by *geqrf*.

c
The pointer `c` points to the matrix C. The second dimension of `c` must be at least $\max(1, n)$.

ldc
The leading dimension of `c`.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *unmqr_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

c
Overwritten by the product QC, Q^HC, CQ, or CQ^H (as specified by `side` and `trans`).

scratchpad
Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

unmqr_scratchpad_size

Computes size of scratchpad memory required for *unmqr* function.

Description

`unmqr_scratchpad_size` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *unmqr* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

unmqr_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t unmqr_scratchpad_size(cl::sycl::queue &queue, oneapi::mkl::side side,␣
↪oneapi::mkl::transpose trans, std::int64_t m, std::int64_t n, std::int64_t k,␣
↪std::int64_t lda, std::int64_t ldc, std::int64_t &scratchpad_size)
}
```

Input Parameters

queue
: Device queue where calculations by *unmqr* function will be performed.

side
: If `side=oneapi::mkl::side::left`, Q or Q^H is applied to C from the left.

: If `side=oneapi::mkl::side::right`, Q or Q^H is applied to C from the right.

trans
: If `trans=oneapi::mkl::transpose::nontrans`, the routine multiplies C by Q.

: If `trans=oneapi::mkl::transpose::conjtrans`, the routine multiplies C by Q^H.

m
: The number of rows in the matrix C ($0 \leq m$).

n
: The number of columns the matrix C ($0 \leq n \leq m$).

k
The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

lda
The leading dimension of `a`.

ldc
The leading dimension of `c`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *unmqr* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

unmrq

Multiplies a complex matrix by the unitary matrix Q of the RQ factorization formed by *gerqf*.

Description

`unmrq` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

The routine multiplies a rectangular complex $m \times n$ matrix C by Q or Q^H, where Q is the complex unitary matrix defined as a product of k elementary reflectors $H(i)$ of order n: $Q = H(1)^H H(2)^H ... H(k)^H$ as returned by the RQ factorization routine *gerqf*.

Depending on the parameters `side` and `trans`, the routine can form one of the matrix products QC, Q^HC, CQ, or CQ^H (overwriting the result over C).

unmrq (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void unmrq(cl::sycl::queue &queue, oneapi::mkl::side side, oneapi::mkl::transpose␣
↪trans, std::int64_t m, std::int64_t n, std::int64_t k, cl::sycl::buffer<T,1> &a,␣
↪std::int64_t lda, cl::sycl::buffer<T,1> &tau, cl::sycl::buffer<T,1> &c, std::int64_t␣
↪ldc, cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue
The queue where the routine should be executed.

side
If `side = oneapi::mkl::side::left`, Q or Q^H is applied to C from the left.

If `side = oneapi::mkl::side::right`, Q or Q^H is applied to C from the right.

trans
If `trans = oneapi::mkl::transpose::nontrans`, the routine multiplies C by Q.

If `trans = oneapi::mkl::transpose::conjtrans`, the routine multiplies C by Q^H.

m
The number of rows in the matrix C ($0 \leq m$).

n
The number of columns in the matrix C ($0 \leq n$).

k
The number of elementary reflectors whose product defines the matrix Q

If `side = oneapi::mkl::side::left`, $0 \leq k \leq m$

If `side = oneapi::mkl::side::right`, $0 \leq k \leq n$

a
The buffer `a` as returned by *gerqf*. The second dimension of `a` must be at least $\max(1, k)$.

lda
The leading dimension of `a`.

tau
The buffer `tau` as returned by *gerqf*.

c
The buffer `c` contains the matrix C. The second dimension of `c` must be at least $\max(1, n)$.

ldc
The leading dimension of `c`.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *unmrq_scratchpad_size* function.

Output Parameters

c
Overwritten by the product QC, Q^HC, CQ, or CQ^H (as specified by `side` and `trans`).

scratchpad
Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

unmrq (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event unmrq(cl::sycl::queue &queue, oneapi::mkl::side side,␣
↪oneapi::mkl::transpose trans, std::int64_t m, std::int64_t n, std::int64_t k, T *a,␣
↪std::int64_t lda, T *tau, T *c, std::int64_t ldc, T *scratchpad, std::int64_t␣
↪scratchpad_size, const std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

side
If `side = oneapi::mkl::side::left`, Q or Q^H is applied to C from the left.

If `side = oneapi::mkl::side::right`, Q or Q^H is applied to C from the right.

trans
If `trans = oneapi::mkl::transpose::nontrans`, the routine multiplies C by Q.

If `trans = oneapi::mkl::transpose::conjtrans`, the routine multiplies C by Q^H.

m
The number of rows in the matrix C ($0 \leq m$).

n
The number of columns in the matrix C ($0 \leq n$).

k
The number of elementary reflectors whose product defines the matrix Q

If `side = oneapi::mkl::side::left`, $0 \leq k \leq m$

If `side = oneapi::mkl::side::right`, $0 \leq k \leq n$

a
The pointer to `a` as returned by *gerqf*. The second dimension of `a` must be at least $\max(1, k)$.

lda
The leading dimension of `a`.

tau
The pointer to `tau` as returned by *gerqf*.

c
The pointer `c` points to the matrix C. The second dimension of `c` must be at least $\max(1, n)$.

ldc
The leading dimension of `c`.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *unmrq_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

c
Overwritten by the product QC, Q^HC, CQ, or CQ^H (as specified by `side` and `trans`).

scratchpad
Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

unmrq_scratchpad_size

Computes size of scratchpad memory required for *unmrq* function.

Description

`unmrq_scratchpad_size` supports the following precisions.

T
std::complex<float>
std::complex<double>

Computes the number of elements of type `T` the scratchpad memory to be passed to *unmrq* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

unmrq_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t unmrq_scratchpad_size(cl::sycl::queue &queue, oneapi::mkl::side side,␣
↪oneapi::mkl::transpose trans, std::int64_t m, std::int64_t n, std::int64_t k,␣
↪std::int64_t lda, std::int64_t ldc)
}
```

Input Parameters

queue
Device queue where calculations by the unmrq function will be performed.

side
If `side = oneapi::mkl::side::left`, Q or Q^T is applied to C from the left. If `side = oneapi::mkl::side::right`, Q or Q^T is applied to C from the right.

trans
If `trans=oneapi::mkl::transpose::nontrans`, the routine multiplies C by Q.

If `trans=oneapi::mkl::transpose::conjtrans`, the routine multiplies C by Q^H.

m
The number of rows in the matrix C ($0 \leq m$).

n
The number of columns in the matrix C ($0 \leq n \leq m$).

k
The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

lda
The leading dimension of `a`.

ldc
The leading dimension of `c`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *unmrq* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

LAPACK Singular Value and Eigenvalue Problem Routines

LAPACK Singular Value and Eigenvalue Problem routines are used for singular value and eigenvalue problems, and for performing a number of related computational tasks. The following table lists the LAPACK Singular Value and Eigenvalue Problem routine groups.

Routines	Scratchpad Size Routines	Description
gebrd	*gebrd_scratchpad_s*	Reduces a general matrix to bidiagonal form.
gesvd	*gesvd_scratchpad*	Computes the singular value decomposition of a general rectangular matrix.
heevd	*heevd_scratchpa*	Computes all eigenvalues and, optionally, all eigenvectors of a complex Hermitian matrix using divide and conquer algorithm.
hegvd	*hegvd_scratchpa*	Computes all eigenvalues and, optionally, all eigenvectors of a complex generalized Hermitian definite eigenproblem using divide and conquer algorithm.
hetrd	*hetrd_scratchpad*	Reduces a complex Hermitian matrix to tridiagonal form.
orgbr	*orgbr_scratchpad*	Generates the real orthogonal matrix Q or P^T determined by gebrd.
orgtr	*orgtr_scratchpad*	Generates the real orthogonal matrix Q determined by sytrd.
ormtr	*ormtr_scratchpa*	Multiplies a real matrix by the orthogonal matrix Q determined by sytrd.
syevd	*syevd_scratchpad*	Computes all eigenvalues and, optionally, all eigenvectors of a real symmetric matrix using divide and conquer algorithm.
sygvd	*sygvd_scratchpad*	Computes all eigenvalues and, optionally, all eigenvectors of a real generalized symmetric definite eigenproblem using divide and conquer algorithm.
sytrd	*sytrd_scratchpad*	Reduces a real symmetric matrix to tridiagonal form.
ungbr	*ungbr_scratchpa*	Generates the complex unitary matrix Q or P^T determined by gebrd.
ungtr	*ungtr_scratchpad*	Generates the complex unitary matrix Q determined by hetrd.
unmtr	*unmtr_scratchpad_s*	Multiplies a complex matrix by the unitary matrix Q determined by hetrd.

gebrd

Reduces a general matrix to bidiagonal form.

Description

`gebrd` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

The routine reduces a general $m \times n$ matrix A to a bidiagonal matrix B by an orthogonal (unitary) transformation.

If $m \geq n$, the reduction is given by $A = QBP^H = \begin{pmatrix} B_1 \\ 0 \end{pmatrix} P^H = Q_1 B_1 P_H$

where B_1 is an $n \times n$ upper diagonal matrix, Q and P are orthogonal or, for a complex A, unitary matrices; Q_1 consists of the first n columns of Q.

If $m < n$, the reduction is given by

$$A = QBP^H = Q \begin{pmatrix} B_1 \\ 0 \end{pmatrix} P^H = Q_1 B_1 P_1^H,$$

where B_1 is an $m \times m$ lower diagonal matrix, Q and P are orthogonal or, for a complex A, unitary matrices; P_1 consists of the first m columns of P.

The routine does not form the matrices Q and P explicitly, but represents them as products of elementary reflectors. Routines are provided to work with the matrices Q and P in this representation:

If the matrix A is real,

- to compute Q and P explicitly, call *orgbr*.

If the matrix A is complex,

- to compute Q and P explicitly, call *ungbr*

gebrd (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void gebrd(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, cl::sycl::buffer<T,
↪1> &a, std::int64_t lda, cl::sycl::buffer<realT,1> &d, cl::sycl::buffer<realT,1> &e,␣
↪cl::sycl::buffer<T,1> &tauq, cl::sycl::buffer<T,1> &taup, cl::sycl::buffer<T,1> &
↪scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

m
: The number of rows in the matrix A ($0 \leq m$).

n
: The number of columns in the matrix A ($0 \leq n$).

a
: The buffer a, size (`lda,*`). The buffer `a` contains the matrix A. The second dimension of `a` must be at least $\max(1, m)$.

lda
: The leading dimension of a.

scratchpad_size
: Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *gebrd_scratchpad_size* function.

Output Parameters

a
: If $m \geq n$, the diagonal and first super-diagonal of a are overwritten by the upper bidiagonal matrix B. The elements below the diagonal, with the buffer tauq, represent the orthogonal matrix Q as a product of elementary reflectors, and the elements above the first superdiagonal, with the buffer `taup`, represent the orthogonal matrix P as a product of elementary reflectors.

  If $m < n$, the diagonal and first sub-diagonal of a are overwritten by the lower bidiagonal matrix B. The elements below the first subdiagonal, with the buffer tauq, represent the orthogonal matrix Q as a product of elementary reflectors, and the elements above the diagonal, with the buffer `taup`, represent the orthogonal matrix P as a product of elementary reflectors.

d
Buffer, size at least $\max(1, \min(m, n))$. Contains the diagonal elements of B.

e
Buffer, size at least $\max(1, \min(m, n) - 1)$. Contains the off-diagonal elements of B.

tauq
Buffer, size at least $\max(1, \min(m, n))$. The scalar factors of the elementary reflectors which represent the orthogonal or unitary matrix Q.

taup
Buffer, size at least $\max(1, \min(m, n))$. The scalar factors of the elementary reflectors which represent the orthogonal or unitary matrix P.

scratchpad
Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If `info=-i`, the `i`-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

gebrd (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event gebrd(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, T *a,␣
↪std::int64_t lda, RealT *d, RealT *e, T *tauq, T *taup, T *scratchpad, std::int64_t␣
↪scratchpad_size, const std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

m
The number of rows in the matrix A ($0 \leq m$).

n
The number of columns in the matrix A ($0 \leq n$).

a
Pointer to matrix A. The second dimension of `a` must be at least $\max(1, m)$.

lda
The leading dimension of `a`.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type T. Size should not be less than the value returned by *gebrd_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a

If $m \geq n$, the diagonal and first super-diagonal of a are overwritten by the upper bidiagonal matrix B. The elements below the diagonal, with the array tauq, represent the orthogonal matrix Q as a product of elementary reflectors, and the elements above the first superdiagonal, with the array `taup`, represent the orthogonal matrix P as a product of elementary reflectors.

If $m < n$, the diagonal and first sub-diagonal of a are overwritten by the lower bidiagonal matrix B. The elements below the first subdiagonal, with the array tauq, represent the orthogonal matrix Q as a product of elementary reflectors, and the elements above the diagonal, with the array `taup`, represent the orthogonal matrix P as a product of elementary reflectors.

d
Pointer to memory of size at least $\max(1, \min(m, n))$. Contains the diagonal elements of B.

e
Pointer to memory of size at least $\max(1, \min(m, n) - 1)$. Contains the off-diagonal elements of B.

tauq
Pointer to memory of size at least $\max(1, \min(m, n))$. The scalar factors of the elementary reflectors which represent the orthogonal or unitary matrix Q.

taup
Pointer to memory of size at least $\max(1, \min(m, n))$. The scalar factors of the elementary reflectors which represent the orthogonal or unitary matrix P.

scratchpad
Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

> Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:
>
> If `info=-i`, the `i`-th parameter had an illegal value.
>
> If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

gebrd_scratchpad_size

Computes size of scratchpad memory required for *gebrd* function.

Description

`gebrd_scratchpad_size` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *gebrd* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t gebrd_scratchpad_size(cl::sycl::queue &queue, std::int64_t m, std::int64_
↪t n, std::int64_t lda)
}
```

Input Parameters

queue
Device queue where calculations by *gebrd* function will be performed.

m
The number of rows in the matrix A ($0 \leq m$).

n
The number of columns in the matrix A ($0 \leq n$).

lda
The leading dimension of `a`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *gebrd* function should be able to hold.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

gesvd

Computes the singular value decomposition of a general rectangular matrix.

Description

`gesvd` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

gesvd (Buffer Version)

Description

The routine computes the singular value decomposition (SVD) of a real/complex $m \times n$ matrix A, optionally computing the left and/or right singular vectors. The SVD is written as

$A = U\Sigma V^T$ for real routines

$A = U\Sigma V^H$ for complex routines

where Σ is an $m \times n$ diagonal matrix, U is an $m \times m$ orthogonal/unitary matrix, and V is an $n \times n$ orthogonal/unitary matrix. The diagonal elements of Σ are the singular values of A; they are real and non-negative, and are returned in descending order. The first $\min(m, n)$ columns of U and V are the left and right singular vectors of A.

Syntax

```
namespace oneapi::mkl::lapack {
  void gesvd(cl::sycl::queue &queue, oneapi::mkl::job jobu, oneapi::mkl::job jobvt,␣
↪std::int64_t m, std::int64_t n, cl::sycl::buffer<T,1> &a, std::int64_t lda,␣
↪cl::sycl::buffer<realT,1> &s, cl::sycl::buffer<T,1> &u, std::int64_t ldu,␣
↪cl::sycl::buffer<T,1> &vt, std::int64_t ldvt, cl::sycl::buffer<T,1> &scratchpad,␣
↪std::int64_t scratchpad_size)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

jobu
: Must be `job::allvec`, `job::somevec`, `job::overwritevec`, or `job::novec`. Specifies options for computing all or part of the matrix U.

  If `jobu = job::allvec`, all m columns of U are returned in the buffer `u`;

  if `jobu = job::somevec`, the first $\min(m, n)$ columns of U (the left singular vectors) are returned in the buffer `u`;

  if `jobu = job::overwritevec`, the first $\min(m, n)$ columns of U (the left singular vectors) are overwritten on the buffer a;

  if `jobu = job::novec`, no columns of U (no left singular vectors) are computed.

jobvt

Must be `job::allvec, job::somevec`, `job::overwritevec`, or `job::novec`. Specifies options for computing all or part of the matrix V^T/V^H.

If `jobvt = job::allvec`, all n columns of V^T/V^H are returned in the buffer vt;

if `jobvt = job::somevec`, the first $\min(m,n)$ columns of V^T/V^H (the left singular vectors) are returned in the buffer vt;

if `jobvt = job::overwritevec`, the first $\min(m,n)$ columns of V^T/V^H (the left singular vectors) are overwritten on the buffer `a`;

if `jobvt = job::novec`, no columns of V^T/V^H (no left singular vectors) are computed.

`jobvt` and `jobu` cannot both be `job::overwritevec`.

m

The number of rows in the matrix A ($0 \leq m$).

a

The buffer `a`, size `(lda,*)`. The buffer `a` contains the matrix A. The second dimension of `a` must be at least $\max(1,m)$.

lda

The leading dimension of `a`.

ldu

The leading dimension of `u`.

ldvt

The leading dimension of `vt`.

scratchpad_size

Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *gesvd_scratchpad_size* function.

Output Parameters

a

On exit,

If `jobu = job::overwritevec`, `a` is overwritten with the first $\min(m,n)$ columns of U (the left singular vectors stored columnwise);

If `jobvt = job::overwritevec`, `a` is overwritten with the first $\min(m,n)$ rows of V^T/V^H (the right singular vectors stored rowwise);

If `jobu` $\neq$ `job::overwritevec` and `jobvt` $\neq$ `job::overwritevec`, the contents of a are destroyed.

s

Buffer containing the singular values, size at least $\max(1,\min(m,n))$. Contains the singular values of A sorted so that $s(i) \geq s(i+1)$.

u

Buffer containing U; the second dimension of `u` must be at least $\max(1,m)$ if `jobu = job::allvec`, and at least $\max(1,\min(m,n))$ if `jobu = job::somevec`.

If `jobu = job::allvec`, `u` contains the $m \times m$ orthogonal/unitary matrix U.

If `jobu = job::somevec`, `u` contains the first $\min(m,n)$ columns of U (the left singular vectors stored columnwise).

If `jobu = job::novec` or `job::overwritevec`, `u` is not referenced.

vt

Buffer containing V^T; the second dimension of `vt` must be at least $\max(1, n)$.

If `jobvt = job::allvec`, `vt` contains the $n \times n$ orthogonal/unitary matrix V^T/V^H.

If `jobvt = job::somevec`, `vt` contains the first $\min(m, n)$ rows of V^T/V^H (the right singular vectors stored row-wise).

If `jobvt = job::novec` or `job::overwritevec`, `vt` is not referenced.

scratchpad

Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If `info=-i`, the i-th parameter had an illegal value.

If `info=i`, then if `bdsqr` did not converge, i specifies how many superdiagonals of the intermediate bidiagonal form B did not converge to zero, and `scratchpad(2:min(m,n))` contains the unconverged superdiagonal elements of an upper bidiagonal matrix B whose diagonal is in `s` (not necessarily sorted). B satisfies $A = UBV^T$, so it has the same singular values as A, and singular vectors related by U and V^T.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

gesvd (USM Version)

Description

The routine computes the singular value decomposition (SVD) of a real/complex $m \times n$ matrix A, optionally computing the left and/or right singular vectors. The SVD is written as

$A = U\Sigma V^T$ for real routines

$A = U\Sigma V^H$ for complex routines

where Σ is an $m \times n$ diagonal matrix, U is an $m \times m$ orthogonal/unitary matrix, and V is an $n \times n$ orthogonal/unitary matrix. The diagonal elements of Σ are the singular values of A; they are real and non-negative, and are returned in descending order. The first $\min(m, n)$ columns of U and V are the left and right singular vectors of A.

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event gesvd(cl::sycl::queue &queue, oneapi::mkl::job jobu, oneapi::mkl::job␣
↪jobvt, std::int64_t m, std::int64_t n, T *a, std::int64_t lda, RealT *s, T *u,␣
↪std::int64_t ldu, T *vt, std::int64_t ldvt, T *scratchpad, std::int64_t scratchpad_
↪size, const std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

jobu
Must be `job::allvec`, `job::somevec`, `job::overwritevec`, or `job::novec`. Specifies options for computing all or part of the matrix U.

If `jobu = job::allvec`, all m columns of U are returned in the array `u`;

if `jobu = job::somevec`, the first $\min(m, n)$ columns of U (the left singular vectors) are returned in the array `u`;

if `jobu = job::overwritevec`, the first $\min(m, n)$ columns of U (the left singular vectors) are overwritten on the array a;

if `jobu = job::novec`, no columns of U (no left singular vectors) are computed.

jobvt
Must be `job::allvec, job::somevec`, `job::overwritevec`, or `job::novec`. Specifies options for computing all or part of the matrix V^T/V^H.

If `jobvt = job::allvec`, all n columns of V^T/V^H are returned in the array `vt`;

if `jobvt = job::somevec`, the first $\min(m, n)$ columns of V^T/V^H (the left singular vectors) are returned in the array vt;

if `jobvt = job::overwritevec`, the first $\min(m, n)$ columns of V^T/V^H (the left singular vectors) are overwritten on the array `a`;

if `jobvt = job::novec`, no columns of V^T/V^H (no left singular vectors) are computed.

`jobvt` and `jobu` cannot both be `job::overwritevec`.

m
The number of rows in the matrix A ($0 \leq m$).

a
Pointer to array `a`, size `(lda,*)`, containing the matrix A. The second dimension of `a` must be at least $\max(1, m)$.

lda
The leading dimension of `a`.

ldu
The leading dimension of `u`.

ldvt
The leading dimension of `vt`.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *gesvd_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a
On exit,

If `jobu = job::overwritevec`, `a` is overwritten with the first $\min(m, n)$ columns of U (the left singular vectors stored columnwise);

If `jobvt = job::overwritevec`, `a` is overwritten with the first $\min(m, n)$ rows of V^T/V^H (the right singular vectors stored rowwise);

If `jobu` $\neq$ `job::overwritevec` and `jobvt` $\neq$ `job::overwritevec`, the contents of a are destroyed.

s
Array containing the singular values, size at least $\max(1, \min(m, n))$. Contains the singular values of A sorted so that $s(i) \geq s(i + 1)$.

u
Array containing U; the second dimension of `u` must be at least $\max(1, m)$ if `jobu = job::allvec`, and at least $\max(1, \min(m, n))$ if `jobu = job::somevec`.

If `jobu = job::allvec`, `u` contains the $m \times m$ orthogonal/unitary matrix U.

If `jobu = job::somevec`, `u` contains the first $\min(m, n)$ columns of U (the left singular vectors stored column-wise).

If `jobu = job::novec` or `job::overwritevec`, `u` is not referenced.

vt
Array containing V^T; the second dimension of `vt` must be at least $\max(1, n)$.

If `jobvt = job::allvec`, `vt` contains the $n \times n$ orthogonal/unitary matrix V^T/V^H.

If `jobvt = job::somevec`, `vt` contains the first $\min(m, n)$ rows of V^T/V^H (the right singular vectors stored row-wise).

If `jobvt = job::novec` or `job::overwritevec`, `vt` is not referenced.

scratchpad
Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If `info=-i`, the i-th parameter had an illegal value.

If `info=i`, then if `bdsqr` did not converge, i specifies how many superdiagonals of the intermediate bidiagonal form B did not converge to zero, and `scratchpad(2:min(m,n))` contains the unconverged superdiagonal elements of an upper bidiagonal matrix B whose diagonal is in `s` (not necessarily sorted). B satisfies $A = UBV^T$, so it has the same singular values as A, and singular vectors related by U and V^T.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

gesvd_scratchpad_size

Computes size of scratchpad memory required for *gesvd* function.

Description

`gesvd_scratchpad_size` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

Computes the number of elements of type `T` the scratchpad memory to be passed to *gesvd* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

gesvd_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t gesvd_scratchpad_size(cl::sycl::queue &queue, oneapi::mkl::job jobu,␣
↪oneapi::mkl::job jobvt, std::int64_t m, std::int64_t n, std::int64_t lda, std::int64_t␣
↪ldu, std::int64_t ldvt)
}
```

Input Parameters

queue
Device queue where calculations by *gesvd* function will be performed.

jobu
Must be `job::allvec`, `job::somevec`, `job::overwritevec`, or `job::novec`. Specifies options for computing all or part of the matrix U.

If `jobu = job::allvec`, all m columns of U are returned in the buffer `u`;

if `jobu = job::somevec`, the first $\min(m, n)$ columns of U (the left singular vectors) are returned in the buffer `v`;

if `jobu = job::overwritevec`, the first $\min(m, n)$ columns of U (the left singular vectors) are overwritten on the buffer `a`;

if `jobu = job::novec`, no columns of U (no left singular vectors) are computed.

jobvt
Must be `job::allvec`, `job::somevec`, `job::overwritevec`, or `job::novec`. Specifies options for computing all or part of the matrix V^T/V^H.

If `jobvt = job::allvec`, all n columns of V^T/V^H are returned in the buffer `vt`;

if `jobvt = job::somevec`, the first $\min(m, n)$ columns of V^T/V^H (the left singular vectors) are returned in the buffer `vt`;

if `jobvt = job::overwritevec`, the first $\min(m, n)$ columns of V^T/V^H (the left singular vectors) are overwritten on the buffer `a`;

if `jobvt = job::novec`, no columns of V^T/V^H (no left singular vectors) are computed.

m
The number of rows in the matrix A ($0 \leq m$).

n
The number of columns in the matrix A ($0 \leq n$).

lda
The leading dimension of `a`.

ldu
The leading dimension of `u`.

ldvt
The leading dimension of `vt`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *gesvd* function should be able to hold.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

heevd

Computes all eigenvalues and, optionally, all eigenvectors of a complex Hermitian matrix using divide and conquer algorithm.

Description

`heevd` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a complex Hermitian matrix A. In other words, it can compute the spectral factorization of A as: $A = Z\Lambda Z^H$.

Here Λ is a real diagonal matrix whose diagonal elements are the eigenvalues λ_i, and Z is the (complex) unitary matrix whose columns are the eigenvectors z_i. Thus,

$Az_i = \lambda_i z_i$ for $i = 1, 2, ..., n$.

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

heevd (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void heevd(cl::sycl::queue &queue, oneapi::mkl::job jobz, oneapi::mkl::uplo upper_
↪lower, std::int64_t n, butter<T,1> &a, std::int64_t lda, cl::sycl::buffer<realT,1> &w,␣
↪cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

jobz
: Must be `job::novec` or `job::vec`.

: If `jobz = job::novec`, then only eigenvalues are computed.

: If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

upper_lower
Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = job::upper`, a stores the upper triangular part of A.

If `upper_lower = job::lower`, a stores the lower triangular part of A.

n
The order of the matrix A ($0 \leq n$).

a
The buffer `a`, size (`lda,*`). The buffer `a` contains the matrix A. The second dimension of `a` must be at least $\max(1, n)$.

lda
The leading dimension of `a`. Must be at least $\max(1, n)$.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *heevd_scratchpad_size* function.

Output Parameters

a
If `jobz = job::vec`, then on exit this buffer is overwritten by the unitary matrix Z which contains the eigenvectors of A.

w
Buffer, size at least n. Contains the eigenvalues of the matrix A in ascending order.

scratchpad
Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If `info=-i`, the i-th parameter had an illegal value.

If `info=i`, and `jobz = oneapi::mkl::job::novec`, then the algorithm failed to converge; i indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero.

If `info=i`, and `jobz = oneapi::mkl::job::vec`, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns info$/(n+1)$ through mod(info, $n+1$).

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

heevd (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event heevd(cl::sycl::queue &queue, oneapi::mkl::job jobz, oneapi::mkl::uplo␣
↪upper_lower, std::int64_t n, butter<T,1> &a, std::int64_t lda, RealT *w, T *scratchpad,
↪ std::int64_t scratchpad_size, const std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

jobz
Must be `job::novec` or `job::vec`.

If `jobz = job::novec`, then only eigenvalues are computed.

If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

upper_lower
Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = job::upper`, a stores the upper triangular part of A.

If `upper_lower = job::lower`, a stores the lower triangular part of A.

n
The order of the matrix A ($0 \leq n$).

a
Pointer to array containing A, size (`lda,*`).The second dimension of `a` must be at least $\max(1, n)$.

lda
The leading dimension of `a`. Must be at least $\max(1, n)$.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *heevd_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a
If `jobz = job::vec`, then on exit this array is overwritten by the unitary matrix Z which contains the eigenvectors of A.

w
Pointer to array of size at least n. Contains the eigenvalues of the matrix A in ascending order.

scratchpad
Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If `info=-i`, the i-th parameter had an illegal value.

If `info=i`, and `jobz = oneapi::mkl::job::novec`, then the algorithm failed to converge; i indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero.

If `info=i`, and `jobz = oneapi::mkl::job::vec`, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns info$/(n+1)$ through mod(info, $n+1$).

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

heevd_scratchpad_size

Computes size of scratchpad memory required for *heevd* function.

Description

`heevd_scratchpad_size` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *heevd* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

heevd_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t heevd_scratchpad_size(cl::sycl::queue &queue, oneapi::mkl::job jobz,␣
↪oneapi::mkl::uplo upper_lower, std::int64_t n, std::int64_t lda)
}
```

Input Parameters

queue
Device queue where calculations by *heevd* function will be performed.

jobz
Must be `job::novec` or `job::vec`.

If `jobz = job::novec`, then only eigenvalues are computed.

If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

upper_lower
Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = job::upper`, a stores the upper triangular part of A.

If `upper_lower = job::lower`, a stores the lower triangular part of A.

n
The order of the matrix A ($0 \leq n$).

lda
The leading dimension of `a`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *heevd* function should be able to hold.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

hegvd

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem using a divide and conquer method.

Description

`hegvd` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

$Ax = \lambda Bx$, $ABx = \lambda x$, or $BAx = \lambda x$.

Here A and B are assumed to be Hermitian and B is also positive definite.

It uses a divide and conquer algorithm.

hegvd (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void hegvd(cl::sycl::queue &queue, std::int64_t itype, oneapi::mkl::job jobz,␣
↪oneapi::mkl::uplo upper_lower, std::int64_t n, cl::sycl::buffer<T,1> &a, std::int64_t␣
↪lda, cl::sycl::buffer<T,1> &b, std::int64_t ldb, cl::sycl::buffer<realT,1> &w,␣
↪cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue
The queue where the routine should be executed.

itype
Must be 1 or 2 or 3. Specifies the problem type to be solved:

if itype $= 1$, the problem type is $Ax = \lambda Bx$;

if itype $= 2$, the problem type is $ABx = \lambda x$;

if itype $= 3$, the problem type is $BAx = \lambda x$.

jobz
Must be `job::novec` or `job::vec`.

If `jobz = job::novec`, then only eigenvalues are computed.

If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

upper_lower
Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = uplo::upper`, `a` and `b` store the upper triangular part of A and B.

If `upper_lower = uplo::lower`, `a` and `b` stores the lower triangular part of A and B.

n
The order of the matrices A and B ($0 \leq n$).

a
Buffer, size `a(lda,*)` contains the upper or lower triangle of the Hermitian matrix A, as specified by upper_lower.

The second dimension of `a` must be at least $\max(1, n)$.

lda
The leading dimension of `a`; at least $\max(1, n)$.

b
Buffer, size `b(ldb,*)` contains the upper or lower triangle of the Hermitian matrix B, as specified by upper_lower.

The second dimension of `b` must be at least $\max(1, n)$.

ldb
The leading dimension of `b`; at least $\max(1, n)$.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *hegvd_scratchpad_size* function.

Output Parameters

a

On exit, if `jobz = job::vec`, then if info $= 0$, `a` contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows:

if itype $= 1$ or itype $= 2$, $Z^H BZ = I$;

if itype $= 3$, $Z^H B^{-1} Z = I$;

If `jobz = job::novec`, then on exit the upper triangle (if `upper_lower = uplo::upper`) or the lower triangle (if `upper_lower = uplo::lower`) of A, including the diagonal, is destroyed.

b

On exit, if info $\leq n$, the part of `b` containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization $B = U^H U$ or $B = LL^H$.

w

Buffer, size at least n. If info $= 0$, contains the eigenvalues of the matrix A in ascending order.

scratchpad

Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

For info $\leq n$:

If info $= i$, and `jobz = oneapi::mkl::job::novec`, then the algorithm failed to converge; i indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero;

If info $= i$, and `jobz = oneapi::mkl::job::vec`, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns info$/(n+1)$‘ through mod(info, $n+1$).

For info $> n$:

If info $= n+i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

hegvd (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event hegvd(cl::sycl::queue &queue, std::int64_t itype, oneapi::mkl::job␣
↪jobz, oneapi::mkl::uplo upper_lower, std::int64_t n, T *a, std::int64_t lda, T *b,␣
↪std::int64_t ldb, RealT *w, T *scratchpad, std::int64_t scratchpad_size, const␣
↪std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

itype
Must be 1 or 2 or 3. Specifies the problem type to be solved:

if itype $= 1$, the problem type is $Ax = \lambda Bx$;

if itype $= 2$, the problem type is $ABx = \lambda x$;

if itype $= 3$, the problem type is $BAx = \lambda x$.

jobz
Must be `job::novec` or `job::vec`.

If `jobz = job::novec`, then only eigenvalues are computed.

If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

upper_lower
Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = uplo::upper`, `a` and `b` store the upper triangular part of A and B.

If `upper_lower = uplo::lower`, `a` and `b` stores the lower triangular part of A and B.

n
The order of the matrices A and B ($0 \leq n$).

a
Pointer to array of size `a(lda,*)` containing the upper or lower triangle of the Hermitian matrix A, as specified by upper_lower. The second dimension of `a` must be at least $\max(1, n)$.

lda
The leading dimension of `a`; at least $\max(1, n)$.

b
Pointer to array of size `b(ldb,*)` containing the upper or lower triangle of the Hermitian matrix B, as specified by upper_lower. The second dimension of `b` must be at least $\max(1, n)$.

ldb
The leading dimension of `b`; at least $\max(1, n)$.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *hegvd_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a
On exit, if `jobz = job::vec`, then if info $= 0$, `a` contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows:

if itype $= 1$‘ or itype $= 2$, $Z^H BZ = I$;

if itype $= 3$, $Z^H B^{-1} Z = I$;

If `jobz = job::novec`, then on exit the upper triangle (if `upper_lower = uplo::upper`) or the lower triangle (if `upper_lower = uplo::lower`) of A, including the diagonal, is destroyed.

b
On exit, if info $\leq n$, the part of `b` containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization $B = U^H U$or B = LL^H.

w
Pointer to array of size at least n. If info $= 0$, contains the eigenvalues of the matrix A in ascending order.

scratchpad
Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

For info $\leq n$:

If info $= i$, and `jobz = oneapi::mkl::job::novec`, then the algorithm failed to converge; i indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero;

If info $= i$, and `jobz = oneapi::mkl::job::vec`, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns info$/(n+1)$ through mod(info, $n+1$).

For info $> n$:

If info $= n+i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

hegvd_scratchpad_size

Computes size of scratchpad memory required for *hegvd* function.

Description

`hegvd_scratchpad_size` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *hegvd* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

hegvd_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t hegvd_scratchpad_size(cl::sycl::queue &queue, std::int64_t itype,␣
↪oneapi::mkl::job jobz, oneapi::mkl::uplo upper_lower, std::int64_t n, std::int64_t lda,
↪ std::int64_t ldb)
}
```

Input Parameters

queue
Device queue where calculations by *hegvd* function will be performed.

itype
Must be 1 or 2 or 3. Specifies the problem type to be solved:

if itype $= 1$, the problem type is $Ax = \lambda Bx$;

if itype $= 2$, the problem type is $ABx = \lambda x$;

if itype $= 3$, the problem type is $BAx = \lambda x$.

jobz
Must be `job::novec` or `job::vec`.

If `jobz = job::novec`, then only eigenvalues are computed.

If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

upper_lower
Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = uplo::upper`, `a` and `b` store the upper triangular part of A and B.

If `upper_lower = uplo::lower`, `a` and `b` store the lower triangular part of A and B.

n
The order of the matrices A and B ($0 \leq n$).

lda
The leading dimension of `a`. Currently `lda` is not referenced in this function.

ldb
The leading dimension of `b`. Currently `ldb` is not referenced in this function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *hegvd* function should be able to hold.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

hetrd

Reduces a complex Hermitian matrix to tridiagonal form.

Description

`hetrd` supports the following precisions.

Routine name	T
`chetrd`	`std::complex<float>`
`zhetrd`	`std::complex<double>`

The routine reduces a complex Hermitian matrix A to symmetric tridiagonal form T by a unitary similarity transformation: $A = QTQ^H$. The unitary matrix Q is not formed explicitly but is represented as a product of $n-1$ elementary reflectors. Routines are provided to work with Q in this representation.

hetrd (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void hetrd(cl::sycl::queue &queue, oneapi::mkl::uplo upper_lower, std::int64_t n,␣
↪cl::sycl::buffer<T,1> &a, std::int64_t lda, cl::sycl::buffer<realT,1> &d,␣
↪cl::sycl::buffer<realT,1> &e, cl::sycl::buffer<T,1> &tau, cl::sycl::buffer<T,1> &
↪scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = uplo::upper`, `a` stores the upper triangular part of A.

If `upper_lower = uplo::lower`, `a` stores the lower triangular part of A.

n
The order of the matrices A ($0 \leq n$).

a
Buffer, size `(lda,*)`. The buffer `a` contains either the upper or lower triangle of the Hermitian matrix A, as specified by upper_lower.

The second dimension of `a` must be at least $\max(1, n)$.

lda
The leading dimension of `a`; at least $\max(1, n)$

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *hetrd_scratchpad_size* function.

Output Parameters

a
On exit,

if `upper_lower = uplo::upper`, the diagonal and first superdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements above the first superdiagonal, with the buffer `tau`, represent the orthogonal matrix Q as a product of elementary reflectors;

if `upper_lower = uplo::lower`, the diagonal and first subdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements below the first subdiagonal, with the buffer `tau`, represent the orthogonal matrix Q as a product of elementary reflectors.

d
Buffer containing the diagonal elements of the matrix T. The dimension of `d` must be at least $\max(1, n)$.

e
Buffer containing the off diagonal elements of the matrix T. The dimension of `e` must be at least $\max(1, n-1)$.

tau
Buffer, size at least $\max(1, n-1)$. Stores $(n-1)$ scalars that define elementary reflectors in decomposition of the unitary matrix Q in a product of $n-1$ elementary reflectors.

scratchpad
Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

hetrd (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event hetrd(cl::sycl::queue &queue, oneapi::mkl::uplo upper_lower,␣
↪std::int64_t n, T *a, std::int64_t lda, RealT *d, RealT *e, T *tau, T *scratchpad,␣
↪std::int64_t scratchpad_size, const std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = uplo::upper`, `a` stores the upper triangular part of A.

If `upper_lower = uplo::lower`, `a` stores the lower triangular part of A.

n
The order of the matrices A ($0 \leq n$).

a
The pointer to matrix A, size `(lda,*)`. Contains either the upper or lower triangle of the Hermitian matrix A, as specified by `upper_lower`. The second dimension of `a` must be at least $\max(1, n)$.

lda
The leading dimension of `a`; at least $\max(1, n)$

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *hetrd_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a
On exit,

if `upper_lower = uplo::upper`, the diagonal and first superdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements above the first superdiagonal, with the array `tau`, represent the orthogonal matrix Q as a product of elementary reflectors;

if `upper_lower = uplo::lower`, the diagonal and first subdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements below the first subdiagonal, with the array `tau`, represent the orthogonal matrix Q as a product of elementary reflectors.

d
Pointer to diagonal elements of the matrix T. The dimension of `d` must be at least $\max(1, n)$.

e
Pointer to off diagonal elements of the matrix T. The dimension of `e` must be at least $\max(1, n-1)$.

tau
Pointer to array of size at least $\max(1, n-1)$. Stores $(n-1)$ scalars that define elementary reflectors in decomposition of the unitary matrix Q in a product of $n-1$ elementary reflectors.

scratchpad
Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

hetrd_scratchpad_size

Computes size of scratchpad memory required for *hetrd* function.

Description

`hetrd_scratchpad_size` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *hetrd* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

hetrd_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t hetrd_scratchpad_size(cl::sycl::queue &queue, oneapi::mkl::uplo upper_
↪lower, std::int64_t n, std::int64_t lda)
}
```

Input Parameters

queue
Device queue where calculations by *hetrd* function will be performed.

upper_lower
Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = uplo::upper`, `a` stores the upper triangular part of A and B.

If `upper_lower = uplo::lower`, `a` stores the lower triangular part of A.

n
The order of the matrices A and B ($0 \leq n$).

lda
The leading dimension of `a`. Currently, `lda` is not referenced in this function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *hetrd* function should be able to hold.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

orgbr

Generates the real orthogonal matrix Q or P^T determined by *gebrd*.

`orgbr` supports the following precisions.

T
float
double

Description

The routine generates the whole or part of the orthogonal matrices Q and P^T formed by the routines *gebrd*. All valid combinations of arguments are described in *Input parameters*. In most cases you need the following:

To compute the whole $m \times m$ matrix Q:

```
orgbr(queue, generate::q, m, m, n, a, ...)
```

(note that the array a must have at least m columns).

To form the n leading columns of Q if $m > n$:

```
orgbr(queue, generate::q, m, n, n, a, ...)
```

To compute the whole $n \times n$ matrix P^T:

```
orgbr(queue, generate::p, n, n, m, a, ...)
```

(note that the array a must have at least n rows).

To form the m leading rows of P^T if $m < n$:

```
orgbr(queue, generate::p, m, n, m, a, ...)
```

orgbr (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void orgbr(cl::sycl::queue &queue, oneapi::mkl::generate gen, std::int64_t m,␣
↪std::int64_t n, std::int64_t k, cl::sycl::buffer<T,1> &a, std::int64_t lda,␣
↪cl::sycl::buffer<T,1> &tau, cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_
↪size)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

gen
: Must be `generate::q` or `generate::p`.

: If `gen = generate::q`, the routine generates the matrix Q.

: If `gen = generate::p`, the routine generates the matrix P^T.

m

The number of rows in the matrix Q or P^T to be returned ($0 \leq m$).

If `gen = generate::q`, $m \leq n \leq \min(m, k)$.

If `gen = generate::p`, $n \leq m \leq \min(n, k)$.

n

The number of rows in the matrix Q or P^T to be returned ($0 \leq n$). See m for constraints.

k

If `gen = generate::q`, the number of columns in the original $m \times k$ matrix reduced by *gebrd*.

If `gen = generate::p`, the number of rows in the original $k \times n$ matrix reduced by *gebrd*.

a

The buffer `a` as returned by *gebrd*.

lda

The leading dimension of `a`.

tau

Buffer, size $\min(m, k)$ if `gen = generate::q`, size $\min(n, k)$ if `gen = generate::p`. Scalar factor of the elementary reflectors, as returned by *gebrd* in the array tauq or taup.

scratchpad_size

Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *orgbr_scratchpad_size* function.

Output Parameters

a

Overwritten by n leading columns of the $m \times m$ orthogonal matrix Q or P^T (or the leading rows or columns thereof) as specified by `gen`, `m`, and `n`.

scratchpad

Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

orgbr (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event orgbr(cl::sycl::queue &queue, oneapi::mkl::generate gen, std::int64_t␣
↪m, std::int64_t n, std::int64_t k, T *a, std::int64_t lda, T *tau, T *scratchpad,␣
↪std::int64_t scratchpad_size, const std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

gen
Must be `generate::q` or `generate::p`.

If `gen = generate::q`, the routine generates the matrix Q.

If `gen = generate::p`, the routine generates the matrix P^T.

m
The number of rows in the matrix Q or P^T to be returned ($0 \leq m$).

If `gen = generate::q`, $m \leq n \leq \min(m, k)$.

If `gen = generate::p`, $n \leq m \leq \min(n, k)$.

n
The number of rows in the matrix Q or P^T to be returned ($0 \leq n$). See m for constraints.

k
If `gen = generate::q`, the number of columns in the original $m \times k$ matrix reduced by *gebrd*.

If `gen = generate::p`, the number of rows in the original $k \times n$ matrix reduced by *gebrd*.

a
Pointer to array `a` as returned by *gebrd*.

lda
The leading dimension of `a`.

tau
Pointer to array of size $\min(m, k)$ if `gen = generate::q`, size $\min(n, k)$ if `gen = generate::p`. Scalar factor of the elementary reflectors, as returned by *gebrd* in the array tauq or taup.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *orgbr_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a

Overwritten by n leading columns of the $m \times m$ orthogonal matrix Q or P^T (or the leading rows or columns thereof) as specified by `gen`, `m`, and `n`.

scratchpad

Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

orgbr_scratchpad_size

Computes size of scratchpad memory required for *orgbr* function.

`orgbr_scratchpad_size` supports the following precisions.

T
float
double

Description

Computes the number of elements of type `T` the scratchpad memory to be passed to *orgbr* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

orgbr_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t orgbr_scratchpad_size(cl::sycl::queue &queue, oneapi::mkl::generate gen,␣
↪std::int64_t m, std::int64_t n, std::int64_t k, std::int64_t lda, std::int64_t &
↪scratchpad_size)
}
```

Input Parameters

queue
: Device queue where calculations by *orgbr* function will be performed.

gen
: Must be `generate::q` or `generate::p`.

: If `gen = generate::q`, the routine generates the matrix Q.

: If `gen = generate::p`, the routine generates the matrix P^T.

m
: The number of rows in the matrix Q or P^T to be returned ($0 \leq m$).

: If `gen = generate::q`, $m \leq n \leq \min(m, k)$.

: If `gen = generate::p`, $n \leq m \leq \min(n, k)$.

n
: The number of rows in the matrix Q or P^T to be returned ($0 \leq n$). See `m` for constraints.

k
: If `gen = generate::q`, the number of columns in the original $m \times k$ matrix returned by *gebrd*.

: If `gen = generate::p`, the number of rows in the original $k \times n$ matrix returned by *gebrd*.

lda
: The leading dimension of `a`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

: Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *orgbr* function should be able to hold.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

orgtr

Generates the real orthogonal matrix Q determined by *sytrd*.

Description

`orgtr` supports the following precisions.

T
float
double

The routine explicitly generates the $n \times n$ orthogonal matrix Q formed by *sytrd* when reducing a real symmetric matrix A to tridiagonal form: $A = QTQ^T$. Use this routine after a call to *sytrd*.

orgtr (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void orgtr(cl::sycl::queue &queue, oneapi::mkl::uplo upper_lower, std::int64_t n,␣
↪cl::sycl::buffer<T,1> &a, std::int64_t lda, cl::sycl::buffer<T,1> &tau,␣
↪cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

upper_lower
: Must be `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to *sytrd*.

n
: The order of the matrix Q ($0 \leq n$).

a
: The buffer `a` as returned by *sytrd*. The second dimension of `a` must be at least $\max(1, n)$.

lda
: The leading dimension of `a` ($n \leq$ lda).

tau
: The buffer `tau` as returned by *sytrd*. The dimension of `tau` must be at least $\max(1, n - 1)$.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *orgtr_scratchpad_size* function.

Output Parameters

a
Overwritten by the orthogonal matrix Q.

scratchpad
Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

orgtr (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event orgtr(cl::sycl::queue &queue, oneapi::mkl::uplo upper_lower,␣
↪std::int64_t n, T *a, std::int64_t lda, T *tau, T *scratchpad, std::int64_t scratchpad_
↪size, const std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Must be `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to *sytrd*.

n
The order of the matrix Q ($0 \leq n$).

a
The pointer to `a` as returned by *sytrd*. The second dimension of `a` must be at least $\max(1, n)$.

lda
The leading dimension of `a` ($n \leq$ lda).

tau
The pointer to `tau` as returned by *sytrd*. The dimension of `tau` must be at least $\max(1, n-1)$.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *orgtr_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a
Overwritten by the orthogonal matrix Q.

scratchpad
Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

orgtr_scratchpad_size

Computes size of scratchpad memory required for *orgtr* function.

Description

`orgtr_scratchpad_size` supports the following precisions.

T
`float`
`double`

Computes the number of elements of type `T` the scratchpad memory to be passed to *orgtr* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

orgtr_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t orgtr_scratchpad_size(cl::sycl::queue &queue, oneapi::mkl::uplo upper_
↪lower, std::int64_t n, std::int64_t lda)
}
```

Input Parameters

queue
: Device queue where calculations by *orgtr* function will be performed.

upper_lower
: Must be `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to *sytrd*.

n
: The order of the matrix Q ($0 \leq n$).

lda
: The leading dimension of `a` ($n \leq$ lda).

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *orgtr* function should be able to hold.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

ormtr

Multiplies a real matrix by the real orthogonal matrix Q determined by *sytrd*.

Description

`ormtr` supports the following precisions.

T
float
double

The routine multiplies a real matrix C by Q or Q^T, where Q is the orthogonal matrix Q formed by:ref:*onemkl_lapack_sytrd* when reducing a real symmetric matrix A to tridiagonal form: $A = QTQ^T$. Use this routine after a call to *sytrd*.

Depending on the parameters side and trans, the routine can form one of the matrix products QC, Q^TC, CQ, or CQ^T (overwriting the result on C).

ormtr (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void ormtr(cl::sycl::queue &queue, oneapi::mkl::side side, oneapi::mkl::uplo upper_
↪lower, oneapi::mkl::transpose trans, std::int64_t m, std::int64_t n, cl::sycl::buffer
↪<T,1> &a, std::int64_t lda, cl::sycl::buffer<T,1> &tau, cl::sycl::buffer<T,1> &c,␣
↪std::int64_t ldc, cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

In the descriptions below, `r` denotes the order of Q:

$r = m$	if `side = side::left`
$r = n$	if `side = side::right`

queue
The queue where the routine should be executed.

side
Must be either `side::left` or `side::right`.

If `side = side::left`, Q or Q^T is applied to C from the left.

If `side = side::right`, Q or Q^T is applied to C from the right.

upper_lower
Must be either `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to *sytrd*.

trans
Must be either `transpose::nontrans` or `transpose::trans`.

If `trans = transpose::nontrans`, the routine multiplies C by Q.

If `trans = transpose::trans`, the routine multiplies C by Q^T.

m
The number of rows in the matrix C ($m \geq 0$).

n
The number of columns in the matrix C ($n \geq 0$).

a
The buffer `a` as returned by *sytrd*.

lda
The leading dimension of `a` ($\max(1, r) \leq$ lda).

tau
The buffer `tau` as returned by a *sytrd*. The dimension of `tau` must be at least $\max(1, r - 1)$.

c
The buffer `c` contains the matrix C. The second dimension of `c` must be at least $\max(1, n)$.

ldc
The leading dimension of `c` ($\max(1, n) \leq$ ldc).

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *ormtr_scratchpad_size* function.

Output Parameters

c
Overwritten by the product QC, Q^TC, CQ, or CQ^T (as specified by `side` and `trans`).

scratchpad
Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

ormtr (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event ormtr(cl::sycl::queue &queue, oneapi::mkl::side side,␣
↪oneapi::mkl::uplo upper_lower, oneapi::mkl::transpose trans, std::int64_t m,␣
↪std::int64_t n, T *a, std::int64_t lda, T *tau, T *c, std::int64_t ldc, T *scratchpad,␣
↪std::int64_t scratchpad_size, const std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

In the descriptions below, `r` denotes the order of Q:

$r = m$	if `side = side::left`
$r = n$	if `side = side::right`

queue
The queue where the routine should be executed.

side
Must be either `side::left` or `side::right`.

If `side = side::left`, Q or Q^T is applied to C from the left.

If `side = side::right`, Q or Q^T is applied to C from the right.

upper_lower
Must be either `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to *sytrd*.

trans
Must be either `transpose::nontrans` or `transpose::trans`.

If `trans = transpose::nontrans`, the routine multiplies C by Q.

If `trans = transpose::trans`, the routine multiplies C by Q^T.

m
The number of rows in the matrix C ($m \geq 0$).

n
The number of columns in the matrix C ($n \geq 0$).

a
The pointer to `a` as returned by *sytrd*.

lda
The leading dimension of `a` ($\max(1, r) \leq$ lda).

tau
The buffer `tau` as returned by *sytrd*. The dimension of `tau` must be at least $\max(1, r - 1)$.

c
The pointer to memory containing the matrix C. The second dimension of `c` must be at least $\max(1, n)$.

ldc
The leading dimension of `c` ($\max(1, n) \leq$ ldc).

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *ormtr_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

c
Overwritten by the product QC, Q^TC, CQ, or CQ^T (as specified by `side` and `trans`).

scratchpad
Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

> Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:
>
> If info $= -i$, the i-th parameter had an illegal value.
>
> If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

ormtr_scratchpad_size

Computes size of scratchpad memory required for *ormtr* function.

Description

`ormtr_scratchpad_size` supports the following precisions.

T
`float`
`double`

Computes the number of elements of type `T` the scratchpad memory to be passed to *ormtr* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

ormtr_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t ormtr_scratchpad_size(cl::sycl::queue &queue, oneapi::mkl::side side,␣
↪oneapi::mkl::uplo upper_lower, oneapi::mkl::transpose trans, std::int64_t m,␣
↪std::int64_t n, std::int64_t lda, std::int64_t ldc)
}
```

Input Parameters

In the descriptions below, `r` denotes the order of Q:

$r = m$	if `side = side::left`
$r = n$	if `side = side::right`

queue
Device queue where calculations by *ormtr* function will be performed.

side
Must be either `side::left` or `side::right`.

If `side = side::left`, Q or Q^T is applied to C from the left.

If `side = side::right`, Q or Q^T is applied to C from the right.

upper_lower
Must be either `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to *sytrd*.

trans
Must be either `transpose::nontrans` or `transpose::trans`.

If `trans = transpose::nontrans`, the routine multiplies C by Q.

If `trans = transpose::trans`, the routine multiplies C by Q^T.

m
The number of rows in the matrix C ($m \geq 0$).

n
The number of rows in the matrix C ($n \geq 0$).

lda
The leading dimension of `a` ($\max(1, r) \leq$ lda).

ldc
The leading dimension of `c` ($\max(1, n) \leq$ ldc).

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *ormtr* function should be able to hold.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

syevd

Computes all eigenvalues and, optionally, all eigenvectors of a real symmetric matrix using divide and conquer algorithm.

Description

`syevd` supports the following precisions.

T
float
double

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric matrix A. In other words, it can compute the spectral factorization of A as: $A = Z\lambda Z^T$.

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i, and Z is the orthogonal matrix whose columns are the eigenvectors z_i. Thus,

$Az_i = \lambda_i z_i$ for $i = 1, 2, ..., n$.

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

syevd (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void syevd(cl::sycl::queue &queue, jobz jobz, oneapi::mkl::uplo upper_lower,␣
↪std::int64_t n, cl::sycl::buffer<T,1> &a, std::int64_t lda, cl::sycl::buffer<T,1> &w,␣
↪cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue
The queue where the routine should be executed.

jobz
Must be `job::novec` or `job::vec`.

If `jobz = job::novec`, then only eigenvalues are computed.

If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

upper_lower
Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = job::upper`, a stores the upper triangular part of A.

If `upper_lower = job::lower`, a stores the lower triangular part of A.

n
The order of the matrix A ($0 \leq n$).

a
The buffer `a`, size (`lda,*`). The buffer `a` contains the matrix A. The second dimension of `a` must be at least $\max(1, n)$.

lda
The leading dimension of `a`. Must be at least $\max(1, n)$.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *syevd_scratchpad_size* function.

Output Parameters

a
If `jobz = job::vec`, then on exit this buffer is overwritten by the orthogonal matrix Z which contains the eigenvectors of A.

w
Buffer, size at least n. Contains the eigenvalues of the matrix A in ascending order.

scratchpad
Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If info $= i$, and `jobz = oneapi::mkl::job::novec`, then the algorithm failed to converge; i indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero.

If info $= i$, and `jobz = oneapi::mkl::job::vec`, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns info$/(n+1)$ through mod(info, $n+1$).

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

syevd (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event syevd(cl::sycl::queue &queue, jobz jobz, oneapi::mkl::uplo upper_lower,
↪ std::int64_t n, T *a, std::int64_t lda, T *w, T *scratchpad, std::int64_t scratchpad_
↪size, const std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

jobz
Must be `job::novec` or `job::vec`.

If `jobz = job::novec`, then only eigenvalues are computed.

If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

upper_lower
Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = job::upper`, a stores the upper triangular part of A.

If `upper_lower = job::lower`, a stores the lower triangular part of A.

n
The order of the matrix A ($0 \leq n$).

a
Pointer to array containing A, size (`lda,*`). The second dimension of `a` must be at least $\max(1, n)$.

lda
The leading dimension of `a`. Must be at least $\max(1, n)$.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *syevd_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a
If `jobz = job::vec`, then on exit this array is overwritten by the orthogonal matrix Z which contains the eigenvectors of A.

w
Pointer to array of size at least n. Contains the eigenvalues of the matrix A in ascending order.

scratchpad
Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If info $= i$, and `jobz = oneapi::mkl::job::novec`, then the algorithm failed to converge; i indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero.

If info $= i$, and `jobz = oneapi::mkl::job::vec`, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns info$/(n+1)$ through mod(info, $n+1$).

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

syevd_scratchpad_size

Computes size of scratchpad memory required for *syevd* function.

Description

`syevd_scratchpad_size` supports the following precisions.

T
`float`
`double`

Computes the number of elements of type `T` the scratchpad memory to be passed to *syevd* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

syevd_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t syevd_scratchpad_size(cl::sycl::queue &queue, oneapi::mkl::job jobz,␣
↪oneapi::mkl::uplo upper_lower, std::int64_t n, std::int64_t lda)
}
```

Input Parameters

queue
: Device queue where calculations by *syevd* function will be performed.

jobz
: Must be `job::novec` or `job::vec`.

: If `jobz = job::novec`, then only eigenvalues are computed.

: If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

upper_lower
: Must be `uplo::upper` or `uplo::lower`.

: If `upper_lower = job::upper`, a stores the upper triangular part of A.

: If `upper_lower = job::lower`, a stores the lower triangular part of A.

n
: The order of the matrix A ($0 \leq n$).

lda

The leading dimension of `a`. Currently `lda` is not referenced in this function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *syevd* function should be able to hold.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

sygvd

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem using a divide and conquer method.

Description

`sygvd` supports the following precisions.

T
`float`
`double`

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$Ax = \lambda Bx$, $ABx = \lambda x$, or $BAx = \lambda x$.

Here A and B are assumed to be symmetric and B is also positive definite.

It uses a divide and conquer algorithm.

sygvd (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void sygvd(cl::sycl::queue &queue, std::int64_t itype, oneapi::mkl::job jobz,␣
↪oneapi::mkl::uplo upper_lower, std::int64_t n, cl::sycl::buffer<T,1> &a, std::int64_t␣
↪lda, cl::sycl::buffer<T,1> &b, std::int64_t ldb, cl::sycl::buffer<T,1> &w,␣
↪cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue
The queue where the routine should be executed.

itype
Must be 1 or 2 or 3. Specifies the problem type to be solved:

if itype $= 1$, the problem type is $Ax = \lambda Bx$;

if itype $= 2$, the problem type is $ABx = \lambda x$;

if itype $= 3$, the problem type is $BAx = \lambda x$.

jobz
Must be `job::novec` or `job::vec`.

If `jobz = job::novec`, then only eigenvalues are computed.

If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

upper_lower
Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = job::upper`, `a` and `b` store the upper triangular part of A and B.

If `upper_lower = job::lower`, `a` and `b` stores the lower triangular part of A and B.

n
The order of the matrices A and B ($0 \leq n$).

a
Buffer, size `a(lda,*)` contains the upper or lower triangle of the symmetric matrix A, as specified by `upper_lower`. The second dimension of `a` must be at least $\max(1, n)$.

lda
The leading dimension of `a`; at least $\max(1, n)$.

b
Buffer, size `b (ldb,*)` contains the upper or lower triangle of the symmetric matrix B, as specified by `upper_lower`. The second dimension of `b` must be at least $\max(1, n)$.

ldb
The leading dimension of `b`; at least $\max(1, n)$.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *sygvd_scratchpad_size* function.

Output Parameters

a

On exit, if `jobz = job::vec`, then if info $= 0$, `a` contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows:

if itype $= 1$ or 2 , $Z^T BZ = I$;

if itype $= 3$, $Z^T B^{-1} Z = I$;

If `jobz = job::novec`, then on exit the upper triangle (if `upper_lower = uplo::upper`) or the lower triangle (if `upper_lower = uplo::lower`) of A, including the diagonal, is destroyed.

b

On exit, if info $\leq n$, the part of `b` containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization $B = U^T U$ or $B = LL^T$.

w

Buffer, size at least n. If info $= 0$, contains the eigenvalues of the matrix A in ascending order.

scratchpad

Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

For info $\leq n$:

If info $= i$, and `jobz = oneapi::mkl::job::novec`, then the algorithm failed to converge; i indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero.

If info $= i$, and `jobz = oneapi::mkl::job::vec`, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns info$/(n+1)$ through mod(info, $n+1$).

For info $> n$:

If info $= n+i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

sygvd (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event sygvd(cl::sycl::queue &queue, std::int64_t itype, oneapi::mkl::job␣
↪jobz, oneapi::mkl::uplo upper_lower, std::int64_t n, T *a, std::int64_t lda, T *b,␣
↪std::int64_t ldb, T *w, T *scratchpad, std::int64_t scratchpad_size, const std::vector
↪<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

itype
Must be 1 or 2 or 3. Specifies the problem type to be solved:

if itype $= 1$, the problem type is $Ax = \lambda Bx$;

if itype $= 2$, the problem type is $ABx = \lambda x$;

if itype $= 3$, the problem type is $BAx = \lambda x$.

jobz
Must be `job::novec` or `job::vec`.

If `jobz = job::novec`, then only eigenvalues are computed.

If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

upper_lower
Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = job::upper`, `a` and `b` store the upper triangular part of A and B.

If `upper_lower = job::lower`, `a` and `b` stores the lower triangular part of A and B.

n
The order of the matrices A and B ($0 \leq n$).

a
Pointer to array of size `a(lda,*)` containing the upper or lower triangle of the symmetric matrix A, as specified by `upper_lower`. The second dimension of `a` must be at least $\max(1, n)$.

lda
The leading dimension of `a`; at least $\max(1, n)$.

b
Pointer to array of size `b (ldb,*)` contains the upper or lower triangle of the symmetric matrix B, as specified by `upper_lower`. The second dimension of `b` must be at least $\max(1, n)$.

ldb
The leading dimension of `b`; at least $\max(1, n)$.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *sygvd_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a
On exit, if `jobz = job::vec`, then if info $= 0$, a contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows:

if itype $= 1$ or 2, $Z^T BZ = I$;

if itype $= 3$, $Z^T B^{-1} Z = I$;

If `jobz = job::novec`, then on exit the upper triangle (if `upper_lower = uplo::upper`) or the lower triangle (if `upper_lower = uplo::lower`) of A, including the diagonal, is destroyed.

b
On exit, if info $\leq n$, the part of `b` containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization $B = U^T U$ or $B = LL^T$.

w
Pointer to array of size at least `n`. If info $= 0$, contains the eigenvalues of the matrix A in ascending order.

scratchpad
Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

For info $\leq n$:

If info $= i$, and `jobz = oneapi::mkl::job::novec`, then the algorithm failed to converge; i indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero.

If info $= i$, and `jobz = oneapi::mkl::job::vec`, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns info$/(n+1)$ through mod(info, $n+1$).

For info $> n$:

If info $= n+i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

sygvd_scratchpad_size

Computes size of scratchpad memory required for *sygvd* function.

Description

sygvd_scratchpad_size ` supports the following precisions.

T
float
double

Computes the number of elements of type `T` the scratchpad memory to be passed to *sygvd* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

sygvd_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t sygvd_scratchpad_size(cl::sycl::queue &queue, std::int64_t itype,␣
↪oneapi::mkl::job jobz, oneapi::mkl::uplo upper_lower, std::int64_t n, std::int64_t lda,
↪ std::int64_t ldb)
}
```

Input Parameters

queue
Device queue where calculations by *sygvd* function will be performed.

itype
Must be 1 or 2 or 3. Specifies the problem type to be solved:

if itype $= 1$, the problem type is $Ax = \lambda Bx$;

if itype $= 2$, the problem type is $ABx = \lambda x$;

if itype $= 3$, the problem type is $BAx = \lambda x$.

jobz
Must be `job::novec` or `job::vec`.

If `jobz = job::novec`, then only eigenvalues are computed.

If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

upper_lower
Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = job::upper`, `a` and `b` store the upper triangular part of A and B.

If `upper_lower = job::lower`, `a` and `b` stores the lower triangular part of A and B.

n
The order of the matrices A and B ($0 \leq n$).

lda
The leading dimension of `a`.

ldb
The leading dimension of `b`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *sygvd* function should be able to hold.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

sytrd

Reduces a real symmetric matrix to tridiagonal form.

Description

`sytrd` supports the following precisions.

T
`float`
`double`

The routine reduces a real symmetric matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation: $A = QTQ^T$. The orthogonal matrix Q is not formed explicitly but is represented as a product of $n - 1$ elementary reflectors. Routines are provided for working with Q in this representation .

sytrd (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void sytrd(cl::sycl::queue &queue, oneapi::mkl::uplo upper_lower, std::int64_t n,␣
↪cl::sycl::buffer<T,1> &a, std::int64_t lda, cl::sycl::buffer<T,1> &d, cl::sycl::buffer
↪<T,1> &e, cl::sycl::buffer<T,1> &tau, cl::sycl::buffer<T,1> &scratchpad, std::int64_t␣
↪scratchpad_size)
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = uplo::upper`, `a` stores the upper triangular part of A.

If `upper_lower = uplo::lower`, `a` stores the lower triangular part of A.

n
The order of the matrices A ($0 \leq n$).

a
The buffer `a`, size `(lda,*)`. Contains the upper or lower triangle of the symmetric matrix A, as specified by `upper_lower`.

The second dimension of `a` must be at least $\max(1, n)$.

lda
The leading dimension of `a`; at least $\max(1, n)$.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *sytrd_scratchpad_size* function.

Output Parameters

a
On exit,

if `upper_lower = uplo::upper`, the diagonal and first superdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements above the first superdiagonal, with the buffer `tau`, represent the orthogonal matrix Q as a product of elementary reflectors;

if `upper_lower = uplo::lower`, the diagonal and first subdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements below the first subdiagonal, with the buffer `tau`, represent the orthogonal matrix Q as a product of elementary reflectors.

d
Buffer containing the diagonal elements of the matrix T. The dimension of `d` must be at least $\max(1, n)$.

e
Buffer containing the off diagonal elements of the matrix T. The dimension of `e` must be at least $\max(1, n-1)$.

tau
Buffer, size at least $\max(1, n)$. Stores $(n-1)$ scalars that define elementary reflectors in decomposition of the unitary matrix Q in a product of $n-1$ elementary reflectors. $\tau(n)$ is used as workspace.

scratchpad
Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

sytrd (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event sytrd(cl::sycl::queue &queue, oneapi::mkl::uplo upper_lower,␣
↪std::int64_t n, T *a, std::int64_t lda, T *d, T *e, T *tau, T *scratchpad, std::int64_
↪t scratchpad_size, const std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = uplo::upper`, `a` stores the upper triangular part of A.

If `upper_lower = uplo::lower`, `a` stores the lower triangular part of A.

n
The order of the matrices A ($0 \leq n$).

a
The pointer to matrix A, size `(lda,*)`. Contains the upper or lower triangle of the symmetric matrix A, as specified by `upper_lower`. The second dimension of `a` must be at least $\max(1, n)$.

lda
The leading dimension of `a`; at least $\max(1, n)$.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *sytrd_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a
On exit,

if `upper_lower = uplo::upper`, the diagonal and first superdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements above the first superdiagonal, with the array `tau`, represent the orthogonal matrix Q as a product of elementary reflectors;

if `upper_lower = uplo::lower`, the diagonal and first subdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements below the first subdiagonal, with the array `tau`, represent the orthogonal matrix Q as a product of elementary reflectors.

d
Pointer to diagonal elements of the matrix T. The dimension of `d` must be at least $\max(1, n)$.

e
Pointer to off diagonal elements of the matrix T. The dimension of `e` must be at least $\max(1, n - 1)$.

tau
Pointer to array of size at least $\max(1, n)$. Stores $(n - 1)$ scalars that define elementary reflectors in decomposition of the unitary matrix Q in a product of $n - 1$ elementary reflectors. $\tau(n)$ is used as workspace.

scratchpad
Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

> Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:
>
> If info $= -i$, the i-th parameter had an illegal value.
>
> If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

sytrd_scratchpad_size

Computes size of scratchpad memory required for *sytrd* function.

Description

`sytrd_scratchpad_size` supports the following precisions.

T
`float`
`double`

Computes the number of elements of type `T` the scratchpad memory to be passed to *sytrd* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

sytrd_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t sytrd_scratchpad_size(cl::sycl::queue &queue, oneapi::mkl::uplo upper_
↪lower, std::int64_t n, std::int64_t lda)
}
```

Input Parameters

queue
Device queue where calculations by *sytrd* function will be performed.

upper_lower
Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = uplo::upper`, a stores the upper triangular part of A.

If `upper_lower = uplo::lower`, a stores the lower triangular part of A.

n
The order of the matrices A ($0 \leq n$).

lda
The leading dimension of `a`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *sytrd* function should be able to hold.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

ungbr

Generates the complex unitary matrix Q or P^t determined by *gebrd*.

Description

`ungbr` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

The routine generates the whole or part of the unitary matrices Q and P^H formed by the routines *gebrd*. All valid combinations of arguments are described in *Input Parameters*; in most cases you need the following:

To compute the whole $m \times m$ matrix Q, use:

```
oneapi::mkl::lapack::ungbr(queue, generate::q, m, m, n, a, ...)
```

(note that the buffer `a` must have at least m columns).

To form the n leading columns of Q if $m > n$, use:

```
oneapi::mkl::lapack::ungbr(queue, generate::q, m, n, n, a, ...)
```

To compute the whole $n \times n$ matrix P^T, use:

```
oneapi::mkl::lapack::ungbr(queue, generate::p, n, n, m, a, ...)
```

(note that the array `a` must have at least n rows).

To form the m leading rows of P^T if $m < n$, use:

```
oneapi::mkl::lapack::ungbr(queue, generate::p, m, n, m, a, ...)
```

ungbr (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void ungbr(cl::sycl::queue &queue, oneapi::mkl::generate gen, std::int64_t m,␣
↪std::int64_t n, std::int64_t k, cl::sycl::buffer<T,1> &a, std::int64_t lda,␣
↪cl::sycl::buffer<T,1> &tau, cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_
↪size)
}
```

Input Parameters

queue
: The queue where the routine should be executed.

gen
: Must be `generate::q` or `generate::p`.

: If `gen = generate::q`, the routine generates the matrix Q.

: If `gen = generate::p`, the routine generates the matrix P^T.

m

The number of rows in the matrix Q or P^T to be returned $(0 \leq m)$.

If `gen = generate::q`, $m \geq n \geq \min(m, k)$.

If `gen = generate::p`, $n \geq m \geq \min(n, k)$.

n

The number of columns in the matrix Q or P^T to be returned $(0 \leq n)$. See `m` for constraints.

k

If `gen = generate::q`, the number of columns in the original $m \times k$ matrix returned by *gebrd*.

If `gen = generate::p`, the number of rows in the original $k \times n$ matrix returned by *gebrd*.

a

The buffer `a` as returned by *gebrd*.

lda

The leading dimension of `a`.

tau

For `gen = generate::q`, the array `tauq` as returned by *gebrd*. For `gen = generate::p`, the array `taup` as returned by *gebrd*.

The dimension of `tau` must be at least $\max(1, \min(m, k))$ for `gen = generate::q`, or $\max(1, \min(m, k))$ for `gen = generate::p`.

scratchpad_size

Size of scratchpad memory as a number of floating point elements of type T. Size should not be less than the value returned by *ungbr_scratchpad_size* function.

Output Parameters

a

Overwritten by n leading columns of the $m \times m$ unitary matrix Q or P^T, (or the leading rows or columns thereof) as specified by `gen`, `m`, and `n`.

scratchpad

Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If $info = -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

ungbr (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event ungbr(cl::sycl::queue &queue, oneapi::mkl::generate gen, std::int64_t␣
↪m, std::int64_t n, std::int64_t k, T *a, std::int64_t lda, T *tau, T *scratchpad,␣
↪std::int64_t scratchpad_size, const std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

gen
Must be `generate::q` or `generate::p`.

If `gen = generate::q`, the routine generates the matrix Q.

If `gen = generate::p`, the routine generates the matrix P^T.

m
The number of rows in the matrix Q or P^T to be returned $(0 \geq m)$.

If `gen = generate::q`, $m \geq n \geq \min(m, k)$.

If `gen = generate::p`, $n \geq m \geq \min(n, k)$.

n
The number of columns in the matrix Q or P^T to be returned $(0 \leq n)$. See `m` for constraints.

k
If `gen = generate::q`, the number of columns in the original $m \times k$ matrix returned by *gebrd*.

If `gen = generate::p`, the number of rows in the original $k \times n$ matrix returned by *gebrd*.

a
The pointer to `a` as returned by *gebrd*.

lda
The leading dimension of `a`.

tau
For `gen = generate::q`, the array `tauq` as returned by *gebrd*. For `gen = generate::p`, the array `taup` as returned by *gebrd*.

The dimension of `tau` must be at least $\max(1, \min(m, k))$ for `gen = generate::q`, or $\max(1, \min(m, k))$ for `gen = generate::p`.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type T. Size should not be less than the value returned by *ungbr_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a
Overwritten by n leading columns of the $m \times m$ unitary matrix Q or P^T, (or the leading rows or columns thereof) as specified by `gen`, `m`, and `n`.

scratchpad
Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If $info = -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

ungbr_scratchpad_size

Computes size of scratchpad memory required for *ungbr* function.

Description

`ungbr_scratchpad_size` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type T the scratchpad memory to be passed to *ungbr* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

ungbr_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t ungbr_scratchpad_size(cl::sycl::queue &queue, oneapi::mkl::generate gen,␣
↪std::int64_t m, std::int64_t n, std::int64_t k, std::int64_t lda, std::int64_t &
↪scratchpad_size)
}
```

Input Parameters

queue
: Device queue where calculations by *ungbr* function will be performed.

gen
: Must be `generate::q` or `generate::p`.

: If `gen = generate::q`, the routine generates the matrix Q.

: If `gen = generate::p`, the routine generates the matrix P^T.

m
: The number of rows in the matrix Q or P^T to be returned $(0 \leq m)$.

: If `gen = generate::q`, $m \geq n \geq \min(m, k)$.

: If `gen = generate::p`, $n \geq m \geq \min(n, k)$.

n
: The number of columns in the matrix Q or P^T to be returned $(0 \leq n)$. See m for constraints.

k
: If `gen = generate::q`, the number of columns in the original $m \times k$ matrix reduced by *gebrd*.

: If `gen = generate::p`, the number of rows in the original $k \times n$ matrix reduced by *gebrd*.

lda
: The leading dimension of `a`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *ungbr* function should be able to hold.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

ungtr

Generates the complex unitary matrix Q determined by *hetrd*.

Description

`ungtr` supports the following precisions.

T
std::complex<float>
std::complex<double>

The routine explicitly generates the $n \times n$ unitary matrix Q formed by *hetrd* when reducing a complex Hermitian matrix A to tridiagonal form: $A = QTQ^H$. Use this routine after a call to *hetrd*.

ungtr (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void ungtr(cl::sycl::queue &queue, oneapi::mkl::uplo upper_lower, std::int64_t n,␣
↪cl::sycl::buffer<T,1> &a, std::int64_t lda, cl::sycl::buffer<T,1> &tau,␣
↪cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Must be `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to *hetrd*.

n
The order of the matrix Q ($0 \leq n$).

a
The buffer `a` as returned by *hetrd*. The second dimension of `a` must be at least $\max(1, n)$.

lda
The leading dimension of `a` ($n \leq$ lda).

tau
The buffer `tau` as returned by *hetrd*. The dimension of `tau` must be at least $\max(1, n - 1)$.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *ungtr_scratchpad_size* function.

Output Parameters

a
Overwritten by the unitary matrix Q.

scratchpad
Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

ungtr (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event ungtr(cl::sycl::queue &queue, oneapi::mkl::uplo upper_lower,␣
↪std::int64_t n, T *a, std::int64_t lda, T *tau, T *scratchpad, std::int64_t scratchpad_
↪size, const std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
The queue where the routine should be executed.

upper_lower
Must be `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to *hetrd*.

n
The order of the matrix Q ($0 \leq n$).

a
The pointer to `a` as returned by *hetrd*. The second dimension of `a` must be at least $\max(1, n)$.

lda
The leading dimension of `a` ($n \leq$ lda).

tau
The pointer to `tau` as returned by *hetrd*. The dimension of `tau` must be at least $\max(1, n-1)$.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *ungtr_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a
Overwritten by the unitary matrix Q.

scratchpad
Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

> Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:
>
> If info $= -i$, the i-th parameter had an illegal value.
>
> If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

ungtr_scratchpad_size

Computes size of scratchpad memory required for *ungtr* function.

Description

`ungtr_scratchpad_size` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *ungtr* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

ungtr_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t ungtr_scratchpad_size(cl::sycl::queue &queue, oneapi::mkl::uplo upper_
↪lower, std::int64_t n, std::int64_t lda)
}
```

Input Parameters

queue
Device queue where calculations by *ungtr* function will be performed.

upper_lower
Must be `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to *hetrd*.

n
The order of the matrix Q $(0 \leq n)$.

lda
The leading dimension of `a` $(n \leq lda)$.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type T the scratchpad memory to be passed to *ungtr* function should be able to hold.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

unmtr

Multiplies a complex matrix by the complex unitary matrix Q determined by *hetrd*.

Description

`unmtr` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

The routine multiplies a complex matrix C by Q or Q^H, where Q is the unitary matrix Q formed by *hetrd* when reducing a complex Hermitian matrix A to tridiagonal form: $A = QTQ^H$. Use this routine after a call to *hetrd*.

Depending on the parameters `side` and `trans`, the routine can form one of the matrix products QC, Q^HC, CQ, or CQ^H (overwriting the result on C).

unmtr (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void unmtr(cl::sycl::queue &queue, oneapi::mkl::side side, oneapi::mkl::uplo upper_
↪lower, oneapi::mkl::transpose trans, std::int64_t m, std::int64_t n, cl::sycl::buffer
↪<T,1> &a, std::int64_t lda, cl::sycl::buffer<T,1> &tau, cl::sycl::buffer<T,1> &c,␣
↪std::int64_t ldc, cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

In the descriptions below, `r` denotes the order of Q:

r=m	if `side = side::left`
r=n	if `side = side::right`

queue
: The queue where the routine should be executed.

side
: Must be either `side::left` or `side::right`.

: If `side=side::left`, Q or Q^H is applied to C from the left.

: If `side=side::right`, Q or Q^H is applied to C from the right.

upper_lower
: Must be either `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to *hetrd*.

trans
: Must be either `transpose::nontrans` or `transpose::conjtrans`.

: If `trans=transpose::nontrans`, the routine multiplies C by Q.

: If `trans=transpose::conjtrans`, the routine multiplies C by Q^H.

m
: The number of rows in the matrix C ($m \geq 0$).

n
The number of columns the matrix C ($n \geq 0$).

k
The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

a
The buffer `a` as returned by *hetrd*.

lda
The leading dimension of `a` ($\max(1, r) \leq$ lda).

tau
The buffer `tau` as returned by *hetrd*. The dimension of `tau` must be at least $\max(1, r - 1)$.

c
The buffer `c` contains the matrix C. The second dimension of `c` must be at least $\max(1, n)$.

ldc
The leading dimension of `c` ($\max(1, n) \leq$ ldc).

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *unmtr_scratchpad_size* function.

Output Parameters

c
Overwritten by the product QC, $Q^H C$, CQ, or CQ^H (as specified by `side` and `trans`).

scratchpad
Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If $info = -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

unmtr (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event unmtr(cl::sycl::queue &queue, oneapi::mkl::side side,␣
↪oneapi::mkl::uplo upper_lower, oneapi::mkl::transpose trans, std::int64_t m,␣
↪std::int64_t n, T *a, std::int64_t lda, T *tau, T *c, std::int64_t ldc, T *scratchpad,␣
↪std::int64_t scratchpad_size, const std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

In the descriptions below, `r` denotes the order of Q:

r=m	if `side = side::left`
r=n	if `side = side::right`

queue
: The queue where the routine should be executed.

side
: Must be either `side::left` or `side::right`.

: If `side=side::left`, Q or Q^H is applied to C from the left.

: If `side=side::right`, Q or Q^H is applied to C from the right.

upper_lower
: Must be either `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to *hetrd*.

trans
: Must be either `transpose::nontrans` or `transpose::conjtrans`.

: If `trans=transpose::nontrans`, the routine multiplies C by Q.

: If `trans=transpose::conjtrans`, the routine multiplies C by Q^H.

m
: The number of rows in the matrix C ($m \geq 0$).

n
: The number of columns the matrix C ($n \geq 0$).

k
: The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

a
: The pointer to `a` as returned by *hetrd*.

lda
: The leading dimension of `a` ($\max(1, r) \leq$ lda).

tau
: The pointer to `tau` as returned by *hetrd*. The dimension of `tau` must be at least $\max(1, r-1)$.

c
: The array `c` contains the matrix C. The second dimension of `c` must be at least $\max(1, n)$.

ldc
The leading dimension of `c` ($\max(1, n) \leq$ ldc).

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *unmtr_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

c
Overwritten by the product QC, Q^HC, CQ, or CQ^H (as specified by `side` and trans).

scratchpad
Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If $info = -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

unmtr_scratchpad_size

Computes size of scratchpad memory required for *unmtr* function.

Description

`unmtr_scratchpad_size` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *unmtr* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

unmtr_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t unmtr_scratchpad_size(cl::sycl::queue &queue, oneapi::mkl::side side,␣
↪oneapi::mkl::uplo upper_lower, oneapi::mkl::transpose trans, std::int64_t m,␣
↪std::int64_t n, std::int64_t lda, std::int64_t ldc)
}
```

Input Parameters

queue
Device queue where calculations by *unmtr* function will be performed.

side
Must be either `side::left` or `side::right`.

If `side=side::left`, Q or Q^H is applied to C from the left.

If `side=side::right`, Q or Q^H is applied to C from the right.

upper_lower
Must be either `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to *hetrd*.

trans
Must be either `transpose::nontrans` or `transpose::conjtrans`.

If `trans=transpose::nontrans`, the routine multiplies C by Q.

If `trans=transpose::conjtrans`, the routine multiplies C by Q^H.

m
The number of rows in the matrix C ($m \geq 0$).

n
The number of columns the matrix C ($n \geq 0$).

k
The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

lda
The leading dimension of a ($\max(1, r) \leq$ lda).

ldc

The leading dimension of c ($\max(1, n) \leq$ ldc).

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type T the scratchpad memory to be passed to *unmtr* function should be able to hold.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

LAPACK-like Extensions Routines

oneAPI Math Kernel Library DPC++ provides additional routines to extend the functionality of the LAPACK routines. These include routines to compute many independent factorizations, linear equation solutions, and similar. The following table lists the LAPACK-like Extensions routine groups.

Routines	Scratchpad Size Routines	Description
geqrf_ba	*geqrf_batch_scra*	Computes the QR factorizations of a batch of general matrices.
getrf_ba	*getrf_batch_scrat*	Computes the LU factorizations of a batch of general matrices.
getri_ba	*getri_batch_scrat*	Computes the inverses of a batch of LU-factored general matrices.
getrs_ba	*getrs_batch_scra*	Solves systems of linear equations with a batch of LU-factored square coefficient matrices, with multiple right-hand sides.
orgqr_b	*orgqr_batch_scra*	Generates the real orthogonal/complex unitary matrix Q_i of the QR factorization formed by geqrf_batch.
potrf_ba	*potrf_batch_scra*	Computes the Cholesky factorization of a batch of symmetric (Hermitian) positive-definite matrices.
potrs_ba	*potrs_batch_scra*	Solves systems of linear equations with a batch of Cholesky-factored symmetric (Hermitian) positive-definite coefficient matrices, with multiple right-hand sides.
ungqr_b	*ungqr_batch_scra*	Generates the complex unitary matrix Q_i with the QR factorization formed by geqrf_batch.

geqrf_batch

Computes the QR factorizations of a batch of general matrices.

Description

`geqrf_batch` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

geqrf_batch (Buffer Version)

Description

The buffer version of `geqrf_batch` supports only the strided API.

Strided API

Syntax

```
namespace oneapi::mkl::lapack {
  void geqrf_batch(cl::sycl::queue &queue, std::int64_t m, std::int64_t n,␣
↪cl::sycl::buffer<T> &a, std::int64_t lda, std::int64_t stride_a, cl::sycl::buffer<T> &
↪tau, std::int64_t stride_tau, std::int64_t batch_size, cl::sycl::buffer<T> &scratchpad,
↪ std::int64_t scratchpad_size)
}
```

Input Parameters

queue
: Device queue where calculations will be performed.

m
: Number of rows in matrices A_i ($0 \leq m$).

n
: Number of columns in matrices A_i ($0 \leq n$).

a
: Array holding input matrices A_i.

lda
: Leading dimension of matrices A_i.

stride_a
: Stride between the beginnings of matrices A_i inside the batch array `a`.

stride_tau
Stride between the beginnings of arrays τ_i inside the array `tau`.

batch_size
Number of problems in a batch.

scratchpad
Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size
Size of scratchpad memory as the number of floating point elements of type `T`. Size should not be less than the value returned by the Strided API of the *geqrf_batch_scratchpad_size* function.

Output Parameters

a
Factorization data as follows: The elements on and above the diagonal of A_i contain the $\min(m, n) \times n$ upper trapezoidal matrices R_i (R_i is upper triangular if $m \geq n$); the elements below the diagonal, with the array τ_i, contain the orthogonal matrix Q_i as a product of $\min(m, n)$ elementary reflectors.

tau
Array to store batch of τ_i, each of size $\min(m, n)$, containing scalars that define elementary reflectors for the matrices Q_i in its decomposition in a product of elementary reflectors.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

geqrf_batch (USM Version)

Description

The USM version of `geqrf_batch` supports the group API and strided API.

Group API

The routine forms the $Q_i R_i$ factorizations of a general $m \times n$ matrices A_i, $i \in \{1...batch\_size\}$, where `batch_size` is the sum of all parameter group sizes as provided with `group_sizes` array. No pivoting is performed during factorization. The routine does not form the matrices Q_i explicitly. Instead, Q_i is represented as a product of $\min(m, n)$ elementary reflectors. Routines are provided to work with Q_i in this representation. The total number of problems to solve, `batch_size`, is a sum of sizes of all of the groups of parameters as provided by `group_sizes` array.

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event geqrf_batch(cl::sycl::queue &queue, std::int64_t *m, std::int64_t *n,␣
↪T **a, std::int64_t *lda, T **tau, std::int64_t group_count, std::int64_t *group_sizes,
↪ T *scratchpad, std::int64_t scratchpad_size, const std::vector<cl::sycl::event> &
↪events = {})
}
```

Input Parameters

queue
: Device queue where calculations will be performed.

m
: Array of `group_count` m_g parameters. Each m_g specifies the number of rows in matrices A_i from array `a`, belonging to group g.

n
: Array of `group_count` n_g parameters. Each n_g specifies the number of columns in matrices A_i from array `a`, belonging to group g.

a
: Array of `batch_size` pointers to input matrices A_i, each of size $\text{lda}_g \cdot n_g$ (g is an index of group to which A_i belongs)

lda
: Array of `group_count` lda_g‘ parameters, each representing the leading dimensions of input matrices A_i from array `a`, belonging to group g.

group_count
: Specifies the number of groups of parameters. Must be at least 0.

group_sizes
: Array of `group_count` integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

scratchpad
: Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size
Size of scratchpad memory as the number of floating point elements of type `T`. Size should not be less than the value returned by the Group API of the *geqrf_batch_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a
Factorization data as follows: The elements on and above the diagonal of A_i contain the $\min(m_g, n_g) \times n_g$ upper trapezoidal matrices R_i (R_i is upper triangular if $m_g \geq n_g$); the elements below the diagonal, with the array τ_i, contain the orthogonal matrix Q_i as a product of $\min(m_g, n_g)$ elementary reflectors. Here g is the index of the parameters group corresponding to the i-th decomposition.

tau
Array of pointers to store arrays τ_i, each of size $\min(m_g, n_g)$, containing scalars that define elementary reflectors for the matrices Q_i in its decomposition in a product of elementary reflectors. Here g is the index of the parameters group corresponding to the i-th decomposition.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

Strided API

The routine forms the $Q_i R_i$ factorizations of general $m \times n$ matrices A_i. No pivoting is performed. The routine does not form the matrices Q_i explicitly. Instead, Q_i is represented as a product of $\min(m, n)$ elementary reflectors. Routines are provided to work with Q_i in this representation.

Syntax

```
namespace oneapi::mkl::lapack {
  sycl::event geqrf_batch(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, T *a,␣
↪std::int64_t lda, std::int64_t stride_a, T *tau, std::int64_t stride_tau, std::int64_t␣
↪batch_size, T *scratchpad, std::int64_t scratchpad_size, const std::vector
↪<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
Device queue where calculations will be performed.

m
Number of rows in matrices A_i ($0 \leq m$).

n
Number of columns in matrices A_i ($0 \leq n$).

a
Array holding input matrices A_i.

lda
Leading dimensions of A_i.

stride_a
Stride between the beginnings of matrices A_i inside the batch array `a`.

stride_tau
Stride between the beginnings of arrays τ_i inside the array `tau`.

batch_size
Number of problems in a batch.

scratchpad
Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size
Size of scratchpad memory as the number of floating point elements of type `T`. Size should not be less than the value returned by the Strided API of the *geqrf_batch_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a
Factorization data as follows: The elements on and above the diagonal of A_i contain the $\min(m, n) \times n$ upper trapezoidal matrices R_i (R_i is upper triangular if $m \geq n$); the elements below the diagonal, with the array τ_i, contain the orthogonal matrix Q_i as a product of $\min(m, n)$ elementary reflectors.

tau
Array to store batch of τ_i, each of size $\min(m, n)$, containing scalars that define elementary reflectors for the matrices Q_i in its decomposition in a product of elementary reflectors.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

Parent topic: *LAPACK-like Extensions Routines*

geqrf_batch_scratchpad_size

Computes size of scratchpad memory required for the *geqrf_batch* function.

Description

`geqrf_batch_scratchpad_size` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Group API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *geqrf_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t geqrf_batch_scratchpad_size(cl::sycl::queue &queue, std::int64_t *m,␣
↪std::int64_t *n, std::int64_t *lda, std::int64_t group_count, std::int64_t *group_
↪sizes)
}
```

Input Parameters

queue
Device queue where calculations will be performed.

m

Array of `group_count` m_g parameters.
Each of m_g specifies the number of rows in the matrices A_i belonging to group g.

n

Array of `group_count` n_g parameters.
Each of n_g specifies the number of columns in the matrices A_i belonging to group g.

lda
Array of `group_count` lda_g parameters, each representing the leading dimensions of input matrices belonging to group g.

group_count
Number of groups of parameters. Must be at least 0.

group_sizes
Array of `group_count` integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *geqrf_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Strided API

Computes the number of elements of type T the scratchpad memory should able to hold to be passed to the Strided API of the *geqrf_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t geqrf_batch_scratchpad_size(cl::sycl::queue &queue, std::int64_t m,␣
↪std::int64_t n, std::int64_t lda, std::int64_t stride_a, std::int64_t stride_tau,␣
↪std::int64_t batch_size)
};
```

Input Parameters

queue
Device queue where calculations will be performed.

m
Number of rows in the matrices A_i ($0 \leq m$).

n
Number of columns in A_i ($0 \leq n$).

lda
Leading dimension of A_i.

stride_a
Stride between the beginnings of matrices A_i inside the batch array `a`.

stride_tau
Stride between the beginnings of arrays τ_i inside the array `tau`.

batch_size
Number of problems in a batch.

Return Values

Number of elements of type T the scratchpad memory should able to hold to be passed to the Strided API of the *geqrf_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Parent topic: *LAPACK-like Extensions Routines*

getrf_batch

Computes the LU factorizations of a batch of general matrices.

Description

`getrf_batch` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

getrf_batch (Buffer Version)

Description

The buffer version of `getrf_batch` supports only the strided API.

Strided API

The routine computes the LU factorizations of general $m \times n$ matrices A_i as $A_i = P_i L_i U_i$, where P_i is a permutation matrix, L_i is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U_i is upper triangular (upper trapezoidal if $m < n$). The routine uses partial pivoting, with row interchanges.

Syntax

```
namespace oneapi::mkl::lapack {
  void getrf_batch(cl::sycl::queue &queue, std::int64_t m, std::int64_t n,␣
↪cl::sycl::buffer<T> &a, std::int64_t lda, std::int64_t stride_a, cl::sycl::buffer
↪<std::int64_t> &ipiv, std::int64_t stride_ipiv, std::int64_t batch_size,␣
↪cl::sycl::buffer<T> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue
: Device queue where calculations will be performed.

m
: Number of rows in matrices A_i ($0 \leq m$).

n
: Number of columns in matrices A_i ($0 \leq n$).

a
: Array holding input matrices A_i.

lda
: Leading dimension of matrices A_i.

stride_a
Stride between the beginnings of matrices A_i inside the batch array `a`.

stride_ipiv
Stride between the beginnings of arrays $ipiv_i$ inside the array `ipiv`.

batch_size
Number of problems in a batch.

scratchpad
Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by the Strided API of the *getrf_batch_scratchpad_size* function.

Output Parameters

a
L_i and U_i. The unit diagonal elements of L_i are not stored.

ipiv
Array containing batch of the pivot indices ipiv_i each of size at least $\max(1, \min(m, n))$; for $1 \leq k \leq \min(m, n)$, where row k of A_i was interchanged with row $\text{ipiv}_i(k)$.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

If `info` is positive, then the factorization has been completed, but some of U_i are exactly singular. Division by 0 will occur if you use the factor U_i for solving a system of linear equations.

The indices of such matrices in the batch can be obtained with *ids()* method of the exception object. The indices of first zero diagonal elements in these U_i matrices can be obtained by *exceptions()* method of exception object.

getrf_batch (USM Version)

Description

The USM version of `getrf_batch` supports the group API and strided API.

Group API

The routine computes the batch of LU factorizations of general $m \times n$ matrices A_i ($i \in \{1...batch\_size\}$) as $A_i = P_i L_i U_i$, where P_i is a permutation matrix, L_i is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U_i is upper triangular (upper trapezoidal if $m < n$). The routine uses partial pivoting, with row interchanges. Total number of problems to solve, `batch_size`, is a sum of sizes of all of the groups of parameters as provided by `group_sizes` array.

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event getrf_batch(cl::sycl::queue &queue, std::int64_t *m, std::int64_t *n,␣
↪T **a, std::int64_t *lda, std::int64_t **ipiv, std::int64_t group_count, std::int64_t␣
↪*group_sizes, T *scratchpad, std::int64_t scratchpad_size, const std::vector
↪<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
: Device queue where calculations will be performed.

m
: Array of `group_count` parameters m_g specifying the number of rows in matrices A_i ($0 \leq m_g$) belonging to group g.

n
: Array of `group_count` parameters n_g specifying the number of columns in matrices A_i ($0 \leq n_g$) belonging to group g.

a
: Array holding `batch_size` pointers to input matrices A_i.

lda
: Array of `group_count` parameters lda_g specifying the leading dimensions of A_i belonging to group g.

group_count
: Number of groups of parameters. Must be at least 0.

group_sizes
: Array of group_count integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

scratchpad
: Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size
: Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by the Group API of the *getrf_batch_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a
L_i and U_i. The unit diagonal elements of L_i are not stored.

ipiv
Arrays of batch_size pointers to arrays containing pivot indices ipiv_i each of size at least $\max(1, \min(m_g, n_g))$; for $1 \leq k \leq \min(m_g, n_g)$, where row k of A_i was interchanged with row $\text{ipiv}_i(k)$.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

If `info` is positive, then the factorization has been completed, but some of U_i are exactly singular. Division by 0 will occur if you use the factor U_i for solving a system of linear equations.

The indices of such matrices in the batch can be obtained with *ids()* method of the exception object. The indices of first zero diagonal elements in these U_i matrices can be obtained by *exceptions()* method of exception object.

Strided API

The routine computes the LU factorizations of general $m \times n$ matrices A_i as $A_i = P_i L_i U_i$, where P_i is a permutation matrix, L_i is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U_i is upper triangular (upper trapezoidal if $m < n$). The routine uses partial pivoting, with row interchanges.

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event getrf_batch(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, T␣
↪*a, std::int64_t lda, std::int64_t stride_a, std::int64_t *ipiv, std::int64_t stride_
↪ipiv, std::int64_t batch_size, T *scratchpad, std::int64_t scratchpad_size, const␣
↪std::vector<cl::sycl::event> &events = {})
};
```

Input Parameters

queue
Device queue where calculations will be performed.

m
Number of rows in matrices A_i ($0 \leq m$).

n
Number of columns in matrices A_i ($0 \leq n$).

a
Array holding input matrices A_i.

lda
Leading dimension of matrices A_i.

stride_a
Stride between the beginnings of matrices A_i inside the batch array `a`.

stride_ipiv
Stride between the beginnings of arrays ipiv_i inside the array `ipiv`.

batch_size
Number of problems in a batch.

scratchpad
Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by the Strided API of the *getrf_batch_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a
L_i and U_i. The unit diagonal elements of L_i are not stored.

ipiv
Array containing batch of the pivot indices ipiv_i each of size at least $\max(1, \min(m, n))$; for $1 \leq k \leq \min(m, n)$, where row k of A_i was interchanged with row $\text{ipiv}_i(k)$.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

If `info` is positive, then the factorization has been completed, but some of U_i are exactly singular. Division by 0 will occur if you use the factor U_i for solving a system of linear equations.

The indices of such matrices in the batch can be obtained with *ids()* method of the exception object. The indices of first zero diagonal elements in these U_i matrices can be obtained by *exceptions()* method of exception object.

Parent topic: *LAPACK-like Extensions Routines*

getrf_batch_scratchpad_size

Computes size of scratchpad memory required for the *getrf_batch* function.

Description

`getrf_batch_scratchpad_size` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Group API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *getrf_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t getrf_batch_scratchpad_size(cl::sycl::queue &queue, std::int64_t *m,␣
↪std::int64_t *n, std::int64_t *lda, std::int64_t group_count, std::int64_t *group_
↪sizes)
}
```

Input Parameters

queue
Device queue where calculations will be performed.

m
Array of `group_count` parameters m_g specifying the number of rows in the matrices belonging to group g.

n
Array of `group_count` parameters n_g specifying the number of columns in matrices belonging to group g.

lda
Array of `group_count` parameters lda_g specifying the leading dimensions of matrices belonging to group g.

group_count
Number of groups of parameters. Must be at least 0.

group_sizes
Array of `group_count` integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *getrf_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Strided API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Strided API of the *getrf_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t getrf_batch_scratchpad_size(cl::sycl::queue &queue, std::int64_t m,␣
↪std::int64_t n, std::int64_t lda, std::int64_t stride_a, std::int64_t stride_ipiv,␣
↪std::int64_t batch_size)
};
```

Input Parameters

queue
Device queue where calculations will be performed.

m
Number of rows in the matrices A_i ($0 \leq m$).

n
Number of columns in A_i ($0 \leq n$).

lda
Leading dimension of A_i.

stride_a
Stride between the beginnings of matrices A_i inside the batch array `a`.

stride_ipiv
Stride between the beginnings of arrays ipiv_i inside the array `ipiv`.

batch_size
Number of problems in a batch.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Strided API of the *getrf_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Parent topic: *LAPACK-like Extensions Routines*

getri_batch

Computes the inverses of a batch of LU-factored matrices determined by *getrf_batch*.

Description

`getri_batch` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

getri_batch (Buffer Version)

Description

The buffer version of `getri_batch` supports only the strided API.

Strided API

The routine computes the inverses A_i^{-1} of general matrices A_i. Before calling this routine, call the Strided API of the *getrf_batch (Buffer Version)* function to factorize A_i.

Syntax

```
namespace oneapi::mkl::lapack {
  void getri_batch(cl::sycl::queue &queue, std::int64_t n, cl::sycl::buffer<T> &a,␣
↪std::int64_t lda, std::int64_t stride_a, cl::sycl::buffer<std::int64_t> &ipiv,␣
↪std::int64_t stride_ipiv, std::int64_t batch_size, cl::sycl::buffer<T> &scratchpad,␣
↪std::int64_t scratchpad_size)
}
```

Input Parameters

queue
: Device queue where calculations will be performed.

n
: Order of the matrices A_i ($0 \leq n$).

a
: Result of the Strided API of the *getrf_batch (Buffer Version)* function.

lda
: Leading dimension of A_i ($n \leq$ lda).

stride_a
: Stride between the beginnings of matrices A_i inside the batch array `a`.

ipiv
Arrays returned by the Strided API of the *getrf_batch (Buffer Version)* function.

stride_ipiv
Stride between the beginnings of arrays ipiv$_i$ inside the array `ipiv`.

batch_size
Number of problems in a batch.

scratchpad
Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by the Strided API of the *getri_batch_scratchpad_size* function.

Output Parameters

a
Inverse $n \times n$ matrices A_i^{-1}.

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

getri_batch (USM Version)

Description

The USM version of `getri_batch` supports the group API and strided API.

Group API

The routine computes the inverses A_i^{-1} of general matrices A_i, $i \in \{1...batch\_size\}$. Before calling this routine, call the Group API of the *getrf_batch (USM Version)* function to factorize A_i. Total number of problems to solve, `batch_size`, is a sum of sizes of all of the groups of parameters as provided by `group_sizes` array.

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event getri_batch(cl::sycl::queue &queue, std::int64_t *n, T **a, std::int64_
↪t *lda, std::int64_t **ipiv, std::int64_t group_count, std::int64_t *group_sizes, T␣
↪*scratchpad, std::int64_t scratchpad_size, const std::vector<cl::sycl::event> &events␣
↪= {})
}
```

Input Parameters

queue
Device queue where calculations will be performed.

n
Array of `group_count` n_g parameters specifying the order of the matrices A_i ($0 \leq n_g$) belonging to group g.

a
Result of the Group API of the *getrf_batch (USM Version)* function.

lda
Array of `group_count` lda_g parameters specifying the leading dimensions of the matrices A_i ($n_g \leq \text{lda}_g$) belonging to group g.

ipiv
Arrays returned by the Group API of the *getrf_batch (USM Version)* function.

group_count
Number of groups of parameters. Must be at least 0.

group_sizes
Array of `group_count` integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

scratchpad
Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by the Group API of the *getri_batch_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a
Inverse $n_g \times n_g$ matrices A_i^{-1}.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

Strided API

The routine computes the inverses A_i^{-1} of general matrices A_i. Before calling this routine, call the Strided API of the *getrf_batch (USM Version)* function to factorize A_i.

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event getri_batch(cl::sycl::queue &queue, std::int64_t n, T *a, std::int64_t␣
↪lda, std::int64_t stride_a, std::int64_t *ipiv, std::int64_t stride_ipiv, std::int64_t␣
↪batch_size, T *scratchpad, std::int64_t scratchpad_size, const std::vector
↪<cl::sycl::event> &events = {})
};
```

Input Parameters

queue
Device queue where calculations will be performed.

n
Order of the matrices A_i ($0 \leq n$).

a
Result of the Strided API of the *getrf_batch (USM Version)* function.

lda
Leading dimension of A_i ($n \leq$ lda).

stride_a
Stride between the beginnings of matrices A_i inside the batch array `a`.

ipiv
Arrays returned by the Strided API of the *getrf_batch (USM Version)* function.

stride_ipiv
Stride between the beginnings of arrays ipiv_i inside the array `ipiv`.

batch_size
Number of problems in a batch.

scratchpad
Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by the Strided API of the *getri_batch_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a
Inverse $n \times n$ matrices A_i^{-1}.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

Parent topic: *LAPACK-like Extensions Routines*

getri_batch_scratchpad_size

Computed size of scratchpad memory required for the *getri_batch* function.

Description

`getri_batch_scratchpad_size` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Group API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *getri_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t getri_batch_scratchpad_size(cl::sycl::queue &queue, std::int64_t *n,␣
↪std::int64_t *lda, std::int64_t group_count, std::int64_t *group_sizes)
}
```

Input Parameters

queue
: Device queue where calculations will be performed.

n
: Array of `group_count` n_g parameters specifying the order of the matrices belonging to group g.

lda
: Array of `group_count` lda_g parameters specifying the leading dimensions of the matrices belonging to group g.

group_count
: Number of groups of parameters. Must be at least 0.

group_sizes
: Array of `group_count` integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *getri_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Strided API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Strided API of the *getri_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t getri_batch_scratchpad_size(cl::sycl::queue &queue, std::int64_t n,␣
↪std::int64_t lda, std::int64_t stride_a, std::int64_t stride_ipiv, std::int64_t batch_
↪size)
};
```

Input Parameters

queue
> Device queue where calculations will be performed.

n
> The order of the matrices A_i (0 le n).

lda
> Leading dimension of A_i ($n \leq$ lda).

stride_a
> Stride between the beginnings of matrices A_i inside the batch array `a`.

stride_ipiv
> Stride between the beginnings of arrays $ipiv_i$ inside the array ipiv.

batch_size
> Specifies the number of problems in a batch.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Strided API of the *getri_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Parent topic: *LAPACK-like Extensions Routines*

getrs_batch

Solves a system of linear equations with a batch of LU-factored square coefficient matrices, with multiple right-hand sides.

Description

`getrs_batch` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

getrs_batch (Buffer Version)

Description

The buffer version of `getrs_batch` supports only the strided API.

Strided API

> The routine solves for the following systems of linear equations X_i:
> $A_i X_i = B_i$, if `trans=mkl::transpose::nontrans`
> $A_i^T X_i = B_i$, if `trans=mkl::transpose::trans`
> $A_i^H X_i = B_i$, if `trans=mkl::transpose::conjtrans`
> Before calling this routine, the Strided API of the *getrf_batch (Buffer Version)* function should be called to compute the LU factorizations of A_i.

Syntax

```
namespace oneapi::mkl::lapack {
  void getrs_batch(cl::sycl::queue &queue, mkl::transpose trans, std::int64_t n,␣
↪std::int64_t nrhs, cl::sycl::buffer<T> &a, std::int64_t lda, std::int64_t stride_a,␣
↪cl::sycl::buffer<std::int64_t> &ipiv, std::int64_t stride_ipiv, cl::sycl::buffer<T> &b,
↪ std::int64_t ldb, std::int64_t stride_b, std::int64_t batch_size, cl::sycl::buffer<T>␣
↪&scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue
Device queue where calculations will be performed.

trans
Form of the equations:
If `trans = mkl::transpose::nontrans`, then $A_i X_i = B_i$ is solved for Xi.
If `trans = mkl::transpose::trans`, then $A_i^T X_i = B_i$ is solved for X_i.
If `trans = mkl::transpose::conjtrans`, then $A_i^H X_i = B_i$ is solved for X_i.

n
Order of the matrices A_i and the number of rows in matrices B_i ($0 \leq n$).

nrhs
Number of right-hand sides ($0 \leq$ nrhs).

a
Array containing the factorizations of the matrices A_i, as returned the Strided API of the *getrf_batch (Buffer Version)* function.

lda
Leading dimension of A_i.

stride_a
Stride between the beginnings of matrices A_i inside the batch array `a`.

ipiv
`ipiv` array, as returned by the Strided API of the *getrf_batch (Buffer Version)* function.

stride_ipiv
Stride between the beginnings of arrays ipiv$_i$ inside the array `ipiv`.

b
Array containing the matrices B_i whose columns are the right-hand sides for the systems of equations.

ldb
Leading dimension of B_i.

stride_b
Stride between the beginnings of matrices B_i inside the batch array `b`.

batch_size
Specifies the number of problems in a batch.

scratchpad
Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by the Strided API of the *getrs_batch_scratchpad_size* function.

Output Parameters

b
Solution matrices X_i.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

If `info` is zero, then diagonal element of some of U_i is zero, and the solve could not be completed. The indices of such matrices in the batch can be obtained with *ids()* method of the exception object. The indices of first zero diagonal elements in these U_i matrices can be obtained by *exceptions()* method of exception object.

getrs_batch (USM Version)

Description

The USM version of `getrs_batch` supports the group API and strided API.

Group API

The routine solves the following systems of linear equations for X_i ($i \in \{1...batch\_size\}$):
$A_i X_i = B_i$, if `trans=mkl::transpose::nontrans`
$A_i^T X_i = B_i$, if `trans=mkl::transpose::trans`
$A_i^H X_i = B_i$, if `trans=mkl::transpose::conjtrans`
Before calling this routine, call the Group API of the *getrf_batch (USM Version)* function to compute the LU factorizations of A_i.
Total number of problems to solve, `batch_size`, is a sum of sizes of all of the groups of parameters as provided by `group_sizes` array.

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event getrs_batch(cl::sycl::queue &queue, mkl::transpose *trans, std::int64_
↪t *n, std::int64_t *nrhs, T **a, std::int64_t *lda, std::int64_t **ipiv, T **b,␣
↪std::int64_t *ldb, std::int64_t group_count, std::int64_t *group_sizes, T *scratchpad,␣
↪std::int64_t scratchpad_size, const std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
Device queue where calculations will be performed.

trans
Array of `group_count` parameters $trans_g$ indicating the form of the equations for the group g:
If `trans = mkl::transpose::nontrans`, then $A_i X_i = B_i$ is solved for X_i.
If `trans = mkl::transpose::trans`, then $A_i^T X_i = B_i$ is solved for X_i.
If `trans = mkl::transpose::conjtrans`, then $A_i^H X_i = B_i$ is solved for X_i.

n
Array of `group_count` parameters n_g specifying the order of the matrices A_i and the number of rows in matrices B_i ($0 \leq n_g$) belonging to group g.

nrhs
Array of `group_count` parameters nrhs_g specifying the number of right-hand sides ($0 \leq \text{nrhs}_g$) for group g.

a
Array of `batch_size` pointers to factorizations of the matrices A_i, as returned by the Group API of the:ref:*onemkl_lapack_getrf_batch_usm* function.

lda
Array of `group_count` parameters lda_g specifying the leading dimensions of A_i from group g.

ipiv
`ipiv` array, as returned by the Group API of the *getrf_batch (USM Version)* function.

b
The array containing `batch_size` pointers to the matrices B_i whose columns are the right-hand sides for the systems of equations.

ldb
Array of `group_count` parameters ldb_g specifying the leading dimensions of B_i in the group g.

group_count
Specifies the number of groups of parameters. Must be at least 0.

group_sizes
Array of `group_count` integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

scratchpad
Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by the Group API of the *getrs_batch_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

b
Solution matrices X_i.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of problems during calculations. The info code of the problem can be obtained by info() method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

If `info` is zero, then diagonal element of some of U_i is zero, and the solve could not be completed. The indices of such matrices in the batch can be obtained with *ids()* method of the exception object. The indices of first zero diagonal elements in these U_i matrices can be obtained by *exceptions()* method of exception object.

Strided API

The routine solves the following systems of linear equations for X_i:
$A_i X_i = B_i$, if `trans=mkl::transpose::nontrans`
$A_i^T X_i = B_i$, if `trans=mkl::transpose::trans`
$A_i^H X_i = B_i$, if `trans=mkl::transpose::conjtrans`
Before calling this routine, the Strided API of the *getrf_batch* function should be called to compute the LU factorizations of A_i.

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event getrs_batch(cl::sycl::queue &queue, mkl::transpose trans, std::int64_t␣
↪n, std::int64_t nrhs, T *a, std::int64_t lda, std::int64_t stride_a, std::int64_t␣
↪*ipiv, std::int64_t stride_ipiv, T *b, std::int64_t ldb, std::int64_t stride_b,␣
↪std::int64_t batch_size, T *scratchpad, std::int64_t scratchpad_size, const std::vector
↪<cl::sycl::event> &events = {})
};
```

Input Parameters

queue
Device queue where calculations will be performed.

trans
Form of the equations:
If `trans = mkl::transpose::nontrans`, then $A_i X_i = B_i$ is solved for X_i.
If `trans = mkl::transpose::trans`, then $A_i^T X_i = B_i$ is solved for X_i.
If `trans = mkl::transpose::conjtrans`, then $A_i^H X_i = B_i$ is solved for X_i.

n
Order of the matrices A_i and the number of rows in matrices B_i ($0 \leq n$).

nrhs
Number of right-hand sides ($0 \leq$ nrhs).

a
Array containing the factorizations of the matrices A_i, as returned by the Strided API of the:ref:*onemkl_lapack_getrf_batch_usm* function.

lda
Leading dimension of A_i.

stride_a
Stride between the beginnings of matrices A_i inside the batch array `a`.

ipiv
`ipiv` array, as returned by getrf_batch (USM) function.

stride_ipiv
Stride between the beginnings of arrays ipiv_i inside the array `ipiv`.

b
Array containing the matrices B_i whose columns are the right-hand sides for the systems of equations.

ldb
Leading dimensions of B_i.

stride_b
Stride between the beginnings of matrices B_i inside the batch array `b`.

batch_size
Number of problems in a batch.

scratchpad
Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by the Strided API of the *getrs_batch_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

b
Solution matrices X_i.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

If `info` is zero, then diagonal element of some of U_i is zero, and the solve could not be completed. The indices of such matrices in the batch can be obtained with *ids()* method of the exception object. The indices of first zero diagonal elements in these U_i matrices can be obtained by *exceptions()* method of exception object.

Parent topic: *LAPACK-like Extensions Routines*

getrs_batch_scratchpad_size

Computes size of scratchpad memory required for the *getrs_batch* function.

Description

`getrs_batch_scratchpad_size` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Group API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *getrs_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t getrs_batch_scratchpad_size(cl::sycl::queue &queue, mkl::transpose *trans,
↪ std::int64_t *n, std::int64_t *nrhs, std::int64_t *lda, std::int64_t *ldb, std::int64_
↪t group_count, std::int64_t *group_sizes)
}
```

Input Parameters

queue
Device queue where calculations will be performed.

trans

Array of `group_count` parameters trans_g indicating the form of the equations for the group g:
If `trans = mkl::transpose::nontrans`, then $A_i X_i = B_i$ is solved for X_i.
If `trans = mkl::transpose::trans`, then $A_i^T X_i = B_i$ is solved for X_i.
If `trans = mkl::transpose::conjtrans`, then $A_i H X_i = B_i$ is solved for X_i.

n
Array of `group_count` parameters n_g specifying the order of the matrices A_i and the number of rows in matrices B_i ($0 \leq n_g$) belonging to group g.

nrhs
Array of `group_count` parameters nrhsg specifying the number of right-hand sides ($0 \leq \text{nrhs}_g$) for group g.

lda
Array of `group_count` parameters lda_g specifying the leading dimensions of A_i from group g.

ldb
Array of `group_count` parameters ldb_g specifying the leading dimensions of B_i in the group g.

group_count
Number of groups of parameters. Must be at least 0.

group_sizes
Array of `group_count` integers. Array element with index g specifies the number of problems to solve for each

of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *getrs_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Strided API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Strided API of the *getrs_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t getrs_batch_scratchpad_size(cl::sycl::queue &queue, mkl::transpose trans,␣
↪std::int64_t n, std::int64_t nrhs, std::int64_t lda, std::int64_t stride_a, std::int64_
↪t stride_ipiv, std::int64_t ldb, std::int64_t stride_b, std::int64_t batch_size)
};
```

Input Parameters

queue

Device queue where calculations will be performed.

trans

Indicates the form of the equations:
`If trans = mkl::transpose::nontrans`, then $A_i X_i = B_i$ is solved for X_i.
If `trans = mkl::transpose::trans`, then $A_i^T X_i = B_i$ is solved for X_i.
If `trans = mkl::transpose::conjtrans`, then $A_i^H X_i = B_i$ is solved for X_i.

n

Order of the matrices A_i and the number of rows in matrices B_i ($0 \leq n$).

nrhs

Number of right-hand sides ($0 \leq$ nrhs).

lda

Leading dimension of A_i.

stride_a
Stride between the beginnings of matrices A_i inside the batch array `a`.

stride_ipiv
Stride between the beginnings of arrays $ipiv_i$ inside the array `ipiv`.

ldb
Leading dimension of B_i.

stride_b
Stride between the beginnings of matrices B_i inside the batch array `b`.

batch_size
Number of problems in a batch.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Strided API of the *getrs_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Parent topic: *LAPACK-like Extensions Routines*

orgqr_batch

Generates the orthogonal/unitary matrix Q_i of the QR factorizations for a group of general matrices.

Description

`orgqr_batch` supports the following precisions.

T
float
double

orgqr_batch (Buffer Version)

Description

The buffer version of `orgqr_batch` supports only the strided API.

Strided API

The routine generates the wholes or parts of $m \times n$ orthogonal matrices Q_i of the batch of QR factorizations formed by the Strided API of the *geqrf_batch (Buffer Version)* function.

Usually Q_i is determined from the QR factorization of an $m \times p$ matrix A_i with $m \geq p$.

To compute the whole matrices Q_i, use:

```
orgqr_batch(queue, m, m, p, a, ...)
```

To compute the leading p columns of Q_i (which form an orthonormal basis in the space spanned by the columns of A_i):

```
orgqr_batch(queue, m, p, p, a, ...)
```

To compute the matrices Q_i^k of the QR factorizations of leading k columns of the matrices A_i:

```
orgqr_batch(queue, m, m, k, a, ...)
```

To compute the leading k columns of Q_i^k (which form an orthonormal basis in the space spanned by leading k columns of the matrices A_i):

```
orgqr_batch(queue, m, k, k, a, ...)
```

Syntax

```
namespace oneapi::mkl::lapack {
  void orgqr_batch(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, std::int64_t␣
↪k, cl::sycl::buffer<T> &a, std::int64_t lda, std::int64_t stride_a, cl::sycl::buffer<T>
↪ &tau, std::int64_t stride_tau, std::int64_t batch_size, cl::sycl::buffer<T> &
↪scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue
Device queue where calculations will be performed.

m
Number of rows in the matrices A_i ($0 \leq m$).

n
Number of columns in the matrices A_i ($0 \leq n \leq m$).

k
Number of elementary reflectors whose product defines the matrices Q_i ($0 \leq k \leq n$).

a
Array resulting after call to the Strided API of the *geqrf_batch (Buffer Version)* function.

lda
Leading dimension of A_i (lda $\leq m$).

stride_a
The stride between the beginnings of matrices A_i inside the batch array `a`.

tau
Array resulting from call to the Strided API of the *geqrf_batch (Buffer Version)* function.

stride_tau
Stride between the beginnings of arrays τ_i inside the array `tau`.

batch_size
Specifies the number of problems in a batch.

scratchpad
Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by the Strided API of the *orgqr_batch_scratchpad_size* function.

Output Parameters

a
Batch of n leading columns of the $m \times m$ orthogonal matrices Q_i.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

orgqr_batch (USM Version)

Description

The USM version of `orgqr_batch` supports the group API and strided API.

Group API

The routine generates the wholes or parts of $m \times n$ orthogonal matrices Q_i of the batch of QR factorizations formed by the Group API of the *geqrf_batch (USM Version)* function.

Usually Q_i is determined from the QR factorization of an $m \times p$ matrix A_i with $m \geq p$.

To compute the whole matrices Q_i, use:

```
orgqr_batch(queue, m, m, p, a, ...)
```

To compute the leading p columns of Q_i (which form an orthonormal basis in the space spanned by the columns of A_i):

`orgqr_batch(queue, m, p, p, a, ...)`

To compute the matrices Q_i^k of the QR factorizations of leading k columns of the matrices A_i:

`orgqr_batch(queue, m, m, k, a, ...)`

To compute the leading k columns of Q_i^k (which form an orthonormal basis in the space spanned by leading k columns of the matrices A_i):

`orgqr_batch(queue, m, k, k, a, ...)`

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event orgqr_batch(cl::sycl::queue &queue, std::int64_t *m, std::int64_t *n,␣
↪std::int64_t *k, T **a, std::int64_t *lda, T **tau, std::int64_t group_count,␣
↪std::int64_t *group_sizes, T *scratchpad, std::int64_t scratchpad_size, const␣
↪std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
Device queue where calculations will be performed.

m
Array of `group_count` m_g parameters as previously supplied to group version of geqrf_batch function.

n
Array of `group_count` n_g parameters as previously supplied to group version of geqrf_batch function.

k
Array of `group_count` k_g parameters as previously supplied to the Group API of the *geqrf_batch (USM Version)* function. The number of elementary reflectors whose product defines the matrices Q_i ($0 \leq k_g \leq n_g$).

a
Array resulting after call to the Group API of the *geqrf_batch (USM Version)* function.

lda
Array of leading dimensions of A_i as previously supplied to the Group API of the *geqrf_batch (USM Version)* function.

tau
Array resulting after call to the Group API of the *geqrf_batch (USM Version)* function.

group_count
Number of groups of parameters. Must be at least 0.

group_sizes
Array of `group_count` integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

scratchpad
Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by Group API of the *orgqr_batch_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a

n_g leading columns of the $m_g \times m_g$ orthogonal matrices Q_i, where g is an index of group of parameters corresponding to Q_i.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

Strided API

The routine generates the wholes or parts of $m \times n$ orthogonal matrices Q_i of the batch of QR factorizations formed by the Strided API of the *geqrf_batch (USM Version)* function.

Usually Q_i is determined from the QR factorization of an $m \times p$ matrix A_i with $m \geq p$.

To compute the whole matrices Q_i, use:

```
orgqr_batch(queue, m, m, p, a, ...)
```

To compute the leading p columns of Q_i (which form an orthonormal basis in the space spanned by the columns of A_i):

```
orgqr_batch(queue, m, p, p, a, ...)
```

To compute the matrices Q_i^k of the QR factorizations of leading k columns of the matrices A_i:

```
orgqr_batch(queue, m, m, k, a, ...)
```

To compute the leading k columns of Q_i^k (which form an orthonormal basis in the space spanned by leading k columns of the matrices A_i):

```
orgqr_batch(queue, m, k, k, a, ...)
```

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event orgqr_batch(cl::sycl::queue &queue, std::int64_t m, std::int64_t n,␣
↪std::int64_t k, T *a, std::int64_t lda, std::int64_t stride_a, T *tau, std::int64_t␣
↪stride_tau, std::int64_t batch_size, T *scratchpad, std::int64_t scratchpad_size,␣
↪const std::vector<cl::sycl::event> &events = {})
};
```

Input Parameters

queue
Device queue where calculations will be performed.

m
Number of rows in the matrices A_i ($0 \leq m$).

n
Number of columns in the matrices A_i ($0 \leq n \leq m$).

k
Number of elementary reflectors whose product defines the matrices Q_i ($0 \leq k \leq n$).

a
Array resulting after call to the Strided API of the *geqrf_batch (USM Version)* function.

lda
Leading dimension of A_i (lda $\leq m$).

stride_a
The stride between the beginnings of matrices A_i inside the batch array `a`.

tau
Array resulting from call to the Strided API of the *geqrf_batch (USM Version)* function.

stride_tau
Stride between the beginnings of arrays τ_i inside the array `tau`.

batch_size
Specifies the number of problems in a batch.

scratchpad
Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by the Strided API of the *orgqr_batch_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a

Batch of n leading columns of the $m \times m$ orthogonal matrices Q_i.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

Parent topic: *LAPACK-like Extensions Routines*

orgqr_batch_scratchpad_size

Computes size of scratchpad memory required for the *orgqr_batch* function.

Description

`orgqr_batch_scratchpad_size` supports the following precisions.

T
float
double

Group API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *orgqr_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t orgqr_batch_scratchpad_size(cl::sycl::queue &queue, std::int64_t *m,␣
↪std::int64_t *n, std::int64_t *k, std::int64_t *lda, std::int64_t group_count,␣
↪std::int64_t *group_sizes)
}
```

Input Parameters

queue
Device queue where calculations will be performed.

m
Array of `group_count` m_g parameters.

n
Array of `group_count` n_g parameters.

k
Array of `group_count` kg parameters. The number of elementary reflectors whose product defines the matrices Q_i ($0 \leq k_g \leq n_g$).

lda
Array of leading dimensions of A_i.

group_count
Number of groups of parameters. Must be at least 0.

group_sizes
Array of `group_count` integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *orgqr_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Strided API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Strided API of the *orgqr_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t orgqr_batch_scratchpad_size(cl::sycl::queue &queue, std::int64_t m,␣
↪std::int64_t n, std::int64_t k, std::int64_t lda, std::int64_t stride_a, std::int64_t␣
↪stride_tau, std::int64_t batch_size)
};
```

Input Parameters

queue
Device queue where calculations will be performed.

m
Number of rows in the matrices A_i ($0 \leq m$).

n
Number of columns in the matrices Ai ($0 \leq n$).

k
Number of elementary reflectors whose product defines the matrices Q_i ($0 \leq k \leq n$).

lda
Leading dimension of A_i (lda $\leq m$).

stride_a
Stride between the beginnings of matrices A_i inside the batch array `a`.

stride_tau
Stride between the beginnings of arrays tau_i inside the array `tau`.

batch_size
Number of problems in a batch.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Strided API of the *orgqr_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Parent topic: *LAPACK-like Extensions Routines*

potrf_batch

Computes the LU factorizations of a batch of general matrices.

Description

`potrf_batch` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

potrf_batch (Buffer Version)

Description

The buffer version of `potrf_batch` supports only the strided API.

Strided API

The routine forms the Cholesky factorizations of a symmetric positive-definite or, for complex data, Hermitian positive-definite matrices A_i, $i \in \{1...batch\_size\}$:
$A_i = U_i^T U_i$ for real data, $A_i = U_i^H U_i$ for complex data if `uplo = mkl::uplo::upper`,
$A_i = L_i L_i^T$ for real data, $A_i = L_i L_i^H$ for complex data if `uplo = mkl::uplo::lower`,
where L_i is a lower triangular matrix and U_i is upper triangular.

Syntax

```
namespace oneapi::mkl::lapack {
  void potrf_batch(cl::sycl::queue &queue, mkl::uplo uplo, std::int64_t n,␣
↪cl::sycl::buffer<T> &a, std::int64_t lda, std::int64_t stride_a, std::int64_t batch_
↪size, cl::sycl::buffer<T> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue
Device queue where calculations will be performed.

uplo

Indicates whether the upper or lower triangular part of A_i is stored and how A_i is factored:
If `uplo = mkl::uplo::upper`, the array `a` stores the upper triangular parts of the matrices A_i,
If `uplo = mkl::uplo::lower`, the array `a` stores the lower triangular parts of the matrices A_i.

n
Order of the matrices A_i, $(0 \leq n)$.

a
Array containing batch of input matrices A_i, each of A_i being of size lda $\cdot n$ and holding either upper or lower triangular parts of the matrices A_i (see `uplo`).

lda
Leading dimension of A_i.

stride_a
Stride between the beginnings of matrices A_i inside the batch.

batch_size
Number of problems in a batch.

scratchpad
Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by the Strided API of the *potrf_batch_scratchpad_size* function.

Output Parameters

a
Cholesky factors U_i or L_i, as specified by `uplo`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

If `info` is zero, then the leading minors of some of matrices (and therefore some matrices A_i themselves) are not positive-definite, and the factorizations could not be completed for these matrices from the batch. The indices of such matrices in the batch can be obtained with *ids()* method of the exception object. The orders of corresponding not positive-definite leading minors of these matrices can be obtained by *exceptions()* method of exception object.

potrf_batch (USM Version)

Description

The USM version of `potrf_batch` supports the group API and strided API.

Group API

The routine forms the Cholesky factorizations of symmetric positive-definite or, for complex data, Hermitian positive-definite matrices A_i, $i \in \{1...batch\_size\}$:
$A_i = U_i^T U_i$ for real data ($A_i = U_i^H U_i$ for complex), if uplo_g is `mkl::uplo::upper`,
$A_i = L_i L_i^T$ for real data ($A_i = L_i L_i^H$ for complex), if uplo_g is `mkl::uplo::lower`,
where L_i is a lower triangular matrix and U_i is upper triangular, g is an index of group of parameters corresponding to A_i, and total number of problems to solve, `batch_size`, is a sum of sizes of all of the groups of parameters as provided by `group_sizes` array

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event potrf_batch(cl::sycl::queue &queue, mkl::uplo *uplo, std::int64_t *n,␣
↪T **a, std::int64_t *lda, std::int64_t group_count, std::int64_t *group_sizes, T␣
↪*scratchpad, std::int64_t scratchpad_size, const std::vector<cl::sycl::event> &events␣
↪= {})
}
```

Input Parameters

queue
Device queue where calculations will be performed.

uplo

Array of `group_count` uplo_g parameters. Each uplo_g indicates whether the upper or lower triangular parts of the input matrices are provided:
If uplo_g is `mkl::uplo::upper`, input matrices from array `a` belonging to group g store the upper triangular parts,
If uplo_g is `mkl::uplo::lower`, input matrices from array `a` belonging to group g store the lower triangular parts.

n

Array of `group_count` n_g parameters. Each n_g specifies the order of the input matrices from array a belonging to group g.

a

Array of `batch_size` pointers to input matrices A_i, each being of size $\text{lda}_g \cdot n_g$ (g is an index of group to which A_i belongs to) and holding either upper or lower triangular part as specified by uplo_g.

lda

Array of `group_count` lda_g parameters. Each lda_g specifies the leading dimensions of the matrices from a belonging to group g.

group_count
Number of groups of parameters. Must be at least 0.

group_sizes
Array of group_count integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

scratchpad
Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by the Group API of the *potrf_batch_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a
Cholesky factors U_i or L_i, as specified by uplo_g from corresponding group of parameters.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

If `info` is zero, then the leading minors of some of the input matrices (and therefore some matrices themselves) are not positive-definite, and the factorizations could not be completed for these matrices from the batch. The indices of such matrices in the batch can be obtained with *ids()* method of the exception object. The orders of corresponding not positive-definite leading minors of these matrices can be obtained by *exceptions()* method of the exception object.

Strided API

The routine forms the Cholesky factorizations of a symmetric positive-definite or, for complex data, Hermitian positive-definite matrices A_i, $i \in \{1...batch\_size\}$:
$A_i = U_i^T U_i$ for real data, $A_i = U_i^H U_i$ for complex data if `uplo = mkl::uplo::upper`,
$A_i = L_i L_i^T$ for real data, $A_i = L_i L_i^H$ for complex data if `uplo = mkl::uplo::lower`,

where L_i is a lower triangular matrix and U_i is upper triangular.

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event potrf_batch(cl::sycl::queue &queue, mkl::uplo uplo, std::int64_t n, T␣
↪*a, std::int64_t lda, std::int64_t stride_a, std::int64_t batch_size, T *scratchpad,␣
↪std::int64_t scratchpad_size, const std::vector<cl::sycl::event> &events = {})
};
```

Input Parameters

queue
Device queue where calculations will be performed.

uplo

Indicates whether the upper or lower triangular part of A_i is stored and how A_i is factored:
If `uplo = mkl::uplo::upper`, the array `a` stores the upper triangular parts of the matrices A_i,
If `uplo = mkl::uplo::lower`, the array `a` stores the lower triangular parts of the matrices A_i.

n
Order of the matrices A_i, ($0 \leq n$).

a
Array containing batch of input matrices A_i, each of A_i being of size lda $\cdot$ n and holding either upper or lower triangular parts of the matrices A_i (see `uplo`).

lda
Leading dimension of A_i.

stride_a
Stride between the beginnings of matrices A_i inside the batch.

batch_size
Number of problems in a batch.

scratchpad
Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by the Strided API of the *potrf_batch_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a

Cholesky factors U_i or L_i, as specified by `uplo`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

If `info` is zero, then the leading minors of some of matrices (and therefore some matrices A_i themselves) are not positive-definite, and the factorizations could not be completed for these matrices from the batch. The indices of such matrices in the batch can be obtained with *ids()* method of the exception object. The orders of corresponding not positive-definite leading minors of these matrices can be obtained by *exceptions()* method of exception object.

Parent topic: *LAPACK-like Extensions Routines*

potrf_batch_scratchpad_size

Computes size of scratchpad memory required for the *potrf_batch* function.

Description

`potrf_batch_scratchpad_size` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Group API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *potrf_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t potrf_batch_scratchpad_size(cl::sycl::queue &queue, mkl::uplo *uplo,␣
↪std::int64_t *n, std::int64_t *lda, std::int64_t group_count, std::int64_t *group_
↪sizes)
}
```

Input Parameters

queue
: Device queue where calculations will be performed.

uplo

: Array of `group_count` uplo_g parameters.

Each of uplo_g indicates whether the upper or lower triangular parts of the input matrices are provided:

If uplo_g is `mkl::uplo::upper`, input matrices from array `a` belonging to group g store the upper triangular parts,

If uplo_g is `mkl::uplo::lower`, input matrices from array `a` belonging to group g store the lower triangular parts.

n

: Array of `group_count` n_g parameters.

Each ng specifies the order of the input matrices belonging to group g.

lda

: Array of `group_count` lda_g parameters.

Each ldag specifies the leading dimensions of the matrices belonging to group g.

group_count
: Number of groups of parameters. Must be at least 0.

group_sizes
: Array of `group_count` integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *potrf_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Strided API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Strided API of the *potrf_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t potrf_batch_scratchpad_size(cl::sycl::queue &queue, mkl::uplo uplo,␣
↪std::int64_t n, std::int64_t lda, std::int64_t stride_a, std::int64_t batch_size)
};
```

Input Parameters

queue
Device queue where calculations will be performed.

uplo
Indicates whether the upper or lower triangular part of A_i is stored and how A_i is factored:
If `uplo = mkl::uplo::upper`, the array `a` stores the upper triangular parts of the matrices A_i,
If `uplo = mkl::uplo::lower`, the array `a` stores the lower triangular parts of the matrices A_i.

n
Order of the matrices A_i, ($0 \leq n$).

lda
Leading dimension of A_i.

stride_a
Stride between the beginnings of matrices A_i inside the batch.

batch_size
Number of problems in a batch.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Strided API of the *potrf_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Parent topic: *LAPACK-like Extensions Routines*

potrs_batch

Computes the LU factorizations of a batch of general matrices.

Description

`potrs_batch` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

potrs_batch (Buffer Version)

Description

The buffer version of `potrs_batch` supports only the strided API.

Strided API

> The routine solves for X_i the systems of linear equations $A_i X_i = B_i$ with a symmetric positive-definite or, for complex data, Hermitian positive-definite matrices A_i, given the Cholesky factorization of A_i, $i \in \{1...batch\_size\}$:
> $A_i = U_i^T U_i$ for real data, $A_i = U_i^H U_i$ for complex data if `uplo = mkl::uplo::upper`,
> $A_i = L_i L_i^T$ for real data, $A_i = L_i L_i^H$ for complex data if `uplo = mkl::uplo::lower`,
> where L_i is a lower triangular matrix and U_i is upper triangular.
> The systems are solved with multiple right-hand sides stored in the columns of the matrices B_i.
> Before calling this routine, matrices A_i should be factorized by call to the Strided API of the *potrf_batch (Buffer Version)* function.

Syntax

```
namespace oneapi::mkl::lapack {
  void potrs_batch(cl::sycl::queue &queue, mkl::uplo uplo, std::int64_t n, std::int64_t␣
↪nrhs, cl::sycl::buffer<T> &a, std::int64_t lda, std::int64_t stride_a, cl::sycl::buffer
↪<T> &b, std::int64_t ldb, std::int64_t stride_b, std::int64_t batch_size,␣
↪cl::sycl::buffer<T> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue
Device queue where calculations will be performed.

uplo

Indicates how the input matrices have been factored:

If `uplo = mkl::uplo::upper`, the upper triangle U_i of A_i is stored, where $A_i = U_i^T U_i$ for real data, $A_i = U_i^H U_i$ for complex data.

If `uplo = mkl::uplo::lower`, the upper triangle L_i of A_i is stored, where $A_i = L_i L_i^T$ for real data, $A_i = L_i L_i^H$ for complex data.

n
The order of matrices A_i ($0 \leq n$).

nrhs
The number of right-hand sides ($0 \leq$ nrhs).

a
Array containing batch of factorizations of the matrices A_i, as returned by the Strided API of the *potrf_batch (Buffer Version)* function.

lda
Leading dimension of A_i.

stride_a
Stride between the beginnings of matrices inside the batch array `a`.

b
Array containing batch of matrices B_i whose columns are the right-hand sides for the systems of equations.

ldb
Leading dimension of B_i.

stride_b
Stride between the beginnings of matrices B_i inside the batch array `b`.

batch_size
Number of problems in a batch.

scratchpad
Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by the Strided API of the *potrs_batch_scratchpad_size* function.

Output Parameters

b
Solution matrices X_i.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

If `info` is zero, then for some of the matrices diagonal element of the Cholesky factor is zero, and the solve could not be completed. The indices of such matrices in the batch can be obtained with *ids()* method of the exception object. The indices of first zero diagonal elements in these matrices can be obtained by *exceptions()* method of exception object.

potrs_batch (USM Version)

Description

The USM version of `potrs_batch` supports the group API and strided API.

Group API

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event potrs_batch(cl::sycl::queue &queue, mkl::uplo *uplo, std::int64_t *n,␣
↪std::int64_t *nrhs, T **a, std::int64_t *lda, T **b, std::int64_t *ldb, std::int64_t␣
↪group_count, std::int64_t *group_sizes, T *scratchpad, std::int64_t scratchpad_size,␣
↪const std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
Device queue where calculations will be performed.

uplo

Array of `group_count` uplo_g parameters.
Each of uplo_g indicates whether the upper or lower triangular parts of the input matrices are provided:
If uplo_g is `mkl::uplo::upper`, input matrices from array `a` belonging to group g store the upper triangular parts,
If uplo_g is `mkl::uplo::lower`, input matrices from array `a` belonging to group g store the lower triangular parts.

n

Array of `group_count` n_g parameters.
Each n_g specifies the order of the input matrices from array `a` belonging to group g.

nrhs

Array of `group_count` nrhs_g parameters.
Each nrhs_g specifies the number of right-hand sides supplied for group g in corresponding part of array `b`.

a
Array of `batch_size` pointers to Cholesky factored matrices A_i as returned by the Group API of the *potrf_batch (USM Version)* function.

lda

Array of `group_count` lda_g parameters.
Each lda_g specifies the leading dimensions of the matrices from `a` belonging to group g.

b
Array of `batch_size` pointers to right-hand side matrices B_i, each of size $\text{ldb}_g \cdot \text{nrhs}_g$, where g is an index of group corresponding to B_i.

ldb

Array of `group_count` ldb_g parameters.
Each ldb_g specifies the leading dimensions of the matrices from `b` belonging to group g.

group_count
Number of groups of parameters. Must be at least 0.

group_sizes
Array of `group_count` integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

scratchpad
Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by the Group API of the *potrs_batch_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

b

Solution matrices X_i.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

If `info` is zero, then for some of the matrices diagonal element of the Cholesky factor is zero, and the solve could not be completed. The indices of such matrices in the batch can be obtained with *ids()* method of the exception object. The indices of first zero diagonal elements in these matrices can be obtained by *exceptions()* method of exception object.

Strided API

The routine solves for X_i the systems of linear equations $A_i X_i = B_i$ with a symmetric positive-definite or, for complex data, Hermitian positive-definite matrices A_i, given the Cholesky factorization of A_i, $i \in \{1...batch\_size\}$:
$A_i = U_i^T U_i$ for real data, $A_i = U_i^H U_i$ for complex data if `uplo = mkl::uplo::upper`,
$A_i = L_i L_i^T$ for real data, $A_i = L_i L_i^H$ for complex data if `uplo = mkl::uplo::lower`,
where L_i is a lower triangular matrix and U_i is upper triangular.
The systems are solved with multiple right-hand sides stored in the columns of the matrices B_i.
Before calling this routine, matrices A_i should be factorized by call to the Strided API of the *potrf_batch (USM Version)* function.

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event potrs_batch(cl::sycl::queue &queue, mkl::uplo uplo, std::int64_t n,␣
↪std::int64_t nrhs, T *a, std::int64_t lda, std::int64_t stride_a, T *b, std::int64_t␣
↪ldb, std::int64_t stride_b, std::int64_t batch_size, T *scratchpad, std::int64_t␣
↪scratchpad_size, const std::vector<cl::sycl::event> &events = {})
};
```

Input Parameters

queue
Device queue where calculations will be performed.

uplo

Indicates how the input matrices have been factored:

If `uplo = mkl::uplo::upper`, the upper triangle U_i of A_i is stored, where $A_i = U_i^T U_i$ for real data, $A_i = U_i^H U_i$ for complex data.

If `uplo = mkl::uplo::lower`, the upper triangle L_i of A_i is stored, where $A_i = L_i L_i^T$ for real data, $A_i = L_i L_i^H$ for complex data.

n
The order of matrices A_i ($0 \leq n$).

nrhs
The number of right-hand sides ($0 \leq nrhs$).

a
Array containing batch of factorizations of the matrices A_i, as returned by the Strided API of the *potrf_batch (USM Version)* function.

lda
Leading dimension of A_i.

stride_a
Stride between the beginnings of matrices inside the batch array `a`.

b
Array containing batch of matrices B_i whose columns are the right-hand sides for the systems of equations.

ldb
Leading dimension of B_i.

stride_b
Stride between the beginnings of matrices B_i inside the batch array `b`.

batch_size
Number of problems in a batch.

scratchpad
Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by the Strided API of the *potrs_batch_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

b
Solution matrices X_i.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of problems during calculations. The info code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

If `info` is zero, then for some of the matrices diagonal element of the Cholesky factor is zero, and the solve could not be completed. The indices of such matrices in the batch can be obtained with *ids()* method of the exception object. The indices of first zero diagonal elements in these matrices can be obtained by *exceptions()* method of exception object.

Parent topic: *LAPACK-like Extensions Routines*

potrs_batch_scratchpad_size

Computes size of scratchpad memory required for the *potrs_batch* function.

Description

`potrs_batch_scratchpad_size` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Group API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *potrs_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t potrs_batch_scratchpad_size(cl::sycl::queue &queue, mkl::uplo *uplo,␣
↪std::int64_t *n, std::int64_t *nrhs, std::int64_t *lda, std::int64_t *ldb, std::int64_
↪t group_count, std::int64_t *group_sizes)
}
```

Input Parameters

queue
: Device queue where calculations will be performed.

uplo

: Array of `group_count` uplo_g parameters.
Each of uplo_g indicates whether the upper or lower triangular parts of the input matrices are provided:
If uplo_g is `mkl::uplo::upper`, input matrices from array `a` belonging to group g store the upper triangular parts,
If uplo_g is `mkl::uplo::lower`, input matrices from array `a` belonging to group g store the lower triangular parts.

n

: Array of `group_count` n_g parameters.
Each n_g specifies the order of the input matrices belonging to group g.

nrhs

: Array of `group_count` nrhs_g parameters.
Each rhs_g specifies the number of right-hand sides supplied for group g.

lda

: Array of `group_count` lda_g parameters.
Each lda_g specifies the leading dimensions of the matrices belonging to group g.

ldb

: Array of `group_count` ldb_g parameters.
Each ldb_g specifies the leading dimensions of the matrices belonging to group g.

group_count
: Number of groups of parameters. Must be at least 0.

group_sizes Array of group_count integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *potrs_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Strided API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Strided API of the *potrs_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t potrs_batch_scratchpad_size(cl::sycl::queue &queue, mkl::uplo uplo,␣
↪std::int64_t n, std::int64_t nrhs, std::int64_t lda, std::int64_t stride_a, std::int64_
↪t ldb, std::int64_t stride_b, std::int64_t batch_size)
};
```

Input Parameters

queue
Device queue where calculations will be performed.

uplo

Indicates how the input matrices have been factored:
If `uplo = mkl::uplo::upper`, the upper triangle U_i of A_i is stored, where $A_i = U_i^T U_i$ for real data, $A_i = U_i^H U_i$ for complex data.
If `uplo = mkl::uplo::lower`, the upper triangle L_i of A_i is stored, where $A_i = L_i L_i^T$ for real data, $A_i = L_i L_i^H$ for complex data.

n
Order of matrices A_i ($0 \leq n$).

nrhs
Number of right-hand sides ($0 \leq$ nrhs).

lda
Leading dimension of A_i.

stride_a
Stride between the beginnings of matrices inside the batch array `a`.

ldb
Leading dimensions of B_i.

stride_b
Stride between the beginnings of matrices B_i inside the batch array `b`.

batch_size
Number of problems in a batch.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Strided API of the *potrs_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Parent topic: *LAPACK-like Extensions Routines*

ungqr_batch

Generates the complex unitary matrices Q_i of the batch of QR factorizations formed by the *geqrf_batch* function.

Description

`ungqr_batch` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

ungqr_batch (Buffer Version)

Description

The buffer version of `ungqr_batch` supports only the strided API.

Strided API

The routine generates the wholes or parts of :math`m times m` unitary matrices Q_i of the batch of QR factorization formed by the Strided API of the *geqrf_batch (Buffer Version)*.

Usually Q_i is determined from the QR factorization of an $m \times p$ matrix A_i with :math`m ge p`.

To compute the whole matrices Q_i, use:

`ungqr_batch(queue, m, m, p, a, ...)`
To compute the leading p columns of Q_i (which form an orthonormal basis in the space spanned by the columns of A_i):
`ungqr_batch(queue, m, p, p, a, ...)`
To compute the matrices ${Q_i}^{'k}$ of the QR factorizations of leading k columns of the matrices A_i:
`ungqr_batch(queue, m, m, k, a, ...)`
To compute the leading k columns of Q_i^k (which form an orthonormal basis in the space spanned by leading k columns of the matrices A_i):
`ungqr_batch(queue, m, k, k, a, ...)`

Syntax

```
namespace oneapi::mkl::lapack {
  void ungqr_batch(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, std::int64_t␣
↪k, cl::sycl::buffer<T> &a, std::int64_t lda, std::int64_t stride_a, cl::sycl::buffer<T>
↪ &tau, std::int64_t stride_tau, std::int64_t batch_size, cl::sycl::buffer<T> &
↪scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue
Device queue where calculations will be performed.

m
Number of rows in the matrices A_i ($0 \leq m$).

n
Number of columns in the matrices A_i ($0 \leq n$).

k
Number of elementary reflectors whose product defines the matrices Q_i ($0 \leq k \leq n$).

a
Array resulting after call to the Strided API of the *geqrf_batch (USM Version)* function.

lda
Leading dimension of A_i (lda $\leq m$).

stride_a
Stride between the beginnings of matrices A_i inside the batch array `a`.

tau
Array resulting after call to the Strided API of the *geqrf_batch (USM Version)* function.

stride_tau
Stride between the beginnings of arrays tau_i inside the array `tau`.

batch_size
Number of problems in a batch.

scratchpad
Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by strided version of the Strided API of the *ungqr_batch_scratchpad_size* function.

Output Parameters

a
Array data is overwritten by `a` batch of n leading columns of the $m \times m$ unitary matrices Q_i.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of problems during calculations. The info code of the problem can be obtained by info() method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

ungqr_batch (USM Version)

Description

The USM version of `ungqr_batch` supports the group API and strided API.

Group API

The routine generates the wholes or parts of :math`m times m` unitary matrices Q_i of the batch of QR factorization formed by the Group API of the *geqrf_batch (Buffer Version)*.

Usually Q_i is determined from the QR factorization of an $m \times p$ matrix A_i with :math`m ge p`.

To compute the whole matrices Q_i, use:

```
ungqr_batch(queue, m, m, p, a, ...)
```

To compute the leading p columns of Q_i (which form an orthonormal basis in the space spanned by the columns of A_i):

```
ungqr_batch(queue, m, p, p, a, ...)
```

To compute the matrices ${Q_i}^k$ of the QR factorizations of leading k columns of the matrices A_i:

```
ungqr_batch(queue, m, m, k, a, ...)
```

To compute the leading k columns of Q_i^k (which form an orthonormal basis in the space spanned by leading k columns of the matrices A_i):

```
ungqr_batch(queue, m, k, k, a, ...)
```

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event ungqr_batch(cl::sycl::queue &queue, std::int64_t *m, std::int64_t *n,␣
↪std::int64_t *k, T **a, std::int64_t *lda, T **tau, std::int64_t group_count,␣
↪std::int64_t *group_sizes, T *scratchpad, std::int64_t scratchpad_size, const␣
↪std::vector<cl::sycl::event> &events = {})
}
```

Input Parameters

queue
Device queue where calculations will be performed.

m
Array of `group_count` m_g parameters as previously supplied to the Group API of the *geqrf_batch (USM Version)* function.

n
Array of `group_count` n_g parameters as previously supplied to the Group API of the *geqrf_batch (USM Version)* function.

k
Array of `group_count` k_g parameters as previously supplied to the Group API of the *geqrf_batch (USM Version)* function.
The number of elementary reflectors whose product defines the matrices Q_i ($0 \leq k_g \leq n_g$).

a
Array resulting after call to the Group API of the *geqrf_batch (USM Version)* function.

lda
Array of leading dimensions of A_i as previously supplied to the Group API of the *geqrf_batch (USM Version)* function.

tau
Array resulting after call to the Group API of the *geqrf_batch (USM Version)* function.

group_count
Number of groups of parameters. Must be at least 0.

group_sizes
Array of `group_count` integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

scratchpad
Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by Group API of the *ungqr_batch_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a

Matrices pointed to by array `a` are overwritten by n_g leading columns of the $m_g \times m_g$ orthogonal matrices Q_i, where g is an index of group of parameters corresponding to Q_i.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value. If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

Strided API

The routine generates the wholes or parts of :math`m times m` unitary matrices Q_i of the batch of QR factorization formed by the Strided API of the *geqrf_batch (USM Version)*.

Usually Q_i is determined from the QR factorization of an $m \times p$ matrix A_i with :math`m ge p`.

To compute the whole matrices Q_i, use:

```
ungqr_batch(queue, m, m, p, a, ...)
```

To compute the leading p columns of Q_i (which form an orthonormal basis in the space spanned by the columns of A_i):

```
ungqr_batch(queue, m, p, p, a, ...)
```

To compute the matrices $Q_i{}^k$ of the QR factorizations of leading k columns of the matrices A_i:

```
ungqr_batch(queue, m, m, k, a, ...)
```

To compute the leading k columns of Q_i^k (which form an orthonormal basis in the space spanned by leading k columns of the matrices A_i):

```
ungqr_batch(queue, m, k, k, a, ...)
```

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event ungqr_batch(cl::sycl::queue &queue, std::int64_t m, std::int64_t n,␣
↪std::int64_t k, T *a, std::int64_t lda, std::int64_t stride_a, T *tau, std::int64_t␣
↪stride_tau, std::int64_t batch_size, T *scratchpad, std::int64_t scratchpad_size,␣
↪const std::vector<cl::sycl::event> &events = {})
};
```

Input Parameters

queue
Device queue where calculations will be performed.

m
Number of rows in the matrices A_i ($0 \leq m$).

n
Number of columns in the matrices A_i ($0 \leq n$).

k
Number of elementary reflectors whose product defines the matrices Q_i ($0 \leq k \leq n$).

a
Array resulting after call to the Strided API of the *geqrf_batch (USM Version)* function.

lda
Leading dimension of A_i (lda $\leq m$).

stride_a
Stride between the beginnings of matrices A_i inside the batch array `a`.

tau
Array resulting after call to the Strided API of the *geqrf_batch (USM Version)* function.

stride_tau
Stride between the beginnings of arrays tau_i inside the array `tau`.

batch_size
Number of problems in a batch.

scratchpad
Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size
Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by strided version of the Strided API of the *ungqr_batch_scratchpad_size* function.

events
List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a

Array data is overwritten by `a` batch of n leading columns of the $m \times m$ unitary matrices Q_i.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

Parent topic: *LAPACK-like Extensions Routines*

ungqr_batch_scratchpad_size

Computes size of scratchpad memory required for the *ungqr_batch* function.

Description

`ungqr_batch_scratchpad_size` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

Group API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *ungqr_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t ungqr_batch_scratchpad_size(cl::sycl::queue &queue, std::int64_t *m,␣
↪std::int64_t *n, std::int64_t *k, std::int64_t *lda, std::int64_t group_count,␣
↪std::int64_t *group_sizes)
}
```

Input Parameters

queue
Device queue where calculations will be performed.

m
Array of `group_count` m_g parameters.

n
Array of `group_count` n_g parameters.

k
Array of `group_count` k_g parameters.
Number of elementary reflectors whose product defines the matrices Q_i ($0 \leq k_g \leq n_g$).

lda
Array of leading dimensions of A_i.

group_count
Number of groups of parameters. Must be at least 0.

group_sizes
Array of `group_count` integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *ungqr_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Strided API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Strided API of the *ungqr_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t ungqr_batch_scratchpad_size(cl::sycl::queue &queue, std::int64_t m,␣
↪std::int64_t n, std::int64_t k, std::int64_t lda, std::int64_t stride_a, std::int64_t␣
↪stride_tau, std::int64_t batch_size)
};
```

Input Parameters

queue
Device queue where calculations will be performed.

m
Number of rows in the matrices A_i ($0 \leq m$).

n
Number of columns in the matrices A_i ($0 \leq n$).

k
Number of elementary reflectors whose product defines the matrices Q_i ($0 \leq k \leq n$).

lda
Leading dimensions of A_i (lda $\leq m$).

stride_a
Stride between the beginnings of matrices A_i inside the batch array `a`.

stride_tau
Stride between the beginnings of arrays τ_i inside the array `tau`.

batch_size
Number of problems in a batch.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Strided API of the *ungqr_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Parent topic: *LAPACK-like Extensions Routines*

Note

Different arrays used as parameters to oneMKL LAPACK routines must not overlap.

Warning

LAPACK routines assume that input matrices do not contain IEEE 754 special values such as INF or NaN values. Using these special values may cause LAPACK to return unexpected results or become unstable.

Parent topic: *Dense Linear Algebra*

9.2.2 Sparse Linear Algebra

The oneAPI Math Kernel Library provides a C++ interface to a set of Sparse Linear Algebra routines using SYCL.

Sparse BLAS provides basic operations on sparse vectors and matrices. Most operations are split into three stages: query of the external workspace size, optimization stage and execution. For a given configuration, the first two stages would typically be called once for a set of input arguments and the execution stage may be called multiple times. During the optimization stage, the API may inspect the matrix properties including size, sparsity pattern and available parallelism, and may apply matrix format or structure changes to enable a more optimized algorithm. User-provided matrix data remain unmodified if such optimizations are made. In the execution stage, multiple routine calls can take advantage of the optimization stage data in order to improve performance. Each operation has a descriptor type that is used to carry information across the different stages.

Sparse BLAS

Sparse BLAS routines provide basic operations on sparse vectors and matrices.

Routines and Objects	Description
Data handles	Matrix and vector handle types
spmm	Compute the product of a sparse matrix with a dense matrix
spmv	Compute the product of a sparse matrix with a dense vector
spsv	Solve a triangular sparse linear system

Data handles

Dense vector handle

Definition

```
namespace oneapi::mkl::sparse {

    struct dense_vector_handle;
    using dense_vector_handle_t = dense_vector_handle*;

}
```

Description

Defines `dense_vector_handle_t` as an opaque pointer to the incomplete type `dense_vector_handle`. Each backend may provide a different implementation of the type `dense_vector_handle`.

See related functions:

- *init_dense_vector*
- *set_dense_vector_data*
- *release_dense_vector*

Dense matrix handle

Definition

```
namespace oneapi::mkl::sparse {

    struct dense_matrix_handle;
    using dense_matrix_handle_t = dense_matrix_handle*;

}
```

Description

Defines `dense_matrix_handle_t` as an opaque pointer to the incomplete type `dense_matrix_handle`. Each backend may provide a different implementation of the type `dense_matrix_handle`.

See related functions:

- *init_dense_matrix*
- *set_dense_matrix_data*
- *release_dense_matrix*

Sparse matrix handle

Definition

```
namespace oneapi::mkl::sparse {

    struct matrix_handle;
    using matrix_handle_t = matrix_handle*;

}
```

Description

Defines `matrix_handle_t` as an opaque pointer to the incomplete type `matrix_handle`. Each backend may provide a different implementation of the type `matrix_handle`.

See related functions:

- *init_coo_matrix*
- *init_csr_matrix*
- *set_coo_matrix_data*
- *set_csr_matrix_data*
- *set_matrix_property*
- *release_sparse_matrix*

See a description of the supported *sparse formats*.

init_dense_vector

Initializes a `dense_vector_handle_t` object with the provided data.

Description and Assumptions

The `oneapi::mkl::sparse::init_dense_vector` function initializes the `dense_vector_handle_t` object with the provided data.

In the case of buffers, the reference count of the provided buffer is incremented which extends the lifetime of the underlying buffer until the dense vector handle is destroyed with `release_dense_vector` or the data is reset with `set_dense_vector_data`.

In the case of USM, the object does not take ownership of the data.

See *Dense vector handle*.

init_dense_vector (Buffer version)

Syntax

```
namespace oneapi::mkl::sparse {

    template <typename dataType>
    void init_dense_vector (sycl::queue                                  &queue,
                            oneapi::mkl::sparse::dense_vector_handle_t *p_dvhandle,
                            std::int64_t                                 size,
                            sycl::buffer<dataType, 1>                    val);

}
```

Template parameters

dataType
See *supported template types*.

Input parameters

queue
The SYCL command queue which will be used for SYCL kernels execution.

p_dvhandle
The address of the `p_dvhandle` object to be initialized. Must only be called on an uninitialized `dense_vector_handle_t` object.

size
Number of elements of the provided data `val`. Must be at least 1.

val
Buffer of length at least `size`. Holds the data to initialize `p_dvhandle` with.

Output parameters

p_dvhandle
On return, the address is updated to point to a newly allocated and initialized `dense_vector_handle_t` object that can be filled and used to perform sparse BLAS operations.

Throws

This function shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::unsupported_device

init_dense_vector (USM version)

Syntax

```
namespace oneapi::mkl::sparse {

    template <typename dataType>
    void init_dense_vector (sycl::queue                                  &queue,
                            oneapi::mkl::sparse::dense_vector_handle_t *p_dvhandle,
                            std::int64_t                                 size,
                            dataType                                     *val);

}
```

Template parameters

dataType
See *supported template types*.

Input parameters

queue
The SYCL command queue which will be used for SYCL kernels execution.

p_dvhandle
The address of the `p_dvhandle` object to be initialized. Must only be called on an uninitialized `dense_vector_handle_t` object.

size
Number of elements of the provided data `val`. Must be at least 1.

val
USM pointer of length at least `size`. Holds the data to initialize `p_dvhandle` with. The data must be accessible on the device. Using a USM pointer with a smaller allocated memory size is undefined behavior.

Output parameters

p_dvhandle
On return, the address is updated to point to a newly allocated and initialized `dense_vector_handle_t` object that can be filled and used to perform sparse BLAS operations.

Throws

This function shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::unsupported_device

Parent topic: *Data handles*

init_dense_matrix

Initializes a `dense_matrix_handle_t` object with the provided data.

Description and Assumptions

The `oneapi::mkl::sparse::init_dense_matrix` function initializes the `dense_matrix_handle_t` object with the provided data.

In the case of buffers, the reference count of the provided buffer is incremented which extends the lifetime of the underlying buffer until the dense matrix handle is destroyed with `release_dense_matrix` or the data is reset with `set_dense_matrix_data`.

In the case of USM, the object does not take ownership of the data.

See *Dense matrix handle*.

init_dense_matrix (Buffer version)

Syntax

```
namespace oneapi::mkl::sparse {

    template <typename dataType>
    void init_dense_matrix (sycl::queue                                  &queue,
                            oneapi::mkl::sparse::dense_matrix_handle_t *p_dmhandle,
                            std::int64_t                                 num_rows,
                            std::int64_t                                 num_cols,
                            std::int64_t                                 ld,
                            layout                                       dense_layout,
                            sycl::buffer<dataType, 1>                    val);

}
```

Template parameters

dataType
: See *supported template types*.

Input parameters

queue
: The SYCL command queue which will be used for SYCL kernels execution.

p_dmhandle
: The address of the `p_dmhandle` object to be initialized. Must only be called on an uninitialized `dense_matrix_handle_t` object.

num_rows
: Number of rows of the provided data `val`. Must be at least 1.

num_cols
: Number of columns of the provided data `val`. Must be at least 1.

ld
: Leading dimension of the provided data `val`. Must be at least `num_rows` if column major layout is used or at least `num_cols` if row major layout is used.

dense_layout
Specify whether the data uses row major or column major.

val
Buffer of length at least `ld*num_cols` if column major is used or `ld*num_rows` if row major is used. Holds the data to initialize `p_dmhandle` with.

Output parameters

p_dmhandle
On return, the address is updated to point to a newly allocated and initialized `dense_matrix_handle_t` object that can be filled and used to perform sparse BLAS operations.

Throws

This function shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::unsupported_device

init_dense_matrix (USM version)

Syntax

```
namespace oneapi::mkl::sparse {

    template <typename dataType>
    void init_dense_matrix (sycl::queue                                  &queue,
                            oneapi::mkl::sparse::dense_matrix_handle_t *p_dmhandle,
                            std::int64_t                                 num_rows,
                            std::int64_t                                 num_cols,
                            std::int64_t                                 ld,
                            layout                                       dense_layout,
                            dataType                                     *val);

}
```

Template parameters

dataType
See *supported template types*.

Input parameters

queue
The SYCL command queue which will be used for SYCL kernels execution.

p_dmhandle
The address of the `p_dmhandle` object to be initialized. Must only be called on an uninitialized `dense_matrix_handle_t` object.

num_rows
Number of rows of the provided data `val`. Must be at least 1.

num_cols
Number of columns of the provided data `val`. Must be at least 1.

ld
Leading dimension of the provided data `val`. Must be at least `num_rows` if column major layout is used or at least `num_cols` if row major layout is used.

dense_layout
Specify whether the data uses row major or column major.

val
USM pointer of length at least `ld*num_cols` if column major is used or `ld*num_rows` if row major is used. Holds the data to initialize `p_dmhandle` with. The data must be accessible on the device. Using a USM pointer with a smaller allocated memory size is undefined behavior.

Output parameters

p_dmhandle
On return, the address is updated to point to a newly allocated and initialized `dense_matrix_handle_t` object that can be filled and used to perform sparse BLAS operations.

Throws

This function shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::unsupported_device

Parent topic: *Data handles*

init_coo_matrix

Initializes a `matrix_handle_t` object with the provided COO data.

Description and Assumptions

The `oneapi::mkl::sparse::init_coo_matrix` function initializes the `matrix_handle_t` object with the provided data.

In the case of buffers, the reference count of the provided buffer is incremented which extends the lifetime of the underlying buffer until the sparse matrix handle is destroyed with `release_sparse_matrix` or the data is reset with `set_coo_matrix_data`.

In the case of USM, the object does not take ownership of the data.

The `oneapi::mkl::sparse::init_coo_matrix` function defined below takes in the number of non-zero elements in the sparse matrix as an argument. However, in certain math operations where the output is a sparse matrix, e.g., sparse matrix addition (sparse matrix + sparse matrix = sparse matrix), and multiplication of two sparse matrices, the number of non-zero elements in the output sparse matrix are not known in advance and must be calculated as part of the operation API. Such APIs are currently not part of the oneMKL Specification, but will be added in the future. This behavior is currently left to be implementation-defined, but may be clarified in the oneMKL Specification in the future.

See *Sparse matrix handle*.

init_coo_matrix (Buffer version)

Syntax

```
namespace oneapi::mkl::sparse {

    template <typename dataType, typename indexType>
    void init_coo_matrix (sycl::queue                              &queue,
                          oneapi::mkl::sparse::matrix_handle_t *p_smhandle,
                          std::int64_t                             num_rows,
                          std::int64_t                             num_cols,
                          std::int64_t                             nnz,
                          index_base                               index,
                          sycl::buffer<indexType, 1>               row_ind,
                          sycl::buffer<indexType, 1>               col_ind,
                          sycl::buffer<dataType, 1>                val);

}
```

Template parameters

dataType
See *supported template types*.

indexType
See *supported template types*.

Input parameters

queue
The SYCL command queue which will be used for SYCL kernels execution.

p_smhandle
The address of the `p_smhandle` object to be initialized. Must only be called on an uninitialized `matrix_handle_t` object.

num_rows
Number of rows of the provided data `val`. Must be at least 0.

num_cols
Number of columns of the provided data `val`. Must be at least 0.

nnz
The number of explicit entries, also known as Number of Non-Zero elements. Must be at least 0.

index
Indicates how input arrays are indexed. The possible options are described in *index_base* enum class.

row_ind
Buffer of length at least `nnz` containing the row indices in `index`-based numbering. Refer to *COO* format for detailed description of `row_ind`.

col_ind
Buffer of length at least `nnz` containing the column indices in `index`-based numbering. Refer to *COO* format for detailed description of `col_ind`.

val
Buffer of length at least `nnz`. Contains the data of the input matrix which is not implicitly zero. The remaining input values are implicit zeros. Refer to *COO* format for detailed description of `val`.

Output parameters

p_smhandle
On return, the address is updated to point to a newly allocated and initialized `matrix_handle_t` object that can be filled and used to perform sparse BLAS operations.

Notes

- The parameters `num_rows`, `num_cols` and `nnz` may be zero if and only if `row_ind`, `col_ind` and `val` are zero-sized, otherwise they must be strictly greater than zero.

Throws

This function shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::unsupported_device

init_coo_matrix (USM version)

Syntax

```
namespace oneapi::mkl::sparse {

    template <typename dataType, typename indexType>
    void init_coo_matrix (sycl::queue                              &queue,
                          oneapi::mkl::sparse::matrix_handle_t *p_smhandle,
                          std::int64_t                             num_rows,
                          std::int64_t                             num_cols,
                          std::int64_t                             nnz,
                          index_base                               index,
                          indexType                                *row_ind,
                          indexType                                *col_ind,
                          dataType                                 *val);

}
```

Template parameters

dataType
See *supported template types*.

indexType
See *supported template types*.

Input parameters

queue
The SYCL command queue which will be used for SYCL kernels execution.

p_smhandle
The address of the `p_smhandle` object to be initialized. Must only be called on an uninitialized `matrix_handle_t` object.

num_rows
Number of rows of the provided data `val`. Must be at least 0.

num_cols
Number of columns of the provided data `val`. Must be at least 0.

nnz
The number of explicit entries, also known as Number of Non-Zero elements. Must be at least 0.

index
Indicates how input arrays are indexed. The possible options are described in *index_base* enum class.

row_ind
USM pointer of length at least `nnz` containing the row indices in `index`-based numbering. Refer to *COO* format for detailed description of `row_ind`. The data must be accessible on the device.

col_ind
USM pointer of length at least `nnz` containing the column indices in `index`-based numbering. Refer to *COO* format for detailed description of `col_ind`. The data must be accessible on the device.

val
USM pointer of length at least `nnz`. Contains the data of the input matrix which is not implicitly zero. The remaining input values are implicit zeros. Refer to *COO* format for detailed description of `val`. The data must be accessible on the device. Using a USM pointer with a smaller allocated memory size is undefined behavior.

Output parameters

p_smhandle
On return, the address is updated to point to a newly allocated and initialized `matrix_handle_t` object that can be filled and used to perform sparse BLAS operations.

Notes

- The parameters `num_rows`, `num_cols` and `nnz` may be zero if and only if `row_ind`, `col_ind` and `val` are null pointers, otherwise they must be strictly greater than zero.

Throws

This function shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::unsupported_device

Parent topic: *Data handles*

init_csr_matrix

Initializes a `matrix_handle_t` object with the provided Compressed Sparse Row (CSR) data.

Description and Assumptions

The `oneapi::mkl::sparse::init_csr_matrix` function initializes the `matrix_handle_t` object with the provided data.

In the case of buffers, the reference count of the provided buffer is incremented which extends the lifetime of the underlying buffer until the sparse matrix handle is destroyed with `release_sparse_matrix` or the data is reset with `set_csr_matrix_data`.

In the case of USM, the object does not take ownership of the data.

The `oneapi::mkl::sparse::init_csr_matrix` function defined below takes in the number of non-zero elements in the sparse matrix as an argument. However, in certain math operations where the output is a sparse matrix, e.g., sparse matrix addition (sparse matrix + sparse matrix = sparse matrix), and multiplication of two sparse matrices, the number of non-zero elements in the output sparse matrix are not known in advance and must be calculated as part of the operation API. Such APIs are currently not part of the oneMKL Specification, but will be added in the future. This behavior is currently left to be implementation-defined, but may be clarified in the oneMKL Specification in the future.

See *Sparse matrix handle*.

init_csr_matrix (Buffer version)

Syntax

```
namespace oneapi::mkl::sparse {

    template <typename dataType, typename indexType>
    void init_csr_matrix (sycl::queue                              &queue,
                          oneapi::mkl::sparse::matrix_handle_t *p_smhandle,
                          std::int64_t                             num_rows,
                          std::int64_t                             num_cols,
                          std::int64_t                             nnz,
                          index_base                               index,
                          sycl::buffer<indexType, 1>               row_ptr,
                          sycl::buffer<indexType, 1>               col_ind,
                          sycl::buffer<dataType, 1>                val);

}
```

Template parameters

dataType
See *supported template types*.

indexType
See *supported template types*.

Input parameters

queue
The SYCL command queue which will be used for SYCL kernels execution.

p_smhandle
The address of the `p_smhandle` object to be initialized. Must only be called on an uninitialized `matrix_handle_t` object.

num_rows
Number of rows of the provided data `val`. Must be at least 0.

num_cols
Number of columns of the provided data `val`. Must be at least 0.

nnz
The number of explicit entries, also known as Number of Non-Zero elements. Must be at least 0.

index
Indicates how input arrays are indexed. The possible options are described in *index_base* enum class.

row_ptr
Buffer of length at least `num_rows+1`. Refer to *CSR* format for detailed description of `row_ptr`.

col_ind
Buffer of length at least `nnz` containing the column indices in `index`-based numbering. Refer to *CSR* format for detailed description of `col_ind`.

val
Buffer of length at least `nnz`. Contains the data of the input matrix which is not implicitly zero. The remaining input values are implicit zeros. Refer to *CSR* format for detailed description of `val`.

Output parameters

p_smhandle
On return, the address is updated to point to a newly allocated and initialized `matrix_handle_t` object that can be filled and used to perform sparse BLAS operations.

Notes

- The parameters `num_rows`, `num_cols` and `nnz` may be zero if and only if `row_ptr`, `col_ind` and `val` are zero-sized, otherwise they must be strictly greater than zero.

Throws

This function shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::unsupported_device

init_csr_matrix (USM version)

Syntax

```
namespace oneapi::mkl::sparse {

    template <typename dataType, typename indexType>
    void init_csr_matrix (sycl::queue                              &queue,
                          oneapi::mkl::sparse::matrix_handle_t *p_smhandle,
                          std::int64_t                             num_rows,
                          std::int64_t                             num_cols,
                          std::int64_t                             nnz,
                          index_base                               index,
                          indexType                                *row_ptr,
                          indexType                                *col_ind,
                          dataType                                 *val);

}
```

Template parameters

dataType
See *supported template types*.

indexType
See *supported template types*.

Input parameters

queue
The SYCL command queue which will be used for SYCL kernels execution.

p_smhandle
The address of the `p_smhandle` object to be initialized. Must only be called on an uninitialized `matrix_handle_t` object.

num_rows
Number of rows of the provided data `val`. Must be at least 0.

num_cols
Number of columns of the provided data `val`. Must be at least 0.

nnz
The number of explicit entries, also known as Number of Non-Zero elements. Must be at least 0.

index
Indicates how input arrays are indexed. The possible options are described in *index_base* enum class.

row_ptr
USM pointer of length at least `num_rows+1`. Refer to *CSR* format for detailed description of `row_ptr`. The data must be accessible on the device.

col_ind
USM pointer of length at least `nnz` containing the column indices in `index`-based numbering. Refer to *CSR* format for detailed description of `col_ind`. The data must be accessible on the device.

val
USM pointer of length at least `nnz`. Contains the data of the input matrix which is not implicitly zero. The remaining input values are implicit zeros. Refer to *CSR* format for detailed description of `val`. The data must be accessible on the device. Using a USM pointer with a smaller allocated memory size is undefined behavior.

Output parameters

p_smhandle
On return, the address is updated to point to a newly allocated and initialized `matrix_handle_t` object that can be filled and used to perform sparse BLAS operations.

Notes

- The parameters `num_rows`, `num_cols` and `nnz` may be zero if and only if `row_ptr`, `col_ind` and `val` are null pointers, otherwise they must be strictly greater than zero.

Throws

This function shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::unsupported_device

Parent topic: *Data handles*

release_dense_vector

Destroys a `dense_vector_handle_t` object.

Description and Assumptions

The `oneapi::mkl::sparse::release_dense_vector` function frees the resources allocated for the handle.

If a buffer was provided, its reference count is decremented.

If a USM pointer was provided, the data is not free'd.

Syntax

```
namespace oneapi::mkl::sparse {

    sycl::event release_dense_vector (sycl::queue                                &queue,
                                      oneapi::mkl::sparse::dense_vector_handle_t␣
↪dvhandle,
                                      const std::vector<sycl::event>             &
↪dependencies = {});

}
```

Input parameters

queue
: The SYCL command queue which will be used for SYCL kernels execution.

dvhandle
: Handle initialized with *init_dense_vector*.

dependencies
: List of events to depend on before starting asynchronous tasks that access data on the device. Defaults to no dependencies.

Return Values

Output event that can be waited upon or added as a dependency for the completion of the function.

Throws

This function shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::unsupported_device

Parent topic: *Data handles*

release_dense_matrix

Destroys a `dense_matrix_handle_t` object.

Description and Assumptions

The `oneapi::mkl::sparse::release_dense_matrix` function frees the resources allocated for the handle.

If a buffer was provided, its reference count is decremented.

If a USM pointer was provided, the data is not free'd.

Syntax

```
namespace oneapi::mkl::sparse {

    sycl::event release_dense_matrix (sycl::queue                                  &queue,
                                      oneapi::mkl::sparse::dense_matrix_handle_t␣
↪dmhandle,
                                      const std::vector<sycl::event>               &
↪dependencies = {});

}
```

Input parameters

queue
The SYCL command queue which will be used for SYCL kernels execution.

dmhandle
Handle initialized with *init_dense_matrix*.

dependencies
List of events to depend on before starting asynchronous tasks that access data on the device. Defaults to no dependencies.

Return Values

Output event that can be waited upon or added as a dependency for the completion of the function.

Throws

This function shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::unsupported_device

Parent topic: *Data handles*

release_sparse_matrix

Destroys a `matrix_handle_t` object.

Description and Assumptions

The `oneapi::mkl::sparse::release_sparse_matrix` function frees the resources allocated for the handle.

If a buffer was provided, its reference count is decremented.

If a USM pointer was provided, the data is not free'd.

Syntax

```
namespace oneapi::mkl::sparse {

    sycl::event release_sparse_matrix (sycl::queue                          &queue,
                                       oneapi::mkl::sparse::matrix_handle_t smhandle,
                                       const std::vector<sycl::event>       &
↪dependencies = {});

}
```

Input parameters

queue
The SYCL command queue which will be used for SYCL kernels execution.

smhandle
Handle initialized with *init_csr_matrix* or *init_coo_matrix*.

dependencies
List of events to depend on before starting asynchronous tasks that access data on the device. Defaults to no dependencies.

Return Values

Output event that can be waited upon or added as a dependency for the completion of the function.

Throws

This function shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::unsupported_device

Parent topic: *Data handles*

set_dense_vector_data

Reset the data of a `dense_vector_handle_t` object.

Description and Assumptions

The `oneapi::mkl::sparse::set_dense_vector_data` function sets new data to the `dense_vector_handle_t` object.

In the case of buffers, the reference count of the provided buffer is incremented which extends the lifetime of the underlying buffer until the `dvhandle` is destroyed with `release_dense_vector` or the data is reset with `set_dense_vector_data`.

In the case of USM, the object does not take ownership of the data.

Also see *init_dense_vector*.

set_dense_vector_data (Buffer version)

Syntax

```
namespace oneapi::mkl::sparse {

    template <typename dataType>
    void set_dense_vector_data (sycl::queue                                &queue,
                                oneapi::mkl::sparse::dense_vector_handle_t dvhandle,
                                std::int64_t                               size,
                                sycl::buffer<dataType, 1>                  val);

}
```

Template parameters

dataType
See *supported template types*. Can be a different type than what was used when creating the `dense_vector_handle_t`.

Input parameters

queue
The SYCL command queue which will be used for SYCL kernels execution.

dvhandle
Handle already initialized with *init_dense_vector*.

size
Number of elements of the provided data `val`. Must be at least 0.

val
Buffer of length at least `size`.

Throws

This function shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

set_dense_vector_data (USM version)

Syntax

```
namespace oneapi::mkl::sparse {

    template <typename dataType>
    void set_dense_vector_data (sycl::queue                                &queue,
                                oneapi::mkl::sparse::dense_vector_handle_t dvhandle,
                                std::int64_t                               size,
                                dataType                                   *val);

}
```

Template parameters

dataType
See *supported template types*. Can be a different type than what was used when creating the `dense_vector_handle_t`.

Input parameters

queue
The SYCL command queue which will be used for SYCL kernels execution.

dvhandle
Handle already initialized with *init_dense_vector*.

size
Number of elements of the provided data `val`. Must be at least 1.

val
USM pointer of length at least `size`. The data must be accessible on the device. Using a USM pointer with a smaller allocated memory size is undefined behavior.

Throws

This function shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

Parent topic: *Data handles*

set_dense_matrix_data

Reset the data of a `dense_matrix_handle_t` object.

Description and Assumptions

The `oneapi::mkl::sparse::set_dense_matrix_data` function sets new data to the `dense_matrix_handle_t` object with the provided data.

In the case of buffers, the reference count of the provided buffer is incremented which extends the lifetime of the underlying buffer until the `dmhandle` is destroyed with `release_dense_matrix` or the data is reset with `set_dense_matrix_data`.

In the case of USM, the object does not take ownership of the data.

Also see *init_dense_matrix*.

set_dense_matrix_data (Buffer version)

Syntax

```
namespace oneapi::mkl::sparse {

    template <typename dataType>
    void set_dense_matrix_data (sycl::queue                                  &queue,
                                oneapi::mkl::sparse::dense_matrix_handle_t dmhandle,
                                std::int64_t                                 num_rows,
                                std::int64_t                                 num_cols,
                                std::int64_t                                 ld,
                                layout                                       dense_layout,
                                sycl::buffer<dataType, 1>                    val);

}
```

Template parameters

dataType
: See *supported template types*. Can be a different type than what was used when creating the `dense_matrix_handle_t`.

Input parameters

queue
: The SYCL command queue which will be used for SYCL kernels execution.

dmhandle
: Handle already initialized with *init_dense_matrix*.

num_rows
: Number of rows of the provided data `val`. Must be at least 0.

num_cols
: Number of columns of the provided data `val`. Must be at least 0.

ld
: Leading dimension of the provided data `val`. Must be at least `num_rows` if column major layout is used or at least `num_cols` if row major layout is used.

dense_layout
: Specify whether the data uses row major or column major.

val
: Buffer of length at least `ld*num_cols` if column major is used or `ld*num_rows` if row major is used.

Throws

This function shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

set_dense_matrix_data (USM version)

Syntax

```
namespace oneapi::mkl::sparse {

    template <typename dataType>
    void set_dense_matrix_data (sycl::queue                                  &queue,
                                oneapi::mkl::sparse::dense_matrix_handle_t dmhandle,
                                std::int64_t                                 num_rows,
                                std::int64_t                                 num_cols,
                                std::int64_t                                 ld,
                                layout                                       dense_layout,
                                dataType                                     *val);

}
```

Template parameters

dataType
See *supported template types*. Can be a different type than what was used when creating the `dense_matrix_handle_t`.

Input parameters

queue
The SYCL command queue which will be used for SYCL kernels execution.

dmhandle
Handle already initialized with *init_dense_matrix*.

num_rows
Number of rows of the provided data `val`. Must be at least 1.

num_cols
Number of columns of the provided data `val`. Must be at least 1.

ld
Leading dimension of the provided data `val`. Must be at least `num_rows` if column major layout is used or at least `num_cols` if row major layout is used.

dense_layout
Specify whether the data uses row major or column major.

val
USM pointer of length at least `ld*num_cols` if column major is used or `ld*num_rows` if row major is used. The data must be accessible on the device. Using a USM pointer with a smaller allocated memory size is undefined behavior.

Throws

This function shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

Parent topic: *Data handles*

set_coo_matrix_data

Reset the data of a `matrix_handle_t` object with the provided COO data.

Description and Assumptions

The `oneapi::mkl::sparse::set_coo_matrix_data` function sets new data to the `matrix_handle_t` object with the provided data.

In the case of buffers, the reference count of the provided buffer is incremented which extends the lifetime of the underlying buffer until the `smhandle` is destroyed with `release_sparse_matrix` or the data is reset with `set_coo_matrix_data`.

In the case of USM, the object does not take ownership of the data.

Also see *init_coo_matrix*.

set_coo_matrix_data (Buffer version)

Syntax

```
namespace oneapi::mkl::sparse {

    template <typename dataType, typename indexType>
    void set_coo_matrix_data (sycl::queue                          &queue,
                              oneapi::mkl::sparse::matrix_handle_t smhandle,
                              std::int64_t                         num_rows,
                              std::int64_t                         num_cols,
                              std::int64_t                         nnz,
```

(continues on next page)

(continued from previous page)

```
                                index_base                              index,
                                sycl::buffer<indexType, 1>              row_ind,
                                sycl::buffer<indexType, 1>              col_ind,
                                sycl::buffer<dataType, 1>               val);

}
```

Template parameters

dataType
See *supported template types*. Can be a different type than what was used when creating the `matrix_handle_t`.

indexType
See *supported template types*. Can be a different type than what was used when creating the `matrix_handle_t`.

Input parameters

queue
The SYCL command queue which will be used for SYCL kernels execution.

smhandle
Handle already initialized with *init_coo_matrix*.

num_rows
Number of rows of the provided data `val`. Must be at least 0.

num_cols
Number of columns of the provided data `val`. Must be at least 0.

nnz
The number of explicit entries, also known as Number of Non-Zero elements. Must be at least 0.

index
Indicates how input arrays are indexed. The possible options are described in *index_base* enum class.

row_ind
Buffer of length at least `nnz` containing the row indices in `index`-based numbering. Refer to *COO* format for detailed description of `row_ind`.

col_ind
Buffer of length at least `nnz` containing the column indices in `index`-based numbering. Refer to *COO* format for detailed description of `col_ind`.

val
Buffer of length at least `nnz`. Contains the data of the input matrix which is not implicitly zero. The remaining input values are implicit zeros. Refer to *COO* format for detailed description of `val`.

Notes

- The parameters `num_rows`, `num_cols` and `nnz` may be zero if and only if `row_ind`, `col_ind` and `val` are zero-sized, otherwise they must be strictly greater than zero.

Throws

This function shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

set_coo_matrix_data (USM version)

Syntax

```
namespace oneapi::mkl::sparse {

    template <typename dataType, typename indexType>
    void set_coo_matrix_data (sycl::queue                              &queue,
                              oneapi::mkl::sparse::matrix_handle_t smhandle,
                              std::int64_t                             num_rows,
                              std::int64_t                             num_cols,
                              std::int64_t                             nnz,
                              index_base                               index,
                              indexType                                *row_ind,
                              indexType                                *col_ind,
                              dataType                                 *val);

}
```

Template parameters

dataType
: See *supported template types*. Can be a different type than what was used when creating the `matrix_handle_t`.

indexType
: See *supported template types*. Can be a different type than what was used when creating the `matrix_handle_t`.

Input parameters

queue
The SYCL command queue which will be used for SYCL kernels execution.

smhandle
Handle already initialized with *init_coo_matrix*.

num_rows
Number of rows of the provided data `val`. Must be at least 0.

num_cols
Number of columns of the provided data `val`. Must be at least 0.

nnz
The number of explicit entries, also known as Number of Non-Zero elements. Must be at least 0.

index
Indicates how input arrays are indexed. The possible options are described in *index_base* enum class.

row_ind
USM pointer of length at least `nnz` containing the row indices in `index`-based numbering. Refer to *COO* format for detailed description of `row_ind`. The data must be accessible on the device.

col_ind
USM pointer of length at least `nnz` containing the column indices in `index`-based numbering. Refer to *COO* format for detailed description of `col_ind`. The data must be accessible on the device.

val
USM pointer of length at least `nnz`. Contains the data of the input matrix which is not implicitly zero. The remaining input values are implicit zeros. Refer to *COO* format for detailed description of `val`. The data must be accessible on the device. Using a USM pointer with a smaller allocated memory size is undefined behavior.

Notes

- The parameters `num_rows`, `num_cols` and `nnz` may be zero if and only if `row_ind`, `col_ind` and `val` are null pointers, otherwise they must be strictly greater than zero.

Throws

This function shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

Parent topic: *Data handles*

set_csr_matrix_data

Reset the data of a `matrix_handle_t` object with the provided CSR data.

Description and Assumptions

The `oneapi::mkl::sparse::set_csr_matrix_data` function sets new data to the `matrix_handle_t` object with the provided data.

In the case of buffers, the reference count of the provided buffer is incremented which extends the lifetime of the underlying buffer until the `smhandle` is destroyed with `release_sparse_matrix` or the data is reset with `set_csr_matrix_data`.

In the case of USM, the object does not take ownership of the data.

Also see *init_csr_matrix*.

set_csr_matrix_data (Buffer version)

Syntax

```
namespace oneapi::mkl::sparse {

    template <typename dataType, typename indexType>
    void set_csr_matrix_data (sycl::queue                              &queue,
                              oneapi::mkl::sparse::matrix_handle_t smhandle,
                              std::int64_t                             num_rows,
                              std::int64_t                             num_cols,
                              std::int64_t                             nnz,
                              index_base                               index,
                              sycl::buffer<indexType, 1>               row_ptr,
                              sycl::buffer<indexType, 1>               col_ind,
                              sycl::buffer<dataType, 1>                val);

}
```

Template parameters

dataType
: See *supported template types*. Can be a different type than what was used when creating the `matrix_handle_t`.

indexType
: See *supported template types*. Can be a different type than what was used when creating the `matrix_handle_t`.

Input parameters

queue
The SYCL command queue which will be used for SYCL kernels execution.

smhandle
Handle already initialized with *init_csr_matrix*.

num_rows
Number of rows of the provided data `val`. Must be at least 0.

num_cols
Number of columns of the provided data `val`. Must be at least 0.

nnz
The number of explicit entries, also known as Number of Non-Zero elements. Must be at least 0.

index
Indicates how input arrays are indexed. The possible options are described in *index_base* enum class.

row_ptr
Buffer of length at least `num_rows+1`. Refer to *CSR* format for detailed description of `row_ptr`.

col_ind
Buffer of length at least `nnz` containing the column indices in `index`-based numbering. Refer to *CSR* format for detailed description of `col_ind`.

val
Buffer of length at least `nnz`. Contains the data of the input matrix which is not implicitly zero. The remaining input values are implicit zeros. Refer to *CSR* format for detailed description of `val`.

Notes

- The parameters `num_rows`, `num_cols` and `nnz` may be zero if and only if `row_ptr`, `col_ind` and `val` are zero-sized, otherwise they must be strictly greater than zero.

Throws

This function shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

set_csr_matrix_data (USM version)

Syntax

```
namespace oneapi::mkl::sparse {

    template <typename dataType, typename indexType>
    void set_csr_matrix_data (sycl::queue                          &queue,
                              oneapi::mkl::sparse::matrix_handle_t smhandle,
                              std::int64_t                         num_rows,
                              std::int64_t                         num_cols,
                              std::int64_t                         nnz,
                              index_base                           index,
                              indexType                            *row_ptr,
                              indexType                            *col_ind,
                              dataType                             *val);

}
```

Template parameters

dataType
See *supported template types*. Can be a different type than what was used when creating the `matrix_handle_t`.

indexType
See *supported template types*. Can be a different type than what was used when creating the `matrix_handle_t`.

Input parameters

queue
The SYCL command queue which will be used for SYCL kernels execution.

smhandle
Handle already initialized with *init_csr_matrix*.

num_rows
Number of rows of the provided data `val`. Must be at least 0.

num_cols
Number of columns of the provided data `val`. Must be at least 0.

nnz
The number of explicit entries, also known as Number of Non-Zero elements. Must be at least 0.

index
Indicates how input arrays are indexed. The possible options are described in *index_base* enum class.

row_ptr
USM pointer of length at least `num_rows+1`. Refer to *CSR* format for detailed description of `row_ptr`. The data must be accessible on the device.

col_ind
USM pointer of length at least `nnz` containing the column indices in `index`-based numbering. Refer to *CSR* format for detailed description of `col_ind`. The data must be accessible on the device.

val
: USM pointer of length at least `nnz`. Contains the data of the input matrix which is not implicitly zero. The remaining input values are implicit zeros. Refer to *CSR* format for detailed description of `val`. The data must be accessible on the device. Using a USM pointer with a smaller allocated memory size is undefined behavior.

Notes

- The parameters `num_rows`, `num_cols` and `nnz` may be zero if and only if `row_ptr`, `col_ind` and `val` are null pointers, otherwise they must be strictly greater than zero.

Throws

This function shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

Parent topic: *Data handles*

set_matrix_property

Matrix properties

```
namespace oneapi::mkl::sparse {

    enum class matrix_property {
       symmetric,
       sorted,
    };

}
```

Matrix properties are optional and "strong" guarantees. Unlike *matrix_view*, the user must ensure that the handle's data holds all the given properties. A property can be set as a hint for backends to optimize some operations. Multiple properties can be set to the same handle.

Value	Description
`symmetric`	Guarantees that the user-provided matrix data are symmetric, meaning the matrix is square, the user data contain both lower and upper triangular regions, and that its transpose is equal to itself.
`sorted`	Guarantees that the user-provided matrix data has some sorting property. For CSR this guarantees that the column indices are sorted in ascending order for a given row. For COO this guarantees that the indices are sorted by row then by column in ascending order.

set_matrix_property

Set a property to a `matrix_handle_t` object.

Description and Assumptions

The `oneapi::mkl::sparse::set_matrix_property` function sets a property to a matrix handle.

Syntax

```
namespace oneapi::mkl::sparse {

    bool set_matrix_property (sycl::queue                          &queue,
                              oneapi::mkl::sparse::matrix_handle_t smhandle,
                              matrix_property                      property);

}
```

Input parameters

queue
: The SYCL command queue which will be used for SYCL kernels execution.

smhandle
: Initialized sparse matrix handle.

property
: Matrix property to set.

Return Values

Return whether the property was set to the backend's handle. A backend may not have an equivalent property.

Throws

This function shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

Parent topic: *Data handles*

Sparse storage formats

There are a variety of matrix storage formats available for representing sparse matrices. Two popular formats are the coordinate (COO) format, and the Compressed Sparse Row (CSR) format.

In this specification, "non-zero" elements or "non-zero" entries refer to explicitly defined elements or entries which may take any value supported by the the *:ref:data type<onemkl_sparse_supported_types>*. Undefined elements or entries are implicitly zeros.

COO

The COO format is the simplest sparse matrix format, represented by three arrays, `row_ind`, `col_ind` and `val`, and an `index` parameter. The `i`-th defined element in the sparse matrix is represented by its row index, column index, and value, that is, (`row_ind[i]`, `col_ind[i]`, `val[i]`). The entries need not be in a sorted order, though performance of Sparse BLAS operations may be improved if they are sorted in some logical way, for instance by row index and then column index subordinate to each row set.

num_r	Number of rows in the sparse matrix.
num_c	Number of columns in the sparse matrix.
nnz	Number of non-zero entries in the sparse matrix. This is also the length of the `row_ind`, `col_ind` and `val` arrays.
index	Parameter that is used to specify whether the matrix has zero or one-based indexing.
val	An array of length `nnz` that contains the non-zero elements of the sparse matrix not necessarily in any sorted order.
row_in	An integer array of length `nnz`. Contains row indices for non-zero elements stored in the `val` array such that `row_ind[i]` is the row number (using zero- or one-based indexing) of the element of the sparse matrix stored in `val[i]`.
col_in	An integer array of length `nnz`. Contains column indices for non-zero elements stored in the `val` array such that `col_ind[i]` is the column number (using zero- or one-based indexing) of the element of the sparse matrix stored in `val[i]`.

A sparse matrix can be represented in a COO format in a following way (assuming one-based indexing):

$$A = \begin{pmatrix} 1 & 0 & 2 \\ 0 & -1 & 4 \\ 3 & 0 & 0 \end{pmatrix}$$

num_rows	3				
num_cols	3				
nnz	5				
index	1				
row_ind	1	1	2	2	3
col_ind	1	3	2	3	1
val	1	2	-1	4	3

CSR

The CSR format is one of the most popular sparse matrix storage formats, represented by three arrays, `row_ptr`, `col_ind` and `val`, and an `index` parameter.

num_	Number of rows in the sparse matrix.
num_	Number of columns in the sparse matrix.
nnz	Number of non-zero entries in the sparse matrix. This is also the length of the `col_ind` and `val` arrays.
index	Parameter that is used to specify whether the matrix has zero or one-based indexing.
val	An array of length `nnz` that contains the non-zero elements of the sparse matrix stored row by row.
col_i	An integer array of length `nnz`. Contains column indices for non-zero elements stored in the `val` array such that `col_ind[i]` is the column number (using zero- or one-based indexing) of the element of the sparse matrix stored in `val[i]`.
row_	An integer array of size equal to `num_rows + 1`. Element `j` of this integer array gives the position of the element in the `val` array that is first non-zero element in a row `j` of `A`. Note that this position is equal to `row_ptr[j] - index`. Last element of the `row_ptr` array (`row_ptr[num_rows]`) stores the sum of, number of non-zero elements and `index` (`nnz + index`).

A sparse matrix can be represented in a CSR format in a following way (assuming zero-based indexing):

$$A = \begin{pmatrix} 1 & 0 & 2 \\ 0 & -1 & 4 \\ 3 & 0 & 0 \end{pmatrix}$$

num_rows	3				
num_cols	3				
nnz	5				
index	0				
row_ptr	0	2	4	5	
col_ind	0	2	1	2	0
val	1	2	-1	4	3

Parent topic: *Data handles*

Parent topic: *Sparse BLAS*

spmm

Computes a sparse matrix by dense matrix product.

Description and Assumptions

The `oneapi::mkl::sparse::spmm` routine computes a sparse matrix by dense matrix product defined as:

$$C \leftarrow \alpha \cdot \mathrm{op}(A) \cdot \mathrm{op}(B) + \beta \cdot C$$

where:

α and β are scalars,
C is a dense matrix of size `m`-by-`n`,
$\mathrm{op}(A)$ is a transformed sparse matrix of size `m`-by-`k`,
$\mathrm{op}(B)$ is a transformed dense matrix of size `k`-by-`n`,
$\mathrm{op}()$ is the transform operation using the following description:

$$\mathrm{op}(A) = \begin{cases} A, & \text{oneapi::mkl::transpose::nontrans} \\ A^{\mathsf{T}}, & \text{oneapi::mkl::transpose::trans} \\ A^{\mathsf{H}}, & \text{oneapi::mkl::transpose::conjtrans} \end{cases}$$

spmm_descr

Definition

```
namespace oneapi::mkl::sparse {

    struct spmm_descr;
    using spmm_descr_t = spmm_descr*;

}
```

Description

Defines `spmm_descr_t` as an opaque pointer to the incomplete type `spmm_descr`. Each backend may provide a different implementation of the type `spmm_descr`. The `spmm_descr_t` object persists through the various stages of the spmm operation to house relevant state, optimizations and workspaces.

init_spmm_descr

Syntax

```
namespace oneapi::mkl::sparse {

    void init_spmm_descr (sycl::queue                          &queue,
                          oneapi::mkl::sparse::spmm_descr_t *p_spmm_descr);

}
```

Input parameters

queue
The SYCL command queue which will be used for SYCL kernels execution.

p_spmm_descr
The address of the `p_spmm_descr` object to be initialized. Must only be called on an uninitialized `spmm_descr_t` object.

Output parameters

p_spmm_descr
On return, the address is updated to point to a newly allocated and initialized `spmm_descr_t` object that can be used to perform spmm.

Throws

This function shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::unsupported_device

release_spmm_descr

Syntax

```
namespace oneapi::mkl::sparse {

    sycl::event release_spmm_descr (sycl::queue                        &queue,
                                    oneapi::mkl::sparse::spmm_descr_t spmm_descr,
                                    const std::vector<sycl::event>    &dependencies = {}
↪);

}
```

Input parameters

queue
The SYCL command queue which will be used for SYCL kernels execution.

spmm_descr
Descriptor initialized with `init_spmm_descr`.

dependencies
List of events to depend on before starting asynchronous tasks that access data on the device. Defaults to no dependencies.

Return Values

Output event that can be waited upon or added as a dependency for the completion of the function.

Throws

This function shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::unsupported_device

spmm_alg

Syntax

```
namespace oneapi::mkl::sparse {

    enum class spmm_alg {
        default_alg,
        no_optimize_alg,
        coo_alg1,
        coo_alg2,
        coo_alg3,
        coo_alg4,
        csr_alg1,
        csr_alg2,
        csr_alg3,
    };

}
```

Description

These algorithm enums are provided in case backends would like to implement various different algorithms for the operation. Behavior of the algorithms (e.g., bitwise reproducibility, atomics usage) and the preconditions to using specific algorithms (e.g. sortedness of matrix arrays) is implementation-defined and must be documented in the library implementing the oneAPI specification.

spmm

Syntax

```
namespace oneapi::mkl::sparse {

    void spmm_buffer_size(
        sycl::queue                                &queue,
        oneapi::mkl::transpose                     opA,
        oneapi::mkl::transpose                     opB,
        const void*                                alpha,
        oneapi::mkl::sparse::matrix_view           A_view,
        oneapi::mkl::sparse::matrix_handle_t       A_handle,
        oneapi::mkl::sparse::dense_matrix_handle_t B_handle,
        const void*                                beta,
        oneapi::mkl::sparse::dense_matrix_handle_t C_handle,
        oneapi::mkl::sparse::spmm_alg              alg,
        oneapi::mkl::sparse::spmm_descr_t          spmm_descr,
        std::size_t                                &temp_buffer_size);

    void spmm_optimize(
        sycl::queue                                &queue,
        oneapi::mkl::transpose                     opA,
        oneapi::mkl::transpose                     opB,
        const void*                                alpha,
        oneapi::mkl::sparse::matrix_view           A_view,
        oneapi::mkl::sparse::matrix_handle_t       A_handle,
        oneapi::mkl::sparse::dense_matrix_handle_t B_handle,
        const void*                                beta,
        oneapi::mkl::sparse::dense_matrix_handle_t C_handle,
        oneapi::mkl::sparse::spmm_alg              alg,
        oneapi::mkl::sparse::spmm_descr_t          spmm_descr,
        sycl::buffer<std::uint8_t, 1>              workspace);

    sycl::event spmm_optimize(
        sycl::queue                                &queue,
        oneapi::mkl::transpose                     opA,
        oneapi::mkl::transpose                     opB,
        const void*                                alpha,
        oneapi::mkl::sparse::matrix_view           A_view,
        oneapi::mkl::sparse::matrix_handle_t       A_handle,
        oneapi::mkl::sparse::dense_matrix_handle_t B_handle,
        const void*                                beta,
        oneapi::mkl::sparse::dense_matrix_handle_t C_handle,
        oneapi::mkl::sparse::spmm_alg              alg,
        oneapi::mkl::sparse::spmm_descr_t          spmm_descr,
        void*                                      workspace,
        const std::vector<sycl::event>             &dependencies = {});

    sycl::event spmm(
        sycl::queue                                &queue,
        oneapi::mkl::transpose                     opA,
```

(continues on next page)

(continued from previous page)

```
        oneapi::mkl::transpose                      opB,
        const void*                                 alpha,
        oneapi::mkl::sparse::matrix_view            A_view,
        oneapi::mkl::sparse::matrix_handle_t        A_handle,
        oneapi::mkl::sparse::dense_matrix_handle_t B_handle,
        const void*                                 beta,
        oneapi::mkl::sparse::dense_matrix_handle_t C_handle,
        oneapi::mkl::sparse::spmm_alg               alg,
        oneapi::mkl::sparse::spmm_descr_t           spmm_descr,
        const std::vector<sycl::event>              &dependencies = {});

}
```

Notes

- `spmm_buffer_size` and `spmm_optimize` must be called at least once before `spmm` with the same arguments. `spmm` can then be called multiple times. Calling `spmm_optimize` on the same descriptor can reset some of the descriptor's data such as the `workspace`.
- The data of the dense handles `B_handle` and `C_handle` and the scalars `alpha` and `beta` can be reset before each call to `spmm`. Changing the data of the sparse handle `A_handle` is undefined behavior.
- `spmm_optimize` and `spmm` are asynchronous.
- The algorithm defaults to `spmm_alg::default_alg` if a backend does not support the provided algorithm.
- The container type of all the handles and `workspace` must be consistent and use either USM pointers or SYCL buffers.

Input Parameters

queue
: The SYCL command queue which will be used for SYCL kernels execution.

opA
: Specifies operation `op()` on the input matrix A. The possible options are described in *transpose* enum class.

opB
: Specifies operation `op()` on the input matrix B. The possible options are described in *transpose* enum class.

alpha
: Host or USM pointer representing α. The USM allocation can be on the host or device. Must be a host pointer if SYCL buffers are used. Must be of the same type than the handles' data type.

A_view
: Specifies which part of the handle should be read as described by *matrix_view*.

A_handle
: Sparse matrix handle object representing A.

B_handle
: Dense matrix handle object representing B.

beta
: Host or USM pointer representing β. The USM allocation can be on the host or device. Must be a host pointer if SYCL buffers are used. Must be of the same type than the handles' data type.

C_handle
Dense matrix handle object representing C.

alg
Specifies the *spmm algorithm* to use.

spmm_descr
Initialized *spmm descriptor*.

temp_buffer_size
Output buffer size in bytes.

workspace

Workspace buffer or USM pointer, must be at least of size `temp_buffer_size` bytes and the address aligned on the size of the handles' data type.

If it is a buffer, its lifetime is extended until the *spmm descriptor* is released or the workspace is reset by `spmm_optimize`. The workspace cannot be a sub-buffer.

If it is a USM pointer, it must not be free'd until the corresponding `spmm` has completed. The data must be accessible on the device.

dependencies
List of events to depend on before starting asynchronous tasks that access data on the device. Ignored if buffers are used. Defaults to no dependencies.

Output Parameters

temp_buffer_size
Output buffer size in bytes. A temporary workspace of at least this size must be allocated to perform the specified spmm.

C_handle
Dense matrix handle object representing C, result of the `spmm` operation.

Return Values

Output event that can be waited upon or added as a dependency for the completion of the function. May be an empty event if buffers are used.

Throws

These functions shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::computation_error
oneapi::mkl::device_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

Parent topic: *Sparse BLAS*

spmv

Computes a sparse matrix by dense vector product.

Description and Assumptions

The `oneapi::mkl::sparse::spmv` routine computes a sparse matrix by dense vector product defined as:

$$y \leftarrow \alpha \cdot \mathrm{op}(A) \cdot x + \beta \cdot y$$

where:

α and β are scalars,
x is a dense vector of size `n`,
y is a dense vector of size `m`,
$\mathrm{op}(A)$ is a transformed sparse matrix of size `m`-by-`n`,
$\mathrm{op}()$ is the transform operation using the following description:

$$\mathrm{op}(A) = \begin{cases} A, & \text{oneapi::mkl::transpose::nontrans} \\ A^{\mathsf{T}}, & \text{oneapi::mkl::transpose::trans} \\ A^{\mathsf{H}}, & \text{oneapi::mkl::transpose::conjtrans} \end{cases}$$

spmv_descr

Definition

```
namespace oneapi::mkl::sparse {

    struct spmv_descr;
    using spmv_descr_t = spmv_descr*;

}
```

Description

Defines `spmv_descr_t` as an opaque pointer to the incomplete type `spmv_descr`. Each backend may provide a different implementation of the type `spmv_descr`. The `spmv_descr_t` object persists through the various stages of the spmv operation to house relevant state, optimizations and workspaces.

init_spmv_descr

Syntax

```
namespace oneapi::mkl::sparse {

    void init_spmv_descr (sycl::queue                          &queue,
                          oneapi::mkl::sparse::spmv_descr_t *p_spmv_descr);

}
```

Input parameters

queue
The SYCL command queue which will be used for SYCL kernels execution.

p_spmv_descr
The address of the `p_spmv_descr` object to be initialized. Must only be called on an uninitialized `spmv_descr_t` object.

Output parameters

p_spmv_descr
On return, the address is updated to point to a newly allocated and initialized `spmv_descr_t` object that can be used to perform spmv.

Throws

This function shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::unsupported_device

release_spmv_descr

Syntax

```
namespace oneapi::mkl::sparse {

    sycl::event release_spmv_descr (sycl::queue                        &queue,
                                    oneapi::mkl::sparse::spmv_descr_t spmv_descr,
                                    const std::vector<sycl::event>    &dependencies = {}
↪);

}
```

Input parameters

queue
The SYCL command queue which will be used for SYCL kernels execution.

spmv_descr
Descriptor initialized with `init_spmv_descr`.

dependencies
List of events to depend on before starting asynchronous tasks that access data on the device. Defaults to no dependencies.

Return Values

Output event that can be waited upon or added as a dependency for the completion of the function.

Throws

This function shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::unsupported_device

spmv_alg

Syntax

```
namespace oneapi::mkl::sparse {

    enum class spmv_alg {
        default_alg,
        no_optimize_alg,
        coo_alg1,
        coo_alg2,
        csr_alg1,
        csr_alg2,
        csr_alg3,
    };

}
```

Description

These algorithm enums are provided in case backends would like to implement various different algorithms for the operation. Behavior of the algorithms (e.g., bitwise reproducibility, atomics usage) and the preconditions to using specific algorithms (e.g. sortedness of matrix arrays) is implementation-defined and must be documented in the library implementing the oneAPI specification.

spmv

Syntax

```
namespace oneapi::mkl::sparse {

    void spmv_buffer_size(
        sycl::queue                                 &queue,
        oneapi::mkl::transpose                      opA,
```

(continues on next page)

(continued from previous page)

```
        const void*                                 alpha,
        oneapi::mkl::sparse::matrix_view            A_view,
        oneapi::mkl::sparse::matrix_handle_t        A_handle,
        oneapi::mkl::sparse::dense_vector_handle_t x_handle,
        const void*                                 beta,
        oneapi::mkl::sparse::dense_vector_handle_t y_handle,
        oneapi::mkl::sparse::spmv_alg               alg,
        oneapi::mkl::sparse::spmv_descr_t           spmv_descr,
        std::size_t                                 &temp_buffer_size);

    void spmv_optimize(
        sycl::queue                                 &queue,
        oneapi::mkl::transpose                      opA,
        const void*                                 alpha,
        oneapi::mkl::sparse::matrix_view            A_view,
        oneapi::mkl::sparse::matrix_handle_t        A_handle,
        oneapi::mkl::sparse::dense_vector_handle_t x_handle,
        const void*                                 beta,
        oneapi::mkl::sparse::dense_vector_handle_t y_handle,
        oneapi::mkl::sparse::spmv_alg               alg,
        oneapi::mkl::sparse::spmv_descr_t           spmv_descr,
        sycl::buffer<std::uint8_t, 1>               workspace);

    sycl::event spmv_optimize(
        sycl::queue                                 &queue,
        oneapi::mkl::transpose                      opA,
        const void*                                 alpha,
        oneapi::mkl::sparse::matrix_view            A_view,
        oneapi::mkl::sparse::matrix_handle_t        A_handle,
        oneapi::mkl::sparse::dense_vector_handle_t x_handle,
        const void*                                 beta,
        oneapi::mkl::sparse::dense_vector_handle_t y_handle,
        oneapi::mkl::sparse::spmv_alg               alg,
        oneapi::mkl::sparse::spmv_descr_t           spmv_descr,
        void*                                       workspace,
        const std::vector<sycl::event>              &dependencies = {});

    sycl::event spmv(
        sycl::queue                                 &queue,
        oneapi::mkl::transpose                      opA,
        const void*                                 alpha,
        oneapi::mkl::sparse::matrix_view            A_view,
        oneapi::mkl::sparse::matrix_handle_t        A_handle,
        oneapi::mkl::sparse::dense_vector_handle_t x_handle,
        const void*                                 beta,
        oneapi::mkl::sparse::dense_vector_handle_t y_handle,
        oneapi::mkl::sparse::spmv_alg               alg,
        oneapi::mkl::sparse::spmv_descr_t           spmv_descr,
        const std::vector<sycl::event>              &dependencies = {});

}
```

Notes

- `spmv_buffer_size` and `spmv_optimize` must be called at least once before `spmv` with the same arguments. `spmv` can then be called multiple times. Calling `spmv_optimize` on the same descriptor can reset some of the descriptor's data such as the `workspace`.
- The data of the dense handles `x_handle` and `y_handle` and the scalars `alpha` and `beta` can be reset before each call to `spmv`. Changing the data of the sparse handle `A_handle` is undefined behavior.
- `spmv_optimize` and `spmv` are asynchronous.
- The algorithm defaults to `spmv_alg::default_alg` if a backend does not support the provided algorithm.
- The container type of all the handles and `workspace` must be consistent and use either USM pointers or SYCL buffers.

Input Parameters

queue
The SYCL command queue which will be used for SYCL kernels execution.

opA
Specifies operation `op()` on the input matrix. The possible options are described in *transpose* enum class.

alpha
Host or USM pointer representing α. The USM allocation can be on the host or device. Must be a host pointer if SYCL buffers are used. Must be of the same type than the handles' data type.

A_view
Specifies which part of the handle should be read as described by *matrix_view*.

A_handle
Sparse matrix handle object representing A.

x_handle
Dense vector handle object representing x.

beta
Host or USM pointer representing β. The USM allocation can be on the host or device. Must be a host pointer if SYCL buffers are used. Must be of the same type than the handles' data type.

y_handle
Dense vector handle object representing y.

alg
Specifies the *spmv algorithm* to use.

spmv_descr
Initialized *spmv descriptor*.

temp_buffer_size
Output buffer size in bytes.

workspace

Workspace buffer or USM pointer, must be at least of size `temp_buffer_size` bytes and the address aligned on the size of the handles' data type.

If it is a buffer, its lifetime is extended until the *spmv descriptor* is released or the workspace is reset by `spmv_optimize`. The workspace cannot be a sub-buffer.

If it is a USM pointer, it must not be free'd until the corresponding `spmv` has completed. The data must be accessible on the device.

dependencies
List of events to depend on before starting asynchronous tasks that access data on the device. Ignored if buffers are used. Defaults to no dependencies.

Output Parameters

temp_buffer_size
Output buffer size in bytes. A temporary workspace of at least this size must be allocated to perform the specified spmv.

y_handle
Dense vector handle object representing y, result of the `spmv` operation.

Return Values

Output event that can be waited upon or added as a dependency for the completion of the function. May be an empty event if buffers are used.

Throws

These functions shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::computation_error
oneapi::mkl::device_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

Parent topic: *Sparse BLAS*

spsv

Solves a system of linear equations where the coefficients are described by a triangular sparse matrix.

Description and Assumptions

The `oneapi::mkl::sparse::spsv` routine solves a system of linear equations for a square matrix:

$$\text{op}(A) \cdot y \leftarrow \alpha \cdot x$$

where:
α is a scalar,
x and y are dense vectors of size `m`,
$\text{op}(A)$ is a transformed sparse matrix of size `m`-by-`m`,
$\text{op}()$ is the transform operation using the following description:

$$\mathrm{op}(A) = \begin{cases} A, & \text{oneapi::mkl::transpose::nontrans} \\ A^{\mathsf{T}}, & \text{oneapi::mkl::transpose::trans} \\ A^{\mathsf{H}}, & \text{oneapi::mkl::transpose::conjtrans} \end{cases}$$

spsv_descr

Definition

```
namespace oneapi::mkl::sparse {

    struct spsv_descr;
    using spsv_descr_t = spsv_descr*;

}
```

Description

Defines `spsv_descr_t` as an opaque pointer to the incomplete type `spsv_descr`. Each backend may provide a different implementation of the type `spsv_descr`. The *spsv_descr_t* object persists through the various stages of the spsv operation to house relevant state, optimizations and workspaces.

init_spsv_descr

Syntax

```
namespace oneapi::mkl::sparse {

    void init_spsv_descr (sycl::queue                          &queue,
                          oneapi::mkl::sparse::spsv_descr_t *p_spsv_descr);

}
```

Input parameters

queue
: The SYCL command queue which will be used for SYCL kernels execution.

p_spsv_descr
: The address of the `p_spsv_descr` object to be initialized. Must only be called on an uninitialized `spsv_descr_t` object.

Output parameters

p_spsv_descr
On return, the address is updated to point to a newly allocated and initialized `spsv_descr_t` object that can be used to perform spsv.

Throws

This function shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::unsupported_device

release_spsv_descr

Syntax

```
namespace oneapi::mkl::sparse {

    sycl::event release_spsv_descr (sycl::queue                        &queue,
                                    oneapi::mkl::sparse::spsv_descr_t spsv_descr,
                                    const std::vector<sycl::event>    &dependencies = {}
↪);

}
```

Input parameters

queue
The SYCL command queue which will be used for SYCL kernels execution.

spsv_descr
Descriptor initialized with `init_spsv_descr`.

dependencies
List of events to depend on before starting asynchronous tasks that access data on the device. Defaults to no dependencies.

Return Values

Output event that can be waited upon or added as a dependency for the completion of the function.

Throws

This function shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::unsupported_device

spsv_alg

Syntax

```
namespace oneapi::mkl::sparse {

    enum class spsv_alg {
        default_alg,
        no_optimize_alg,
    };

}
```

Description

These algorithm enums are provided in case backends would like to implement various different algorithms for the operation. Behavior of the algorithms (e.g., bitwise reproducibility, atomics usage) and the preconditions to using specific algorithms (e.g. sortedness of matrix arrays) is implementation-defined and must be documented in the library implementing the oneAPI specification.

spsv

Syntax

```
namespace oneapi::mkl::sparse {

    void spsv_buffer_size(
        sycl::queue                                &queue,
        oneapi::mkl::transpose                     opA,
        const void*                                alpha,
        oneapi::mkl::sparse::matrix_view           A_view,
        oneapi::mkl::sparse::matrix_handle_t       A_handle,
        oneapi::mkl::sparse::dense_vector_handle_t x_handle,
        oneapi::mkl::sparse::dense_vector_handle_t y_handle,
```

(continues on next page)

(continued from previous page)

```
        oneapi::mkl::sparse::spsv_alg               alg,
        oneapi::mkl::sparse::spsv_descr_t           spsv_descr,
        std::size_t                                 &temp_buffer_size);

    void spsv_optimize(
        sycl::queue                                 &queue,
        oneapi::mkl::transpose                      opA,
        const void*                                 alpha,
        oneapi::mkl::sparse::matrix_view            A_view,
        oneapi::mkl::sparse::matrix_handle_t        A_handle,
        oneapi::mkl::sparse::dense_vector_handle_t x_handle,
        oneapi::mkl::sparse::dense_vector_handle_t y_handle,
        oneapi::mkl::sparse::spsv_alg               alg,
        oneapi::mkl::sparse::spsv_descr_t           spsv_descr,
        sycl::buffer<std::uint8_t, 1>               workspace);

    sycl::event spsv_optimize(
        sycl::queue                                 &queue,
        oneapi::mkl::transpose                      opA,
        const void*                                 alpha,
        oneapi::mkl::sparse::matrix_view            A_view,
        oneapi::mkl::sparse::matrix_handle_t        A_handle,
        oneapi::mkl::sparse::dense_vector_handle_t x_handle,
        oneapi::mkl::sparse::dense_vector_handle_t y_handle,
        oneapi::mkl::sparse::spsv_alg               alg,
        oneapi::mkl::sparse::spsv_descr_t           spsv_descr,
        void*                                       workspace,
        const std::vector<sycl::event>              &dependencies = {});

    sycl::event spsv(
        sycl::queue                                 &queue,
        oneapi::mkl::transpose                      opA,
        const void*                                 alpha,
        oneapi::mkl::sparse::matrix_view            A_view,
        oneapi::mkl::sparse::matrix_handle_t        A_handle,
        oneapi::mkl::sparse::dense_vector_handle_t x_handle,
        oneapi::mkl::sparse::dense_vector_handle_t y_handle,
        oneapi::mkl::sparse::spsv_alg               alg,
        oneapi::mkl::sparse::spsv_descr_t           spsv_descr,
        const std::vector<sycl::event>              &dependencies = {});

}
```

Notes

- `spsv_buffer_size` and `spsv_optimize` must be called at least once before `spsv` with the same arguments. `spsv` can then be called multiple times. Calling `spsv_optimize` on the same descriptor can reset some of the descriptor's data such as the `workspace`.
- The data of the dense handle `x_handle` and scalar `alpha` can be reset before each call to `spsv`. Changing the data of the sparse handle `A_handle` is undefined behavior.
- `spsv_optimize` and `spsv` are asynchronous.
- The algorithm defaults to `spsv_alg::default_alg` if a backend does not support the provided algorithm.
- The container type of all the handles and `workspace` must be consistent and use either USM pointers or SYCL buffers.

Input Parameters

queue
The SYCL command queue which will be used for SYCL kernels execution.

opA
Specifies operation `op()` on the input matrix. The possible options are described in *transpose* enum class.

alpha
Host or USM pointer representing α. The USM allocation can be on the host or device. Must be a host pointer if SYCL buffers are used. Must be of the same type than the handles' data type.

A_view
Specifies which part of the handle should be read as described by *matrix_view*. `A_view.type_view` must be `matrix_descr::triangular` or `matrix_descr::diagonal`.

A_handle
Sparse matrix handle object representing A.

x_handle
Dense vector handle object representing x.

y_handle
Dense vector handle object representing y.

alg
Specifies the *spsv algorithm* to use.

spsv_descr
Initialized *spsv descriptor*.

temp_buffer_size
Output buffer size in bytes.

workspace

Workspace buffer or USM pointer, must be at least of size `temp_buffer_size` bytes and the address aligned on the size of the handles' data type.

If it is a buffer, its lifetime is extended until the *spsv descriptor* is released or the workspace is reset by `spsv_optimize`. The workspace cannot be a sub-buffer.

If it is a USM pointer, it must not be free'd until the corresponding `spsv` has completed. The data must be accessible on the device.

dependencies
List of events to depend on before starting asynchronous tasks that access data on the device. Ignored if buffers are used. Defaults to no dependencies.

Output Parameters

temp_buffer_size
Output buffer size in bytes. A temporary workspace of at least this size must be allocated to perform the specified spsv.

y_handle
Dense vector handle object representing y, result of the `spsv` operation.

Return Values

Output event that can be waited upon or added as a dependency for the completion of the function. May be an empty event if buffers are used.

Throws

These functions shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::computation_error
oneapi::mkl::device_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

Parent topic: *Sparse BLAS*

Matrix view

matrix_descr

Definition

```
namespace oneapi::mkl::sparse {

    enum class matrix_descr {
        general,
        symmetric,
        hermitian,
        triangular,
        diagonal,
    };
```

(continues on next page)

(continued from previous page)

```
}
```

Description

The matrix descriptor describes how an operation should interpret the data.

Value	Description
`general`	General case, use complete data.
`symmetric`	View as symmetric, use given triangular part.
`hermitian`	View as hermitian, use given triangular part.
`triangular`	View as triangular, use given triangular part.
`diagonal`	View as diagonal, use only main diagonal values.

matrix_view

Definition

```
namespace oneapi::mkl::sparse {

    struct matrix_view {
        matrix_descr type_view = matrix_descr::general;
        uplo uplo_view = uplo::lower;
        diag diag_view = diag::nonunit;

        matrix_view() = default;

        matrix_view(matrix_descr type_view);
    };

}
```

Description

The matrix view holds information to specify which part of the matrix should be read without changing the matrix's data.

See *matrix_descr*, *uplo* and *diag* for a description of the members.

The `uplo_view` member is ignored if `type_view` is `general` or `diagonal`.

The `diag_view` member is ignored if `type_view` is `general`.

Syntax

```
namespace oneapi::mkl::sparse {

    matrix_view::matrix_view () = default;

}
```

Default Constructor

Initializes the `matrix_view` with the default values as shown in the class definition.

Syntax

```
namespace oneapi::mkl::sparse {

    matrix_view::matrix_view(matrix_descr type_view);

}
```

Constructor from a matrix_descr

Initializes the `matrix_view` with the provided `matrix_descr`. By default the other members are initialized to the same value as the default constructor.

If the `matrix_desc` is `diagonal`, `diag_view` is initialized to `diag::unit`.

Parent topic: *Sparse BLAS*

Supported template types

Data Types <dataType>	Integer Types <indexType>
float	std::int32_t
double	std::int64_t
std::complex<float>	
std::complex<double>	

`dataType` is used to describe the precision (i.e. number of bits) and domain (i.e. real or complex) of the *data handles* and the operations using them.

`indexType` is used to describe the range of integer types such as indices, offsets or sizes of the *data handles* and the operations using them.

Parent topic: *Sparse BLAS*

Parent topic: *Sparse Linear Algebra*

9.2.3 Discrete Fourier Transforms

The *Discrete Fourier Transform Functions* offer several options for computing Discrete Fourier Transforms (DFTs).

Discrete Fourier Transform Functions

oneMKL provides a DPC++ interface to d-dimensional ($d \in \mathbb{Z}_{>0}$) Discrete Fourier Transforms (DFTs).

Definitions

Let $w^m_{k_1,k_2,\ldots,k_d}$ be the entry of multi-index $(k_1, k_2, \ldots, k_d) \in \mathbb{Z}^d$ in the m-th sequence of a set w of M d-dimensional periodic discrete sequences of period(s) (or "length(s)") $n_1 \times n_2 \times \cdots \times n_d$ ($M \in \mathbb{Z}_{>0}$, $m \in \{0, 1, \ldots, M-1\}$ and $n_\ell \in \mathbb{Z}_{>0}, \forall \ell \in \{1, \ldots, d\}$).

For every $m \in \{0, 1, \ldots, M-1\}$, the DFT of sequence w^m is the d-dimensional $n_1 \times n_2 \times \cdots \times n_d$ periodic discrete sequence z^m whose entries are defined as

$$z^m_{k_1,k_2,\ldots,k_d} = \sigma \sum_{j_d=0}^{n_d-1} \cdots \sum_{j_2=0}^{n_2-1} \sum_{j_1=0}^{n_1-1} w^m_{j_1,j_2,\ldots,j_d} \exp\left[\delta 2\pi\imath \left(\sum_{\ell=1}^{d} \frac{j_\ell k_\ell}{n_\ell}\right)\right] \; \forall\, (k_1, \ldots, k_d) \in \mathbb{Z}^d \tag{9.1}$$

where $\imath^2 = -1$ and σ is a scale factor. In (9.1), δ determines one of the two "directions" of the DFT: $\delta = -1$ defines the "forward DFT" while $\delta = +1$ defines the "backward DFT".

The domain of input (resp. output) discrete sequences for a forward (resp. backward) DFT is referred to as "forward domain". Conversely, the domain of output (resp. input) discrete sequences for forward (resp. backward) DFT is referred to as "backward domain".

oneMKL supports single-precision (fp32) and double-precision (fp64) floating-point arithmetic for the calculation of DFTs, using two types of forward domains:

- the set of complex d-dimensional periodic sequences, referred to as "complex forward domain";
- the set of real d-dimensional periodic sequences, referred to as "real forward domain".

Similarly, we refer to DFTs of complex (resp. real) forward domain as "complex DFTs" (resp. "real DFTs"). Regardless of the type of forward domain, the backward domain's data sequences are always complex.

The calculation of the same DFT for several, *i.e.*, $M > 1$, data sets of the same type of forward domain, using the same precision is referred to as a "batched DFT".

Finite range of indices

In general, given the periodicity of the discrete data considered in any DFT, ranges of indices $(k_1, \ldots, k_d) \in \mathbb{Z}^d$ such that $0 \leq k_\ell < n_\ell, \forall \ell \in \{1, \ldots, d\}$ suffice to determine any relevant d-dimensional sequence unambiguously (for any valid m). In case of real DFTs, the data sequences in backward domain can be fully determined from a smaller range of indices. Indeed, if all entries of w are real in (9.1), then the entries of z are complex and, for any valid m, $\left(z^m_{k_1,k_2,\ldots,k_d}\right)^* = z^m_{n_1-k_1,n_2-k_2,\ldots,n_d-k_d} \; \forall\, (k_1, k_2, \ldots, k_d) \in \mathbb{Z}^d$ where λ^* represents the conjugate of complex number λ. This conjugate symmetry relation makes roughly half the data redundant in backward domain: in case of real DFTs, the data sequences in backward domain can be fully determined even if one of the d indices k_ℓ is limited to the range $0 \leq k_\ell \leq \lfloor \frac{n_\ell}{2} \rfloor$. In oneMKL, the index k_d, *i.e.*, the last dimension's index, is restricted as such for capturing an elementary set of non-redundant entries of data sequences belonging to the backward domain of real DFTs.

Elementary range of indices

In other words, oneMKL expects and produces a set of M d-dimensional *finite* data sequences $(\cdot)^{m}_{k_1,k_2,\ldots,k_d}$ with integer indices m and k_ℓ $(\ell \in \{1,\ldots,d\})$ in the elementary range

- $0 \leq m < M$;
- $0 \leq k_j < n_j,\ \forall j \in \{1,\ldots,d-1\}$, if $d > 1$;
- $0 \leq k_d < n_d$, except for backward domain's data sequences of real DFTs;
- $0 \leq k_d \leq \lfloor \frac{n_d}{2} \rfloor$, for backward domain's data sequences of real DFTs.

Additional constraints for data in backward domain of real DFTs

Finally, note that the conjugate symmetry relation further constrains some of the entries (or pairs thereof) in the backward domain's data sequences for real DFTs. Specifically, for any of the M sequences,

- the imaginary part must be 0 for any entry of multi-index $(k_1, k_2, \ldots, k_d)$ such that $k_\ell \equiv (n_\ell - k_\ell) \pmod{n_\ell}, \forall \ell \in \{1,\ldots,d\}$, *e.g.*, entry of multi-index $(0, 0, \ldots, 0)$;
- pairs of entries of multi-indices $(k_1, k_2, \ldots, k_d)$ and $(j_1, j_2, \ldots, j_d)$ such that $k_\ell \equiv (n_\ell - j_\ell) \pmod{n_\ell}, \forall \ell \in \{1,\ldots,d\}$ must be complex conjugates of one another, *e.g.*, entries of multi-indices $(1, 0, \ldots, 0)$ and $(n_1 - 1, 0, \ldots, 0)$ must be complex conjugates (note that this case falls back to the above constraint if $n_1 = 2$).

Note: The behavior of oneMKL is undefined for real backward DFT if the input data does not satisfy those constraints. oneMKL considers it the user's responsibility to guarantee that these constraints are satisfied by the input data for real backward DFTs.

Recommended usage

The desired (batched) DFT to be computed is entirely defined by an object `desc` of the *descriptor* class. The desired type of forward domain and precision are determined at `desc`'s construction time by the specialization values chosen for the self-explanatory template parameters `prec` (of type *precision*) and `dom` (of type *domain*), respectively. The transform size $n_1 \times n_2 \times \cdots \times n_d$ is also set at construction time as a required argument to the class constructor. Other configuration details for the (batched) DFT under consideration may be specified by invoking the *set_value* member function of `desc` for every relevant configuration setting (*e.g.*, the number M of sequences to consider in case of a batched DFT). Once configured as desired, the *commit* member function of `desc`, requiring a `sycl::queue` object `Q`, may be invoked. The successful completion of the latter makes `desc` committed to the desired (batched) DFT *as configured*, for the particular device and context encapsulated by `Q`. The *compute_forward* (resp. *compute_backward*) function may then be called and provided with `desc` to enqueue operations relevant to the desired forward (resp. backward) DFT calculations with user-provided, device-accessible data.

Note: Objects of the *descriptor* class

- must be successfully committed prior to providing them to any compute function;
- must be re-committed to account for any change in configuration after it was already successfully committed;
- deliver best performance for DFT calculations when created, configured and committed outside applications' hotpath(s) that use them multiple times for identically-configured (batched) DFTs. *compute_forward* and/or *compute_backward* should be the only oneMKL DFT-related routines invoked in programs' hotpaths.

Summary table

The table below summarizes the object and functions relevant to computing DFTs (all defined in the `oneapi::mkl::dft` namespace).

Routines and Objects	Description
descriptor	A class whose instances define a specific (batched) DFT(s) to be calculated.
descriptor::set_value	A member function of the *descriptor* class to set (writable) *configuration parameters* for an instance of that class.
descriptor::get_value	A member function of the *descriptor* class to query *configuration parameters* from any instance of that class.
descriptor::commit	A member function of the *descriptor* class to commit an instance of that class to the (batched) DFT calculations it defines, on a given queue.
compute_forward	A function requiring a successfully-committed object of the *descriptor* class to compute a forward (batched) DFT, as defined by that object.
compute_backward	A function requiring a successfully-committed object of the *descriptor* class to compute a backward (batched) DFT, as defined by that object.

Parent topic: *oneMKL Domains*

The `descriptor` class

Objects of the `descriptor` class define DFT(s) to be computed.

Description

Any desired (batched) DFT is to be fully determined by an object of the `oneapi::mkl::dft::descriptor` class, defined in the `oneapi::mkl::dft` namespace. The scoped enumeration types *precision*, *domain*, *config_param* and *config_value* defined in the same namespace (and the corresponding ranges of values) are relevant to the definition and configurations of objects of the `descriptor` class. The `descriptor` class allows the user to set several (resp. query all) configuration parameters for (resp. from) any of its instances by using their *set_value* (resp. *get_value*) member function.

Invoking the member function *commit* of an object of the `descriptor` class effectively commits that object to the desired DFT calculations, as configured and determined by that very object, on the specified device encapsulated by the `sycl::queue` object required by that function.

The desired forward (resp. backward) DFT calculations may then be computed by passing such a committed `descriptor` object to the *compute_forward* (resp. *compute_backward*) function (defined in the `oneapi::mkl::dft` namespace as well), along with the relevant data containers (`sycl::buffer` object(s) or pointer(s) to a device-accessible USM allocations) for the desired DFT(s). This function makes the `descriptor` object enqueue the operations relevant for the desired calculations to the `sycl::queue` object it was given when committing it.

Note: The *compute_forward* and *compute_backward* functions may need to be able to access the internals of the `descriptor` object to compute the desired transform(s), this could be done for instance, by labeling them as friend functions of the `descriptor` class.

Syntax

The `descriptor` class is defined in the `oneapi::mkl::dft` namespace.

```
namespace oneapi::mkl::dft {

   template <oneapi::mkl::dft::precision prec, oneapi::mkl::dft::domain dom>
   class descriptor {
    public:

       // Constructor for 1-dimensional DFT
       descriptor(std::int64_t length); // d = 1;

       // Constructor for d-dimensional DFT
       descriptor(std::vector<std::int64_t> lengths); // d = lengths.size();

       descriptor(const descriptor&);

       descriptor(descriptor&&);

       descriptor& operator=(const descriptor&);

       descriptor& operator=(descriptor&&);

       ~descriptor();


       void set_value(oneapi::mkl::dft::config_param param, ...);

       void get_value(oneapi::mkl::dft::config_param param, ...);

       void set_workspace(sycl::buffer<scalar_type, 1> &workspaceBuf);
       void set_workspace(scalar_type* workspaceUSM);

       void commit(sycl::queue &queue);

    };


}
```

Descriptor class template parameters

precision **prec**
: Specifies the floating-point precision in which the user-provided data is to be provided, the transform is to be carried out and the results are to be returned. The possible specialization values are `oneapi::mkl::dft::precision::SINGLE` and `oneapi::mkl::dft::precision::DOUBLE`. Objects of the `descriptor` class specialized with *precision* template parameter `prec` as value `oneapi::mkl::dft::precision::SINGLE` (resp. `oneapi::mkl::dft::precision::DOUBLE`) are referred to as "single-precision descriptors" (resp. "double-precision descriptors").

domain **dom**
: Specifies the forward domain of the transform. The possible specialization values are

`oneapi::mkl::dft::domain::COMPLEX` and `oneapi::mkl::dft::domain::REAL`. Objects of the `descriptor` class specialized with *domain* template parameter `dom` as value `oneapi::mkl::dft::precision::COMPLEX` (resp. `oneapi::mkl::dft::precision::REAL`) are referred to as "complex descriptors" (resp. "real descriptors").

Descriptor class member functions

Routines	Description
constructors	Creates and default-initializes a `descriptor` object for a d-dimensional DFT of user-defined length(s) $\{n_1, \ldots, n_d\}$.
assignment operators	Performs a deep copy of or moves the argument.
set_value	Sets a configuration value for a specific configuration parameter.
get_value	Queries the configuration value associated with a particular configuration parameter.
set_workspace	Sets the external workspace to use when `config_param::WORKSPACE_PLACEMENT` is set to `config_value::WORKSPACE_EXTERNAL`.
commit	Commits the `descriptor` object to enqueue the operations relevant to the (batched) DFT(s) it determines to a given, user-provided `sycl::queue` object; completes all initialization work relevant to and required by the chosen, device-compliant implementation for the particular DFT, as defined by the `descriptor` object.

Descriptor class constructors

The constructors for the `descriptor` object instantiate it with all the relevant default configuration settings (which may depend on the specialization values used for the *precision* template parameter `prec` and for the *domain* template parameter `dom`). The constructors do not perform any significant initialization work as changes in the object's configuration(s) may be operated thereafter (via its *set_value* member function) and modify significantly the nature of that work.

The copy constructor performs a deep copy of `descriptor` objects.

Syntax (one-dimensional transform)

```
namespace oneapi::mkl::dft {

   template <oneapi::mkl::dft::precision prec, oneapi::mkl::dft::domain dom>
   descriptor<prec,dom>(std::int64_t length);

}
```

Syntax (d-dimensional transform with $d > 0$)

```
namespace oneapi::mkl::dft {

   template <oneapi::mkl::dft::precision prec, oneapi::mkl::dft::domain dom>
   descriptor<prec,dom>(std::vector<std::int64_t> lengths);

}
```

Copy constructor

```
namespace oneapi::mkl::dft {

   template <oneapi::mkl::dft::precision prec, oneapi::mkl::dft::domain dom>
   descriptor<prec,dom>(const descriptor<prec,dom>& other);

}
```

Move constructor

```
namespace oneapi::mkl::dft {

   template <oneapi::mkl::dft::precision prec, oneapi::mkl::dft::domain dom>
   descriptor<prec,dom>(descriptor<prec,dom>&& other);

}
```

Input Parameters

length
: Length $n_1 > 0$ of the data sequence(s) for one-dimensional transform(s).

lengths
: Vector of $d > 0$ lengths $\{n_1, \ldots, n_d\}$ of the data sequence(s) for d-dimensional transform(s). The values are to be provided in that order and such that $n_j > 0$, $\forall j \in \{1, \ldots, d\}$.

other
: Another `descriptor` object of the same type to copy or move.

Throws

The `descriptor::descriptor()` constructors shall throw the following exception if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here:

oneapi::mkl::host_bad_alloc()
: If any memory allocations on host have failed, for instance due to insufficient memory.

oneapi::mkl::unimplemented()
: If the dimension d, *i.e.*, the size of vector `lengths`, is larger than what is supported by the library implementation.

Descriptor class member table: *Descriptor class member functions*

Descriptor class assignment operators

The copy assignment operator results in a deep copy.

Copy assignment

```
namespace oneapi::mkl::dft {

   template <oneapi::mkl::dft::precision prec, oneapi::mkl::dft::domain dom>
   descriptor<prec,dom>& descriptor<prec,dom>::operator=(const descriptor<prec,dom>&␣
↪other);

}
```

Move assignment

```
namespace oneapi::mkl::dft {

   template <oneapi::mkl::dft::precision prec, oneapi::mkl::dft::domain dom>
   descriptor<prec,dom>& descriptor<prec,dom>::operator=(descriptor<prec,dom>&& other);

}
```

Input Parameters

other
: The `descriptor` object to copy or move from.

Throws

The assignment operators shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here:

oneapi::mkl::host_bad_alloc()
: If any memory allocations on host have failed, for instance due to insufficient memory.

Descriptor class member table: *Descriptor class member functions*

set_value

The `set_value` member function of the `descriptor` class sets a configuration value corresponding to a (read-write) configuration parameter for the DFT(s) that a `descriptor` object defines. This function is to be used as many times as required for all the necessary configuration parameters to be set prior to committing the `descriptor` object (by calling its member function *commit*).

This function requires and expects exactly **two** arguments: it sets the configuration value (second argument) corresponding to the configuration parameter (first argument) `param` of type `oneapi::mkl::dft::config_param`. The type of the configuration value (second argument) to be set depends on the value of `param`: it can be `oneapi::mkl::dft::config_value` or a native type like `std::int64_t` or `float` (more details available *here*).

Syntax

```
namespace oneapi::mkl::dft {

   template <oneapi::mkl::dft::precision prec, oneapi::mkl::dft::domain dom>
   void descriptor<prec,dom>::set_value(oneapi::mkl::dft::config_param param, ...);

}
```

Input Parameters

param
: One of the possible values of type *config_param* representing the (writable) configuration parameter to be set.

...
: An element of the appropriate type for the configuration value corresponding to the targeted configuration parameter `param` (appropriate type defined *here*).

Throws

The `descriptor::set_value()` routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here:

oneapi::mkl::invalid_argument()
: If the provided *config_param* and/or configuration value is not valid.

oneapi::mkl::unimplemented()
: If the provided *config_param* and configuration value are valid, but not supported by the library implementation.

Descriptor class member table: *Descriptor class member functions*

get_value

The `get_value` member function of the `descriptor` class queries the configuration value corresponding to any configuration parameter for the DFT that a `descriptor` object defines.

This function requires and expects exactly **two** arguments: it returns the configuration value (into the element pointed by the second argument) corresponding to the queried configuration parameter (first argument) `param` of type `oneapi::mkl::dft::config_param`. The type of the second argument depends on the value of `param`: it is a pointer to a writable element of type `oneapi::mkl::dft::domain`, `oneapi::mkl::dft::precision`, `oneapi::mkl::dft::config_value` or a native type like `std::int64_t` or `float` (more details available *here*).

Note: The value returned by `get_value` corresponds to the latest value set for the corresponding configuration parameter being queried or the corresponding default value if that parameter was not set or if it is not writable, even if that value was set after the descriptor was committed.

Syntax

```
namespace oneapi::mkl::dft {

   template <oneapi::mkl::dft::precision prec, oneapi::mkl::dft::domain dom>
   void descriptor<prec,dom>::get_value(oneapi::mkl::dft::config_param param, ...);

}
```

Input Parameters

param
: One of the possible values of type *config_param* representing the configuration parameter being queried.

...
: A pointer to a writable element of the appropriate type for the configuration value corresponding to the queried configuration parameter `param` (appropriate type of pointed element defined *here*).

Throws

The `descriptor::get_value()` routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here:

oneapi::mkl::invalid_argument()
: If the requested *config_param* is not valid.

Descriptor class member table: *Descriptor class member functions*

set_workspace

Sets the workspace for when `config_param::WORKSPACE_PLACEMENT` is set to `config_value::WORKSPACE_EXTERNAL`.

Description

This function sets the workspace to use when computing DFTs for when an external workspace is set. This function may only be called after the descriptor has been committed. The size of the provided workspace must be equal to or larger than the required workspace size obtained by calling `descriptor<prec, dom>::get_value(config_param::WORKSPACE_EXTERNAL_BYTES, &workspaceBytes)`.

A descriptor where `config_value::WORKSPACE_EXTERNAL` is specified for `config_param::WORKSPACE_PLACEMENT` is not a valid descriptor for compute calls until this function has been successfully called.

The type of workspace must match the compute calls for which it is used. That is, if the workspace is provided as a `sycl::buffer`, the compute calls must also use `sycl::buffer` for their arguments. Likewise, a USM allocated workspace must only be used with USM compute calls. Failing to do this will result in an invalid descriptor for compute calls.

If the workspace is a USM allocation, the user must not use it for other purposes in parallel whilst the DFT `compute_forward` or `compute_backward` are in progress.

This function can be called on committed descriptors where the workspace placement is not `config_value::WORKSPACE_EXTERNAL`. The provided workspace may or may not be used in compute calls. However, the aforementioned restrictions will still apply.

Syntax (buffer workspace)

```
namespace oneapi::mkl::dft {

   template <oneapi::mkl::dft::precision prec, oneapi::mkl::dft::domain dom>
   void descriptor<prec,dom>::set_workspace(sycl::buffer<scalar_type, 1> &workspaceBuf);
}
```

Syntax (USM workspace)

```
namespace oneapi::mkl::dft {

   template <oneapi::mkl::dft::precision prec, oneapi::mkl::dft::domain dom>
   void descriptor<prec,dom>::set_workspace(scalar_type* workspaceUSM);

}
```

Input Parameters

workspaceBuf
: A workspace buffer where `scalar_type` is the floating-point type according to `prec`. This buffer must be sufficiently large or an exception will be thrown. A sub-buffer cannot be used.

workspaceUSM
: A workspace USM allocation where `scalar_type` is the floating-point type according to `prec`. This allocation must be accessible on the device on which the descriptor is committed. It is assumed that this USM allocation is sufficiently large. The pointer is expected to be aligned to `scalar_type`.

Throws

The `descriptor::set_workspace()` routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here:

oneapi::mkl::invalid_argument()
: If the provided buffer `workspaceBuf` is not sufficiently large or is a sub-buffer, or if the provided USM allocation `workspaceUSM` is `nullptr` when an external workspace of size greater than zero is required.

oneapi::mkl::uninitialized()
: If `set_workspace` is called before the descriptor is committed.

Descriptor class member table: *Descriptor class member functions*

commit

The `commit` member function commits a `descriptor` object to the DFT calculations it defines consistently with its configuration settings, by completing all the initialization work (*e.g.*, algorithm selection, algorithm tuning, choice of factorization, memory allocations, calculation of twiddle factors, etc.) required by the chosen implementation for the desired DFT(s) on the targeted device. Objects of the `descriptor` class **must** be committed prior to using them in any call to *compute_forward* or *compute_backward* (which trigger actual DFT calculations).

As specified *above*, all required configuration parameters must be set before this function is called. Any change in configuration operated on a `descriptor` object via a call to its *set_value* member function *after* it was committed results in an undefined state not suitable for computation until this `commit` member function is called again.

Syntax

```
namespace oneapi::mkl::dft {

   template <oneapi::mkl::dft::precision prec, oneapi::mkl::dft::domain dom>
   void descriptor<prec,dom>::commit(sycl::queue& queue);
}
```

Input Parameters

queue
: Valid `sycl::queue` object to which the operations relevant to the desired DFT(s) are to be enqueued.

Throws

The `descriptor::commit()` routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here (if the `descriptor` object's configuration was found to be inconsistent, for instance):

oneapi::mkl::invalid_argument()
: If the queue is found to be invalid in any way.

oneapi::mkl::host_bad_alloc()
: If any host side only memory allocations fail, for instance due to lack of memory.

oneapi::mkl::device_bad_alloc()
: If any device or shared memory allocation fail.

Descriptor class member table: *Descriptor class member functions*

Parent topic: *Discrete Fourier Transform Functions*

DFT-related scoped enumeration types

The following scoped enumeration types, defined in the `oneapi::mkl::dft` namespace, are used for constructing and configuring objects of the *descriptor* class consistently with the DFT(s) they are meant to define.

Scoped enumeration type	Description
precision	Represents the precision of the floating-point data format and of the floating-point arithmetic to be used for the desired DFT calculations. A template parameter `prec` of this type is used for the *descriptor* class.
domain	Represents the type of forward domain for the desired DFT(s). A template parameter `dom` of this type is used for the *descriptor* class.
config_param	Represents configuration parameters for objects of the *descriptor* class. The configuration values associated with the configuration parameters can be retrieved (resp. set, for writable parameters) via the object's *get_value* (resp. *set_value*) member function.
config_value	Represents the possible configuration values for some of the *configuration parameters* that may take only a few determined, non-numeric values.

precision

This scoped enumeration type represents the precision of the floating-point format to be used for the desired DFT(s). The same precision is to be used for the user-provided data, the computation being carried out by oneMKL and the results delivered by oneMKL.

Syntax

```
enum class precision {
   SINGLE,
   DOUBLE
};
```

Value	Description
SINGLE	Single-precision floating-point format (FP32) is used for data representation and arithmetic operations.
DOUBLE	Double-precision floating-point format (FP64) is used for data representation and arithmetic operations.

domain

This scoped enumeration type represents the type of forward domain for the desired DFTs (as explained in the *introduction*, the backward domain type is always complex).

Syntax

```
enum class domain {
   REAL,
   COMPLEX
};
```

Value	Description
REAL	The forward domain is the set of real d-dimensional periodic sequences.
COMPLEX	The forward domain is the set of complex d-dimensional periodic sequences.

config_param

This scoped enumeration type represents configuration parameters for objects of the *descriptor* class.

```
enum class config_param {
   // read-only parameters:
   FORWARD_DOMAIN,
   DIMENSION,
   LENGTHS,
   PRECISION,
   COMMIT_STATUS,
   // writable parameters:
   FORWARD_SCALE,
   BACKWARD_SCALE,

   NUMBER_OF_TRANSFORMS,

   COMPLEX_STORAGE,

   PLACEMENT,

   FWD_STRIDES,
   BWD_STRIDES,
   INPUT_STRIDES, // deprecated
   OUTPUT_STRIDES, // deprecated

   FWD_DISTANCE,
   BWD_DISTANCE,

   WORKSPACE_PLACEMENT,
   WORKSPACE_EXTERNAL_BYTES
};
```

Configuration parameters represented by `config_param::FORWARD_DOMAIN` and `config_param::PRECISION` are associated with configuration values of type *domain* and *precision* respectively. Other configuration parameters are associated with configuration values of type *config_value* or of a native type like `std::int64_t`, `std::vector<std::int64_t>`, `float` or `double`. This is further specified in the following table.

Value of config_p	Represented configuration parameter(s)	Type of associated configuration value [default value]
FORWARD_D	Type of forward domain, set at construction time as the specialization value of *domain* template parameter dom. This parameter is read-only.	*domain* [dom]
DIMENSION	Value of the dimension d of the desired DFTs, set at construction time. This parameter is read-only.	std::int64_t [d]
LENGTH	Values $\{n_1, \ldots, n_d\}$ of the periods (or "lengths") of the desired DFT, set at construction time. This parameter is read-only.	std::vector<std::int64_t> of size d or, if $d = 1$, std::int64_t [std::vector<int64_t>({n_1,..., n_d})]
PRECISION	Floating-point precision to be considered by and used for the DFT calculation(s), set at construction time as the specialization value of *precision* template parameter prec. This parameter is read-only.	*precision* [prec]
COMMIT_STA	Status flag indicating whether the object is ready for computations after a successful call to *commit*. This parameter is read-only.	*config_value* (possible values are self-explanatory config_value::COMMITTED or config_value::UNCOMMITTED). [config_value::UNCOMMITTED]
FORWARD_S	Value of σ for the forward DFT.	float (resp. double) for single-precision (resp. double-precision) descriptors [1.0]
BACKWARD_S	Value of σ for the backward DFT.	float (resp. double) for single-precision (resp. double-precision) descriptors [1.0]
*NUMBER_OF_*	Value of M. This is relevant (and *must* be set) for batched DFT(s), *i.e.*, if $M > 1$.	std::int64_t [1]
COMPLEX_ST	Data storage type used (relevant for complex descriptors only).	*config_value* (possible values are config_value::COMPLEX_COMPLEX or config_value::REAL_REAL) [config_value::COMPLEX_COMPLEX]
PLACE-		

config_value

This scoped enumeration type represents possible non-numeric configuration values associated with some *configuration parameters*.

```
enum class config_value {
   // for config_param::COMMIT_STATUS
   COMMITTED,
   UNCOMMITTED,

   // for config_param::COMPLEX_STORAGE,
   COMPLEX_COMPLEX,
   REAL_REAL,

   // for config_param::PLACEMENT
   INPLACE,
   NOT_INPLACE

   // For config_param::WORKSPACE_PLACEMENT
   WORKSPACE_AUTOMATIC,
   WORKSPACE_EXTERNAL,
};
```

Parent topic: *Discrete Fourier Transform Functions*

Configuration of Data Layouts

The DFT interface provides the configuration parameters `config_param::FWD_STRIDES` (resp. `config_param::BWD_STRIDES`) to define the data layout locating entries of relevant data sequences in the forward (resp. backward) domain. In case of batched transforms, *i.e.*, if $M > 1$ is configured by setting `config_param::NUMBER_OF_TRANSFORMS` accordingly, `config_param::FWD_DISTANCE` (resp. `config_param::BWD_DISTANCE`) completes the description of the data layout by specifying the distances between successive data sequences in the forward (resp. backward) domain.

Using the notations from the *introduction* and the superscript fwd (resp. bwd) for data sequences belonging to forward (resp. backward) domain, for any m and multi-index $(k_1, k_2, \ldots, k_d)$ within *valid range*, the corresponding entry $(\cdot)^m_{k_1,k_2,\ldots,k_d}$ - or the real or imaginary part thereof - of the relevant data sequence is located at index

$$s_0^{\mathrm{xwd}} + k_1\, s_1^{\mathrm{xwd}} + k_2\, s_2^{\mathrm{xwd}} + \cdots + k_d\, s_d^{\mathrm{xwd}} + m\, l^{\mathrm{xwd}} \tag{9.2}$$

of the corresponding data container (`sycl::buffer` object or device-accessible USM allocation) provided to the compute function, the base data type of which is (possibly implicitly re-interpreted) as documented in the *table* below. In the index expression (9.2), x = f (resp. x = b) for entries of forward-domain (resp. backward-domain) data sequences and

- s_j^{xwd}, $\forall j \in \{0, \ldots, d\}$ represents the *offset and generalized strides* defining the locations of entries within each d-dimensional data sequence in the forward (resp. backward) domain if x = f (resp. if x = b), counted in number of elements of the relevant *implicitly-assumed elementary data type*;
- l^{xwd} represents the *distance* between successive d-dimensional data sequences in the forward (resp. backward) domain if x = f (resp. if x = b), counted in number of elements of the relevant *implicitly-assumed elementary data type*.

Note: All data sequences (or respective real and imaginary parts thereof if separately stored) must belong to the same block allocation, as a consequence of the generalized index (9.2).

Implicitly-assumed elementary data type

When reading or writing an element at index (9.2) of any user-provided data container used at compute time, a *descriptor* object may re-interpret the base data type of that data container into an implicitly-assumed elementary data type. That implicitly-assumed data type depends on the object type, *i.e.*, on the specialization values used for the template parameters when instantiating the *descriptor* class, and, in case of complex descriptors, on the configuration value set for its configuration parameter `config_param::COMPLEX_STORAGE`. The table below lists the implicitly-assumed data type in either domain (last 2 columns) based on the object type and its configuration value for `config_param::COMPLEX_STORAGE` (first 2 columns).

Object type	Configuration value for configuration parameter `config_param::`
`descriptor<precision::SINGLE, domain::COMPLEX>`	`config_value::COMPLEX_COMPLEX`
`descriptor<precision::DOUBLE, domain::COMPLEX>`	`config_value::COMPLEX_COMPLEX`
`descriptor<precision::SINGLE, domain::COMPLEX>`	`config_value::REAL_REAL`
`descriptor<precision::DOUBLE, domain::COMPLEX>`	`config_value::REAL_REAL`
`descriptor<precision::SINGLE, domain::REAL>`	irrelevant
`descriptor<precision::DOUBLE, domain::REAL>`	irrelevant

Configuring data layouts for batched transforms

The value l^{xwd} in (9.2) above is communicated as an `std::int64_t` configuration value, set for the configuration parameter `config_param::FWD_DISTANCE` if x = f (resp. `config_param::BWD_DISTANCE` if x = b). This value is irrelevant for unbatched transforms, *i.e.*, for descriptors set to handle a number of transforms M equal to 1 (default behavior).

In case of batched transforms, the number $M > 1$ of desired DFTs *must* be set explicitly as an `std::int64_t` configuration value for the configuration parameter `config_param::NUMBER_OF_TRANSFORMS`. In that case, the configuration parameters `config_param::FWD_DISTANCE` and `config_param::BWD_DISTANCE` *must also* be set explicitly since their default configuration values of 0 would break the *consistency requirements* for any $M > 1$.

Configuring strides in forward and backward domains

The values $s_0^{\text{xwd}}, s_1^{\text{xwd}}, \ldots, s_d^{\text{xwd}}$ in (9.2) above are communicated as elements, in that order, of a $(d+1)$-long `std::vector<std::int64_t>` configuration value, set for the configuration parameter `config_param::FWD_STRIDES` if x = f (resp. `config_param::BWD_STRIDES` if x = b). The element s_0^{xwd} represents an absolute offset (or "displacement") in the data sets while the subsequent elements s_j^{xwd} $(j > 0)$ are generalized strides to be considered along dimensions $j \in \{1, \ldots, d\}$.

The default values set for the forward and backward strides correspond to the data layout configurations for unbatched, in-place transforms using unit stride along the last dimension with no offset (and minimal padding in forward domain in case of real descriptors, aligning with the *requirements for in-place transforms*). In other words, the default values are $s_0^{\text{fwd}} = s_0^{\text{bwd}} = 0$, $s_d^{\text{fwd}} = s_d^{\text{bwd}} = 1$ and, for d-dimensional DFTs with $d > 1$,

- $s_{d-1}^{\text{fwd}} = s_{d-1}^{\text{bwd}} = n_d$ for complex descriptors;
- $s_{d-1}^{\text{bwd}} = \lfloor \frac{n_d}{2} \rfloor + 1$, and $s_{d-1}^{\text{fwd}} = 2s_{d-1}^{\text{bwd}}$ for real descriptors;

- if $d > 2$, $s_k^{\text{xwd}} = n_{k+1} s_{k+1}^{\text{xwd}}$ for $k \in \{1, \ldots, d-2\}$ (for x = f and x = b).

General consistency requirements

In general, the distances and strides must be set so that every index value (9.2) corresponds to a *unique* entry of the data sequences under consideration. In other words, there must not be one index value as expressed in (9.2) that corresponds to two different $(d+1)$-tuples $(m, k_1, k_2, \ldots, k_d)$ that are both within the *elementary range of indices considered by oneMKL*.

Additionally, for in-place transforms (configuration value `config_value::INPLACE` associated with configuration parameter `config_param::PLACEMENT`), the smallest stride value must be associated with the same dimension in forward and backward domains and the data layouts must abide by following "*consistency requirement*": the memory address(es) of leading entry(ies) along the last dimension must be identical in forward and backward domains. Specifically, considering any $(d+1)$-tuple $(m, k_1, k_2, \ldots, k_{d-1}, 0)$ within *valid range*, the memory address of the element of corresponding index value (9.2) in forward domain (considering the *implicitly assumed type* in forward domain) must be identical to the memory address of the element of corresponding index value (9.2) in backward domain (considering the *implicitly assumed type* in backward domain). Equivalently,

- for complex descriptors, the offset, stride(s) (and distances, if relevant) must be equal in forward and backward domain;
- for real descriptors, offsets and strides must satisfy $s_j^{\text{fwd}} = 2 s_j^{\text{bwd}}\ \forall j \in \{0, \ldots, d-1\}$ (note that $0 \leq j < d$) and distances, if relevant, must satisfy $l^{\text{fwd}} = 2 l^{\text{bwd}}$. Note that this leads to some data padding being required in forward domain if unit strides are used along the last dimension in forward and backward domains.

Configuring strides for input and output data [deprecated, not recommended]

Instead of specifying strides by domain, one may choose to specify the strides for input and output data sequences. Let $s_j^{\text{x}},\ j \in \{0, 1, \ldots, d\}$ be the stride values for input (resp. output) data sequences if x = i (resp. x = o). Such $s_0^{\text{x}}, s_1^{\text{x}}, \ldots, s_d^{\text{x}}$ values may be communicated as elements, in that order, of a $(d+1)$-long `std::vector<std::int64_t>` configuration value, set for the (deprecated) configuration parameter `config_param::INPUT_STRIDES` if x = i (resp. `config_param::OUTPUT_STRIDES` if x = o).

The values of s_j^{i} and s_j^{o} are to be used and considered by oneMKL if and only if $s_j^{\text{fwd}} = s_j^{\text{bwd}} = 0, \forall j \in \{0, 1, \ldots, d\}$. (This will happen automatically if `config_param::INPUT_STRIDES` and `config_param::OUTPUT_STRIDES` are set and `config_param::FWD_STRIDES` and `config_param::BWD_STRIDES` are not. See note below.) In such a case, *descriptor* objects must consider the data layouts corresponding to the two compute directions separately. As detailed above, relevant data sequence entries are accessed as elements of data containers (`sycl::buffer` objects or device-accessible USM allocations) provided to the compute function, the base data type of which is (possibly implicitly re-interpreted) as documented in *this table*. If using input and output strides, for any m and multi-index $(k_1, k_2, \ldots, k_d)$ within *valid range*, the index to be used when accessing a data sequence entry - or part thereof - in forward domain is

$$s_0^{\text{x}} + k_1\, s_1^{\text{x}} + k_2\, s_2^{\text{x}} + \cdots + k_d\, s_d^{\text{x}} + m\, l^{\text{fwd}}$$

where x = i (resp. x = o) for forward (resp. backward) DFT(s). Similarly, the index to be used when accessing a data sequence entry - or part thereof - in backward domain is

$$s_0^{\text{x}} + k_1\, s_1^{\text{x}} + k_2\, s_2^{\text{x}} + \cdots + k_d\, s_d^{\text{x}} + m\, l^{\text{bwd}}$$

where x = o (resp. x = i) for forward (resp. backward) DFT(s).

As a consequence, configuring *descriptor* objects using these deprecated configuration parameters makes their configuration direction-dependent when different stride values are used in forward and backward domains. Since the intended compute direction is unknown to the *descriptor* object when *committing* it, every direction that results in a *consistent data layout* in forward and backward domains must be supported by successfully committed *descriptor* objects.

Note: For *descriptor* objects with strides configured via these deprecated configuration parameters, the *consistency requirements* may be satisfied for only one of the two compute directions, *i.e.*, for only one of the forward or backward DFT(s). Such a configuration should not cause an exception to be thrown by the descriptor's *commit* member function but the behavior of oneMKL is undefined if using that object for the compute direction that does not align with the *consistency requirements*.

Note: Setting either of `config_param::INPUT_STRIDES` or `config_param::OUTPUT_STRIDES` triggers any default or previously-set values for `config_param::FWD_STRIDES` and `config_param::BWD_STRIDES` to reset to `std::vector<std::int64_t>(d+1, 0)` values, and vice versa. This default behavior prevents mix-and-matching usage of either of `config_param::INPUT_STRIDES` or `config_param::OUTPUT_STRIDES` with either of `config_param::FWD_STRIDES` or `config_param::BWD_STRIDES`, which is **not** to be supported. If such a configuration is attempted, an exception is to be thrown at commit time due to invalid configuration, as the stride values that were implicitly reset surely invalidate the *consistency requirements* for any non-trivial DFT.

If specifying the data layout strides using these deprecated configuration parameters and if the strides differ in forward and backward domain, the descriptor *must* be re-configured and re-committed for computing the DFT in the reverse direction as shown below.

```
// ...
desc.set_value(config_param::INPUT_STRIDES,  fwd_domain_strides);
desc.set_value(config_param::OUTPUT_STRIDES, bwd_domain_strides);
desc.commit(queue);
compute_forward(desc, ...);
// ...
desc.set_value(config_param::INPUT_STRIDES,  bwd_domain_strides);
desc.set_value(config_param::OUTPUT_STRIDES, fwd_domain_strides);
desc.commit(queue);
compute_backward(desc, ...);
```

The `config_param::INPUT_STRIDES` and `config_param::OUTPUT_STRIDES` parameters are deprecated. A warning message "{IN,OUT}PUT_STRIDES are deprecated: please use {F,B}WD_STRIDES, instead." is to be reported to applications using these configuration parameters.

Parent topic *DFT-related scoped enumeration types*

Data storage

The data storage convention observed by a *descriptor* object depends on whether it is a real or complex descriptor and, in case of complex descriptors, on the configuration value associated with configuration parameter `config_param::COMPLEX_STORAGE`.

Complex descriptors

For a complex descriptor, the configuration parameter `config_param::COMPLEX_STORAGE` specifies how the entries of the complex data sequences it consumes and produces are stored. If that configuration parameter is associated with a configuration value `config_value::COMPLEX_COMPLEX` (default behavior), those entries are accessed and stored as `std::complex<float>` (resp. `std::complex<double>`) elements of a single data container (device-accessible USM allocation or `sycl::buffer` object) if the *descriptor* object is a single-precision (resp. double-precision) descriptor. If the configuration value `config_value::REAL_REAL` is used instead, the real and imaginary parts of those entries are accessed and stored as `float` (resp. `double`) elements of two separate, non-overlapping data containers (device-accessible USM allocations or `sycl::buffer` objects) if the *descriptor* object is a single-precision (resp. double-precision) descriptor.

These two behaviors are further specified and illustrated below.

`config_value::COMPLEX_COMPLEX` for `config_param::COMPLEX_STORAGE`

For complex descriptors with parameter `config_param::COMPLEX_STORAGE` set to `config_value::COMPLEX_COMPLEX`, each of forward- and backward-domain data sequences must belong to a single data container (device-accessible USM allocation or `sycl::buffer` object). Any relevant entry $(\cdot)^m_{k_1,k_2,\ldots,k_d}$ is accessed/stored from/in a data container provided at compute time at the index value expressed in eq. (9.2) (from *this page*) of that data container, whose elementary data type is (possibly implicitly re-interpreted as) `std::complex<float>` (resp. `std::complex<double>`) for single-precision (resp. double-precision) descriptors.

The same unique data container is to be used for forward- and backward-domain data sequences for in-place transforms (for *descriptor* objects with configuration value `config_value::INPLACE` for configuration parameter `config_param::PLACEMENT`). Two separate data containers sharing no common elements are to be used for out-of-place transforms (for *descriptor* objects with configuration value `config_value::NOT_INPLACE` for configuration parameter `config_param::PLACEMENT`).

The following snippet illustrates the usage of `config_value::COMPLEX_COMPLEX` for configuration parameter `config_param::COMPLEX_STORAGE`, in the context of in-place, single-precision (fp32) calculations of M three-dimensional $n_1 \times n_2 \times n_3$ complex transforms, using identical (default) strides and distances in forward and backward domains, with USM allocations.

```
namespace dft = oneapi::mkl::dft;
dft::descriptor<dft::precision::SINGLE, dft::domain::COMPLEX> desc({n1, n2, n3});
std::vector<std::int64_t> strides({0, n2*n3, n3, 1});
std::int64_t dist = n1*n2*n3;
std::complex<float> *Z = (std::complex<float> *) malloc_
↪device(2*sizeof(float)*n1*n2*n3*M, queue);
desc.set_value(dft::config_param::FWD_STRIDES, strides);
desc.set_value(dft::config_param::BWD_STRIDES, strides);
desc.set_value(dft::config_param::FWD_DISTANCE, dist);
desc.set_value(dft::config_param::BWD_DISTANCE, dist);
desc.set_value(dft::config_param::NUMBER_OF_TRANSFORMS, M);
desc.set_value(dft::config_param::COMPLEX_STORAGE, dft::config_value::COMPLEX_COMPLEX);
desc.commit(queue);

// initialize forward-domain data such that entry {m;k1,k2,k3}
//   = Z[ strides[0] + k1*strides[1] + k2*strides[2] + k3*strides[3] + m*dist ]
compute_forward(desc, Z); // complex-to-complex in-place DFT
// in backward domain: entry {m;k1,k2,k3}
//   = Z[ strides[0] + k1*strides[1] + k2*strides[2] + k3*strides[3] + m*dist ]
```

config_value::REAL_REAL for config_param::COMPLEX_STORAGE

For complex descriptors with parameter config_param::COMPLEX_STORAGE set to config_value::REAL_REAL, forward- and backward-domain data sequences are read/stored from/in two different, non-overlapping data containers (device-accessible USM allocations or sycl::buffer objects) encapsulating the real and imaginary parts of the relevant entries separately. The real and imaginary parts of any relevant complex entry $(\cdot)^{m}_{k_1,k_2,\ldots,k_d}$ are both stored at the index value expressed in eq. (9.2) (from *this page*) of their respective data containers, whose elementary data type is (possibly implicitly re-interpreted as) float (resp. double) for single-precision (resp. double-precision) descriptors.

The same two data containers are to be used for real and imaginary parts of forward- and backward-domain data sequences for in-place transforms (for *descriptor* objects with configuration value config_value::INPLACE for configuration parameter config_param::PLACEMENT). Four separate data containers sharing no common elements are to be used for out-of-place transforms (for *descriptor* objects with configuration value config_value::NOT_INPLACE for configuration parameter config_param::PLACEMENT).

The following snippet illustrates the usage of config_value::REAL_REAL set for configuration parameter config_param::COMPLEX_STORAGE, in the context of in-place, single-precision (fp32) calculation of M three-dimensional $n_1 \times n_2 \times n_3$ complex transforms, using identical (default) strides and distances in forward and backward domains, with USM allocations.

```
namespace dft = oneapi::mkl::dft;
dft::descriptor<dft::precision::SINGLE, dft::domain::COMPLEX> desc({n1, n2, n3});
std::vector<std::int64_t> strides({0, n2*n3, n3, 1});
std::int64_t dist = n1*n2*n3;
float *ZR = (float *) malloc_device(sizeof(float)*n1*n2*n3*M, queue); // data container␣
↪for real parts
float *ZI = (float *) malloc_device(sizeof(float)*n1*n2*n3*M, queue); // data container␣
↪for imaginary parts
desc.set_value(dft::config_param::FWD_STRIDES, strides);
desc.set_value(dft::config_param::BWD_STRIDES, strides);
desc.set_value(dft::config_param::FWD_DISTANCE, dist);
desc.set_value(dft::config_param::BWD_DISTANCE, dist);
desc.set_value(dft::config_param::NUMBER_OF_TRANSFORMS, M);
desc.set_value(dft::config_param::COMPLEX_STORAGE, dft::config_value::REAL_REAL);
desc.commit(queue);

// initialize forward-domain data such that the real part of entry {m;k1,k2,k3}
//   = ZR[ strides[0] + k1*strides[1] + k2*strides[2] + k3*strides[3] + m*dist ]
// and the imaginary part of entry {m;k1,k2,k3}
//   = ZI[ strides[0] + k1*strides[1] + k2*strides[2] + k3*strides[3] + m*dist ]
compute_forward<decltype(desc), float>(desc, ZR, ZI); // complex-to-complex in-place DFT
// in backward domain: the real part of entry {m;k1,k2,k3}
//   = ZR[ strides[0] + k1*strides[1] + k2*strides[2] + k3*strides[3] + m*dist ]
// and the imaginary part of entry {m;k1,k2,k3}
//   = ZI[ strides[0] + k1*strides[1] + k2*strides[2] + k3*strides[3] + m*dist ]
```

Real descriptors

Real descriptors observe only one type of data storage. Any relevant (real) entry $(\cdot)^m_{k_1,k_2,\ldots,k_d}$ of a data sequence in forward domain is accessed and stored as a `float` (resp. `double`) element of a single data container (device-accessible USM allocation or `sycl::buffer` object) if the *descriptor* object is a single-precision (resp. double-precision) descriptor. Any relevant (complex) entry $(\cdot)^m_{k_1,k_2,\ldots,k_d}$ of a data sequence in backward domain is accessed and stored as a `std::complex<float>` (resp. `std::complex<double>`) element of a single data container (device-accessible USM allocation or `sycl::buffer` object) if the *descriptor* object is a single-precision (resp. double-precision) descriptor.

The following snippet illustrates the usage of a real, single-precision descriptor (and the corresponding data storage) for the in-place, single-precision (fp32), calculation of M three-dimensional $n_1 \times n_2 \times n_3$ real transforms, using default strides in forward and backward domains, with USM allocations.

```
namespace dft = oneapi::mkl::dft;
dft::descriptor<dft::precision::SINGLE, dft::domain::REAL> desc({n1, n2, n3});
// Note: integer divisions here below
std::vector<std::int64_t> fwd_strides({0, 2*n2*(n3/2 + 1), 2*(n3/2 + 1), 1});
std::vector<std::int64_t> bwd_strides({0,   n2*(n3/2 + 1),   (n3/2 + 1), 1});
std::int64_t fwd_dist = 2*n1*n2*(n3/2 + 1);
std::int64_t bwd_dist =   n1*n2*(n3/2 + 1);
float *data = (float *) malloc_device(sizeof(float)*fwd_dist*M, queue); // data container
desc.set_value(dft::config_param::FWD_STRIDES, fwd_strides);
desc.set_value(dft::config_param::BWD_STRIDES, bwd_strides);
desc.set_value(dft::config_param::FWD_DISTANCE, fwd_dist);
desc.set_value(dft::config_param::BWD_DISTANCE, bwd_dist);
desc.set_value(dft::config_param::NUMBER_OF_TRANSFORMS, M);
desc.commit(queue);

// initialize forward-domain data such that real entry {m;k1,k2,k3}
//   = data[ fwd_strides[0] + k1*fwd_strides[1] + k2*fwd_strides[2] + k3*fwd_strides[3]␣
↪+ m*fwd_dist ]
compute_forward(desc, data); // real-to-complex in-place DFT
// in backward domain, the implicitly-assumed type is complex so, considering
//   std::complex<float>* complex_data = static_cast<std::complex<float>*>(data);
//   we have entry {m;k1,k2,k3}
//   = complex_data[ bwd_strides[0] + k1*bwd_strides[1] + k2*bwd_strides[2] + k3*bwd_
↪strides[3] + m*bwd_dist ]
//   for 0 <= k3 <= n3/2.
//   Note: if n3/2 < k3 < n3, entry {m;k1,k2,k3} = std::conj(entry {m;n1-k1,n2-k2,n3-k3})
```

Parent topic *DFT-related scoped enumeration types*

Workspace placement

DFT implementations often require temporary storage for intermediate data whilst computing DFTs. This temporary storage is referred to as a *workspace*. Whilst this is managed automatically by default (`config_value::WORKSPACE_AUTOMATIC` set for `config_param::WORKSPACE_PLACEMENT`), it may be preferable to provide an external workspace (`config_value::WORKSPACE_EXTERNAL` set for `config_param::WORKSPACE_PLACEMENT`) for the following reasons:

- to reduce the number of mallocs / frees;
- to reduce memory consumption.

For some backends and configurations, `config_value::WORKSPACE_EXTERNAL` may reduce performance.

A typical workflow for using `config_value::WORKSPACE_EXTERNAL` is given in the section *below*.

WORKSPACE_PLACEMENT

For `config_param::WORKSPACE_PLACEMENT`, valid configuration values are `config_value::WORKSPACE_AUTOMATIC` and `config_value::WORKSPACE_EXTERNAL`.

WORKSPACE_AUTOMATIC

The default value for the `config_param::WORKSPACE_PLACEMENT` is `config_value::WORKSPACE_AUTOMATIC`.

When set to `config_value::WORKSPACE_AUTOMATIC` the user does not need to provide an external workspace. The workspace will be automatically managed by the backend library.

WORKSPACE_EXTERNAL

The configuration `config_param::WORKSPACE_PLACEMENT` can be set to `config_value::WORKSPACE_EXTERNAL` to allow the workspace to be set manually.

When a descriptor is committed with `config_value::WORKSPACE_EXTERNAL` set for `config_param::WORKSPACE_PLACEMENT`, the user must provide an external workspace before calling any compute function. See *set_workspace* and *Typical usage of WORKSPACE_EXTERNAL*.

Typical usage of `WORKSPACE_EXTERNAL`

Usage of `config_value::WORKSPACE_EXTERNAL` typically involves the following order of operations:

1. `config_value::WORKSPACE_EXTERNAL` is set for the uncommitted descriptor's `config_param::WORKSPACE_EXTERNAL`.
2. The descriptor is committed.
3. The required workspace size is queried.
4. A workspace of sufficient size is provided to the descriptor.
5. Compute functions following the type of external workspace provided are called.
6. The user is responsible for freeing the external workspace.

This is shown in the following example code:

```
// Create a descriptor
mkl::dft::descriptor<mkl::dft::precision::SINGLE, dom> desc(n);
// 1. Set the workspace placement to WORKSPACE_EXTERNAL
desc.set_value(mkl::dft::config_param::WORKSPACE_PLACEMENT,
               mkl::dft::config_value::WORKSPACE_EXTERNAL);
// Set further configuration parameters
// ...
// 2. Commit the descriptor
desc.commit(myQueue);
// 3. Query the required workspace size
std::int64_t workspaceBytes{0};
desc.get_value(mkl::dft::config_param::WORKSPACE_EXTERNAL_BYTES, &workspaceBytes);
```

(continues on next page)

(continued from previous page)

```
// Obtain a sufficiently large USM allocation or buffer. For this example, a USM␣
↪allocation is used.
float* workspaceUsm = sycl::malloc_device<float>(workspaceBytes / sizeof(float),␣
↪myQueue);
// 4. Set the workspace
desc.set_workspace(workspaceUsm);
// 5. Now USM compute functions can be called.
```

Parent topic: *DFT-related scoped enumeration types*

compute_forward

This function computes the forward DFT(s), as defined by an instantiation of the *descriptor* class, on user-provided data.

Description

Given a successfully committed *descriptor* object whose configuration is not inconsistent with forward DFT calculations, this function computes the forward transform defined by that object.

The `compute_forward` function requires a successfully committed object of the *descriptor* class and one, two or four "data container" arguments (depending on the configuration of the *descriptor* object). If using (pointers to) USM allocations as data containers, this function may also be provided with an `std::vector<sycl::event>` object collecting dependencies to be observed by the desired DFT calculations and return a `sycl::event` tracking the progress of the DFT calculations enqueued by this function.

Note: The compute_forward function may need to access the internals and private/protected members of the *descriptor* class. This could be done, for instance, by labeling it as a friend function to the *descriptor* class.

---

compute_forward (Buffer version)

Syntax (in-place transform, except for complex descriptors with config_value::REAL_REAL for config_param::COMPLEX_STORAGE)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type, typename data_type>
   void compute_forward( descriptor_type              &desc,
                         sycl::buffer<data_type, 1>   &inout);
}
```

Syntax (in-place transform, for complex descriptors with config_value::REAL_REAL for config_param::COMPLEX_STORAGE)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type typename data_type>
   void compute_forward( descriptor_type              &desc,
                         sycl::buffer<data_type, 1>   &inout_re,
                         sycl::buffer<data_type, 1>   &inout_im);
}
```

Syntax (out-of-place transform, except for complex descriptors with config_value::REAL_REAL for config_param::COMPLEX_STORAGE)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type, typename input_type, typename output_type>
   void compute_forward( descriptor_type              &desc,
                         sycl::buffer<input_type, 1>  &in,
                         sycl::buffer<output_type, 1> &out);
}
```

Syntax (out-of-place transform, for complex descriptors with config_value::REAL_REAL for config_param::COMPLEX_STORAGE)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type, typename input_type, typename output_type>
   void compute_forward( descriptor_type              &desc,
                         sycl::buffer<input_type, 1>  &in_re,
                         sycl::buffer<input_type, 1>  &in_im,
                         sycl::buffer<output_type, 1> &out_re,
                         sycl::buffer<output_type, 1> &out_im);
}
```

Input Parameters

desc
: A fully configured and committed object of the *descriptor* class, whose configuration is not inconsistent with forward DFT calculations.

inout
: `sycl::buffer` object of sufficient capacity to store the elements defining all the relevant data sequences, as configured by `desc` (configured for in-place operations and not with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*, if complex).

inout_re
: `sycl::buffer` object of sufficient capacity to store the elements defining the real parts of all the relevant data sequences, as configured by `desc`. `data_type` must be single or double precision floating-point, as described by the descriptor's precision. Only with complex descriptors configured for in-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

inout_im
`sycl::buffer` object of sufficient capacity to store the elements defining the imaginary parts of all the relevant data sequences, as configured by `desc`. `data_type` must be single or double precision floating-point, as described by the descriptor's precision. Only with complex descriptors configured for in-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

in
`sycl::buffer` object of sufficient capacity to store the elements defining all the relevant forward-domain data sequences, as configured by `desc` (configured for out-of-place operations and not with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*, if complex).

in_re
`sycl::buffer` object of sufficient capacity to store the elements defining the real parts of all the relevant forward-domain data sequences, as configured by `desc`. Only with complex descriptors configured for out-of-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

in_im
`sycl::buffer` object of sufficient capacity to store the elements defining the imaginary parts of all the relevant forward-domain data sequences, as configured by `desc`. Only with complex descriptors configured for out-of-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

Output Parameters

inout
`sycl::buffer` object of sufficient capacity to store the elements defining all the relevant data sequences, as configured by `desc` (configured for in-place operations and not with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*, if complex).

inout_re
`sycl::buffer` object of sufficient capacity to store the elements defining the real parts of all the relevant data sequences, as configured by `desc`. `data_type` must be single or double precision floating-point, as described by the descriptor's precision. Only with complex descriptors configured for in-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

inout_im
`sycl::buffer` object of sufficient capacity to store the elements defining the imaginary parts of all the relevant data sequences, as configured by `desc`. `data_type` must be single or double precision floating-point, as described by the descriptor's precision. Only with complex descriptors configured for in-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

out
`sycl::buffer` object of sufficient capacity to store the elements defining all the relevant backward-domain data sequences, as configured by `desc` (configured for out-of-place operations and not with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*, if complex).

out_re
`sycl::buffer` object of sufficient capacity to store the elements defining the real parts of all the relevant backward-domain data sequences, as configured by `desc`. Only with complex descriptors configured for out-of-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

out_im
`sycl::buffer` object of sufficient capacity to store the elements defining the imaginary parts of all the relevant backward-domain data sequences, as configured by `desc`. Only with complex descriptors configured for out-of-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

Throws

The `oneapi::mkl::dft::compute_forward` routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here:

oneapi::mkl::invalid_argument()
: If the provided *descriptor* object `desc` is invalid, for instance, if its configuration value associated with configuration parameter `config_param::COMMIT_STATUS` is not `config_param::COMMITTED`.

compute_forward (USM version)

Syntax (in-place transform, except for complex descriptors with config_value::REAL_REAL for config_param::COMPLEX_STORAGE)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type, typename data_type>
   sycl::event compute_forward( descriptor_type                                &desc,
                                data_type                                      *inout,
                                const std::vector<sycl::event>                 &
↪dependencies = {});
}
```

Syntax (in-place transform, for complex descriptors with config_value::REAL_REAL for config_param::COMPLEX_STORAGE)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type, typename data_type>
   sycl::event compute_forward( descriptor_type                                &desc,
                                data_type                                      *inout_re,
                                data_type                                      *inout_im,
                                const std::vector<sycl::event>                 &
↪dependencies = {});
}
```

Syntax (out-of-place transform, except for complex descriptors with config_value::REAL_REAL for config_param::COMPLEX_STORAGE)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type, typename input_type, typename output_type>
   sycl::event compute_forward( descriptor_type                                &desc,
                                input_type                                     *in,
                                output_type                                    *out,
                                const std::vector<sycl::event>                 &
↪dependencies = {});
}
```

Syntax (out-of-place transform, for complex descriptors with config_value::REAL_REAL for config_param::COMPLEX_STORAGE)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type, typename input_type, typename output_type>
   sycl::event compute_forward( descriptor_type                                &desc,
                                input_type                                     *in_re,
                                input_type                                     *in_im,
                                output_type                                    *out_re,
                                output_type                                    *out_im,
                                const std::vector<sycl::event>                 &
↪dependencies = {});
}
```

Input Parameters

desc
: A fully configured and committed object of the *descriptor* class, whose configuration is not inconsistent with forward DFT calculations.

inout
: Pointer to USM allocation of sufficient capacity to store the elements defining all the relevant data sequences, as configured by `desc` (configured for in-place operations and not with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*, if complex).

inout_re
: Pointer to USM allocation of sufficient capacity to store the elements defining the real parts of all the relevant data sequences, as configured by `desc`. `data_type` must be single or double precision floating-point, as described by the descriptor's precision. Only with complex descriptors configured for in-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

inout_im
: Pointer to USM allocation of sufficient capacity to store the elements defining the imaginary parts of all the relevant data sequences, as configured by `desc`. `data_type` must be single or double precision floating-point, as described by the descriptor's precision. Only with complex descriptors configured for in-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

in
: Pointer to USM allocation of sufficient capacity to store the elements defining all the relevant forward-domain data sequences, as configured by `desc` (configured for out-of-place operations and not with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*, if complex).

in_re
: Pointer to USM allocation of sufficient capacity to store the elements defining the real parts of all the relevant forward-domain data sequences, as configured by `desc`. Only with complex descriptors configured for out-of-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

in_im
: Pointer to USM allocation of sufficient capacity to store the elements defining the imaginary parts of all the relevant forward-domain data sequences, as configured by `desc`. Only with complex descriptors configured for out-of-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

dependencies
: An `std::vector<sycl::event>` object collecting the events returned by previously enqueued tasks that must be finished before this transform can be calculated.

Output Parameters

inout
Pointer to USM allocation of sufficient capacity to store the elements defining all the relevant data sequences, as configured by `desc` (configured for in-place operations and not with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*, if complex).

inout_re
Pointer to USM allocation of sufficient capacity to store the elements defining the real parts of all the relevant data sequences, as configured by `desc`. `data_type` must be single or double precision floating-point, as described by the descriptor's precision. Only with complex descriptors configured for in-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

inout_im
Pointer to USM allocation of sufficient capacity to store the elements defining the imaginary parts of all the relevant data sequences, as configured by `desc`. `data_type` must be single or double precision floating-point, as described by the descriptor's precision. Only with complex descriptors configured for in-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

out
Pointer to USM allocation of sufficient capacity to store the elements defining all the relevant backward-domain data sequences, as configured by `desc` (configured for out-of-place operations and not with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*, if complex).

out_re
Pointer to USM allocation of sufficient capacity to store the elements defining the real parts of all the relevant backward-domain data sequences, as configured by `desc`. Only with complex descriptors configured for out-of-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

out_im
Pointer to USM allocation of sufficient capacity to store the elements defining the imaginary parts of all the relevant backward-domain data sequences, as configured by `desc`. Only with complex descriptors configured for out-of-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

Throws

The `oneapi::mkl::dft::compute_forward()` routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here:

oneapi::mkl::invalid_argument()
If the provided *descriptor* object `desc` is invalid, for instance, if its configuration value associated with configuration parameter `config_param::COMMIT_STATUS` is not `config_param::COMMITTED`. It will also be thrown if any required input/output pointer is `nullptr`.

Return Values

This function returns a `sycl::event` object that allows to track progress of the forward DFT, and can be passed as a dependency to other routines that may depend on the result of the forward transform(s) before proceeding with other operations.

Parent topic: *Discrete Fourier Transform Functions*

compute_backward

This function computes the backward DFT(s), as defined by an instantiation of the *descriptor* class, on user-provided data.

Description

Given a successfully committed *descriptor* object whose configuration is not inconsistent with backward DFT calculations, this function computes the backward transform defined by that object.

The `compute_backward` function requires a successfully committed object of the *descriptor* class and one, two or four "data container" arguments (depending on the configuration of the *descriptor* object). If using (pointers to) USM allocations as data containers, this function may also be provided with an `std::vector<sycl::event>` object collecting dependencies to be observed by the desired DFT calculations and return a `sycl::event` tracking the progress of the DFT calculations enqueued by this function.

Note: The compute_backward function may need to access the internals and private/protected members of the *descriptor* class. This could be done, for instance, by labeling it as a friend function to the *descriptor* class.

---

compute_backward (Buffer version)

Syntax (in-place transform, except for complex descriptors with config_value::REAL_REAL for config_param::COMPLEX_STORAGE)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type, typename data_type>
   void compute_backward( descriptor_type              &desc,
                          sycl::buffer<data_type, 1>   &inout);
}
```

Syntax (in-place transform, for complex descriptors with config_value::REAL_REAL for config_param::COMPLEX_STORAGE)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type typename data_type>
   void compute_backward( descriptor_type              &desc,
                          sycl::buffer<data_type, 1>   &inout_re,
                          sycl::buffer<data_type, 1>   &inout_im);
}
```

Syntax (out-of-place transform, except for complex descriptors with config_value::REAL_REAL for config_param::COMPLEX_STORAGE)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type, typename input_type, typename output_type>
   void compute_backward( descriptor_type              &desc,
                          sycl::buffer<input_type, 1>  &in,
                          sycl::buffer<output_type, 1> &out);
}
```

Syntax (out-of-place transform, for complex descriptors with config_value::REAL_REAL for config_param::COMPLEX_STORAGE)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type, typename input_type, typename output_type>
   void compute_backward( descriptor_type              &desc,
                          sycl::buffer<input_type, 1>  &in_re,
                          sycl::buffer<input_type, 1>  &in_im,
                          sycl::buffer<output_type, 1> &out_re,
                          sycl::buffer<output_type, 1> &out_im);
}
```

Input Parameters

desc
: A fully configured and committed object of the *descriptor* class, whose configuration is not inconsistent with backward DFT calculations.

inout
: `sycl::buffer` object of sufficient capacity to store the elements defining all the relevant data sequences, as configured by `desc` (configured for in-place operations and not with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*, if complex).

inout_re
: `sycl::buffer` object of sufficient capacity to store the elements defining the real parts of all the relevant data sequences, as configured by `desc`. `data_type` must be single or double precision floating-point, as described by the descriptor's precision. Only with complex descriptors configured for in-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

inout_im
: `sycl::buffer` object of sufficient capacity to store the elements defining the imaginary parts of all the relevant data sequences, as configured by `desc`. `data_type` must be single or double precision floating-point, as described by the descriptor's precision. Only with complex descriptors configured for in-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

in
: `sycl::buffer` object of sufficient capacity to store the elements defining all the relevant backward-domain data sequences, as configured by `desc` (configured for out-of-place operations and not with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*, if complex).

in_re
: `sycl::buffer` object of sufficient capacity to store the elements defining the real parts of all the relevant

backward-domain data sequences, as configured by `desc`. Only with complex descriptors configured for out-of-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

in_im
`sycl::buffer` object of sufficient capacity to store the elements defining the imaginary parts of all the relevant backward-domain data sequences, as configured by `desc`. Only with complex descriptors configured for out-of-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

Output Parameters

inout
`sycl::buffer` object of sufficient capacity to store the elements defining all the relevant data sequences, as configured by `desc` (configured for in-place operations and not with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*, if complex).

inout_re
`sycl::buffer` object of sufficient capacity to store the elements defining the real parts of all the relevant data sequences, as configured by `desc`. `data_type` must be single or double precision floating-point, as described by the descriptor's precision. Only with complex descriptors configured for in-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

inout_im
`sycl::buffer` object of sufficient capacity to store the elements defining the imaginary parts of all the relevant data sequences, as configured by `desc`. `data_type` must be single or double precision floating-point, as described by the descriptor's precision. Only with complex descriptors configured for in-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

out
`sycl::buffer` object of sufficient capacity to store the elements defining all the relevant forward-domain data sequences, as configured by `desc` (configured for out-of-place operations and not with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*, if complex).

out_re
`sycl::buffer` object of sufficient capacity to store the elements defining the real parts of all the relevant forward-domain data sequences, as configured by `desc`. Only with complex descriptors configured for out-of-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

out_im
`sycl::buffer` object of sufficient capacity to store the elements defining the imaginary parts of all the relevant forward-domain data sequences, as configured by `desc`. Only with complex descriptors configured for out-of-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

Throws

The `oneapi::mkl::dft::compute_backward` routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here:

oneapi::mkl::invalid_argument()
If the provided *descriptor* object `desc` is invalid, for instance, if its configuration value associated with configuration parameter `config_param::COMMIT_STATUS` is not `config_param::COMMITTED`.

compute_backward (USM version)

Syntax (in-place transform, except for complex descriptors with config_value::REAL_REAL for config_param::COMPLEX_STORAGE)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type, typename data_type>
   sycl::event compute_backward( descriptor_type                                &desc,
                                 data_type                                      *inout,
                                 const std::vector<sycl::event>                 &
↪dependencies = {});
}
```

Syntax (in-place transform, for complex descriptors with config_value::REAL_REAL for config_param::COMPLEX_STORAGE)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type, typename data_type>
   sycl::event compute_backward( descriptor_type                                &desc,
                                 data_type                                      *inout_re,
                                 data_type                                      *inout_im,
                                 const std::vector<sycl::event>                 &
↪dependencies = {});
}
```

Syntax (out-of-place transform, except for complex descriptors with config_value::REAL_REAL for config_param::COMPLEX_STORAGE)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type, typename input_type, typename output_type>
   sycl::event compute_backward( descriptor_type                                &desc,
                                 input_type                                     *in,
                                 output_type                                    *out,
                                 const std::vector<sycl::event>                 &
↪dependencies = {});
}
```

Syntax (out-of-place transform, for complex descriptors with config_value::REAL_REAL for config_param::COMPLEX_STORAGE)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type, typename input_type, typename output_type>
   sycl::event compute_backward( descriptor_type                                &desc,
                                 input_type                                     *in_re,
                                 input_type                                     *in_im,
                                 output_type                                    *out_re,
                                 output_type                                    *out_im,
                                 const std::vector<sycl::event>                 &
↪dependencies = {});
}
```

Input Parameters

desc
: A fully configured and committed object of the *descriptor* class, whose configuration is not inconsistent with backward DFT calculations.

inout
: Pointer to USM allocation of sufficient capacity to store the elements defining all the relevant data sequences, as configured by `desc` (configured for in-place operations and not with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*, if complex).

inout_re
: Pointer to USM allocation of sufficient capacity to store the elements defining the real parts of all the relevant data sequences, as configured by `desc`. `data_type` must be single or double precision floating-point, as described by the descriptor's precision. Only with complex descriptors configured for in-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

inout_im
: Pointer to USM allocation of sufficient capacity to store the elements defining the imaginary parts of all the relevant data sequences, as configured by `desc`. `data_type` must be single or double precision floating-point, as described by the descriptor's precision. Only with complex descriptors configured for in-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

in
: Pointer to USM allocation of sufficient capacity to store the elements defining all the relevant backward-domain data sequences, as configured by `desc` (configured for out-of-place operations and not with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*, if complex).

in_re
: Pointer to USM allocation of sufficient capacity to store the elements defining the real parts of all the relevant backward-domain data sequences, as configured by `desc`. Only with complex descriptors configured for out-of-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

in_im
: Pointer to USM allocation of sufficient capacity to store the elements defining the imaginary parts of all the relevant backward-domain data sequences, as configured by `desc`. Only with complex descriptors configured for out-of-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

dependencies
: An `std::vector<sycl::event>` object collecting the events returned by previously enqueued tasks that must be finished before this transform can be calculated.

Output Parameters

inout
Pointer to USM allocation of sufficient capacity to store the elements defining all the relevant data sequences, as configured by `desc` (configured for in-place operations and not with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*, if complex).

inout_re
Pointer to USM allocation of sufficient capacity to store the elements defining the real parts of all the relevant data sequences, as configured by `desc`. `data_type` must be single or double precision floating-point, as described by the descriptor's precision. Only with complex descriptors configured for in-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

inout_im
Pointer to USM allocation of sufficient capacity to store the elements defining the imaginary parts of all the relevant data sequences, as configured by `desc`. `data_type` must be single or double precision floating-point, as described by the descriptor's precision. Only with complex descriptors configured for in-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

out
Pointer to USM allocation of sufficient capacity to store the elements defining all the relevant forward-domain data sequences, as configured by `desc` (configured for out-of-place operations and not with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*, if complex).

out_re
Pointer to USM allocation of sufficient capacity to store the elements defining the real parts of all the relevant forward-domain data sequences, as configured by `desc`. Only with complex descriptors configured for out-of-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

out_im
Pointer to USM allocation of sufficient capacity to store the elements defining the imaginary parts of all the relevant forward-domain data sequences, as configured by `desc`. Only with complex descriptors configured for out-of-place operations with *config_value::REAL_REAL for config_param::COMPLEX_STORAGE*.

Throws

The `oneapi::mkl::dft::compute_backward()` routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here:

oneapi::mkl::invalid_argument()
If the provided *descriptor* object `desc` is invalid, for instance, if its configuration value associated with configuration parameter `config_param::COMMIT_STATUS` is not `config_param::COMMITTED`. It will also be thrown if any required input/output pointer is `nullptr`.

Return Values

This function returns a `sycl::event` object that allows to track progress of the backward DFT, and can be passed as a dependency to other routines that may depend on the result of the backward transform(s) before proceeding with other operations.

Parent topic: *Discrete Fourier Transform Functions*

9.2.4 Random Number Generators

The oneAPI Math Kernel Library Random Number Generators provides a *set of routines* implementing commonly used *pseudorandom, quasi-random, and non-deterministic generators* with *continuous and discrete distributions*.

Random Number Generators (RNG)

Definitions

The pseudo-random number generator is defined by a structure(S, μ, f, U, g), where:

- S is a finite set of states (the state space)
- μ is a probability distribution on S for the initial state (or seed) s_0
- $f : S \to S$ is the transition function
- U – a finite set of output symbols
- $g : S \to U$ an output function

The generation of random numbers is as follows:

1. Generate the initial state (called the seed) s_0 according to μ and compute $u_0 = g(s_0)$.
2. Iterate for $i = 1, ..., ‘ : math : ‘s_i = f(s_{i-1})$ and $u_i = g(s_i)$. Output values u_i are the so-called random numbers produced by the PRNG.

In computational statistics, random variate generation is usually made in two steps:

1. Generating imitations of independent and identically distributed (i.i.d.) random variables having the uniform distribution over the interval $(0, 1)$
2. Applying transformations to these i.i.d. $U(0, 1)$ random variates in order to generate (or imitate) random variates and random vectors from arbitrary distributions.

Execution Models

RNG domain supports two execution models:

1. *Host API*, which is aligned with the rest of oneMKL domains *oneMKL domains*.
2. *Device API*, which is specific for RNG domain. These APIs are designed to be callable from the User's kernels as well as Host code.

Random Number Generators Host Routines

Structure

RNG domain contains two classes types:

- Engines (basic random number generators) classes, which holds the state of generator and is a source of independent and identically distributed random variables. Refer to *Host Engines (Basic Random Number Generators)* for a detailed description.
- Distribution classes templates (transformation classes) for different types of statistical distributions, for example, uniform, normal (Gaussian), binomial, etc. These classes contain all of the distribution's parameters (including generation method). Refer to *Host Distributions* for a detailed description of the distributions.

The RNG domain also contains two types of free functions:

- Generation routines. The current routines are used to obtain random numbers from a given engine with proper statistics defined by a given distribution. Refer to the *Host Generate Routine* section for a detailed description.
- Service routines. The routines are used to modify the engine state. Refer to *Host Service Routines* for a description of these routines.

Engine classes work with both generation and service routines. Distribution classes are used in generation routines only. Refer to the *oneMKL RNG Host Usage Model* section for the description of typical RNG scenario.

oneMKL RNG Host Usage Model

Description

A typical algorithm for random number generators is as follows:

1. Create and initialize the object for basic random number generator.
    - Use the *skip_ahead* or *leapfrog* function if it is required (used in parallel with random number generation for Host and CPU devices).
2. Create and initialize the object for distribution generator.
3. Call the generate routine to get random numbers with appropriate statistical distribution.

The following example demonstrates random numbers generation with PHILOX4X32X10 basic generator (engine).

Buffer-based example

```
#include "oneapi/mkl/rng.hpp"

int main() {
    sycl::queue q;

    // Create the random number generator object
    oneapi::mkl::rng::philox4x32x10 engine(q, seed);
    // Create the distribution object
    oneapi::mkl::rng::gaussian<double> distr(5.0, 2.0);
    // Fill the SYCL buffer with random numbers
    oneapi::mkl::rng::generate(distr, engine, n, sycl_buffer);

    // ...
}
```

USM-based example

```
#include "oneapi/mkl/rng.hpp"

int main() {
    sycl::queue q;

    // Create the random number generator object
    oneapi::mkl::rng::philox4x32x10 engine(q, seed);
    // Create the distribution object
    oneapi::mkl::rng::gaussian<double> distr(5.0, 2.0);
    // Fill the USM memory under the pointer with random numbers
    auto event = oneapi::mkl::rng::generate(distr, engine, n, usm_ptr);
    // ...
    // wait until generation is finalized
    event.wait();
    // ...
}
```

Parent topic: *Random Number Generators Host Routines*

Host Generate Routine

- *generate* Entry point to obtain random numbers from a given engine with proper statistics of a given distribution.

Parent topic: *Random Number Generators Host Routines*

generate

Entry point to obtain random numbers from a given engine with proper statistics of a given distribution.

Description and Assumptions

oneapi::mkl::rng::generate function produces random numbers sequence from the given engine object and applied transformation from a given distribution object.

generate (Buffer version)

Syntax

```
namespace oneapi::mkl::rng {
template<typename DistrType, typename EngineType>
void generate (const DistrType& distr, EngineType& engine, std::int64_t n, sycl::buffer
↪<typename DistrType::result_type, 1>& r);
}
```

Template Parameters

DistrType
Type of distribution which is used for random number generation.

EngineType
Type of engine which is used for random number generation.

Input Parameters

distr
Distribution object. See *Host Distributions* for details.

engine
Engine object. See *Host Engines (Basic Random Number Generators)* for details.

n
Number of random values to be generated.

Output Parameters

r
sycl::buffer of generated values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when n > r.get_count(), or n < 0

generate (USM version)

Syntax

```
namespace oneapi::mkl::rng {
template<typename DistrType, typename EngineType>
sycl::event generate (const DistrType& distr, EngineType& engine, std::int64_t n,␣
↪typename DistrType::result_type* r, const std::vector<sycl::event> & dependencies);
}
```

Template Parameters

DistrType
Type of distribution which is used for random number generation.

EngineType
Type of engine which is used for random number generation.

Input Parameters

distr
Distribution object. See *Host Distributions* for details.

engine
Engine object. See *Host Engines (Basic Random Number Generators)* for details.

n
Number of random values to be generated.

dependencies
Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

r
pointer to generated values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when r == nullptr, or n < 0

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Host Generate Routine*

Host Engines (Basic Random Number Generators)

oneMKL RNG provides pseudorandom, quasi-random, and non-deterministic random number generators for Data Parallel C++:

Routine	Description
default_en	The default random engine
mrg32k3	The combined multiple recursive pseudorandom number generator `MRG32k3a` *[L'Ecuyer99a]*
philox4x	Philox4x32-10 counter-based pseudorandom number generator with a period of 2^{128} `PHILOX4X32X10` *[Salmon11]*
mcg31m	The 31-bit multiplicative congruential pseudorandom number generator MCG(1132489760, 231 -1) *[L'Ecuyer99]*
r250	The 32-bit generalized feedback shift register pseudorandom number generator `GFSR(250, 103)` *[Kirkpatrick81]*
mcg59	The 59-bit multiplicative congruential pseudorandom number generator `MCG(13`^{13}`, 2`^{59}) from NAG Numerical Libraries *[NAG]*
wichmann_h	Wichmann-Hill pseudorandom number generator (a set of 273 basic generators) from NAG Numerical Libraries *[NAG]*
mt19937	Mersenne Twister pseudorandom number generator `MT19937` *[Matsumoto98]* with period length 2^{19937}-1 of the produced sequence
mt2203	Set of 6024 Mersenne Twister pseudorandom number generators `MT2203` *[Matsumoto98]*, *[Matsumoto00]*. Each of them generates a sequence of period length equal to 2^{2203}-1. Parameters of the generators provide mutual independence of the corresponding sequences.
sfmt199	SIMD-oriented Fast Mersenne Twister pseudorandom number generator `SFMT19937` *[Saito08]* with a period length equal to 2^{19937}-1 of the produced sequence.
sobol	Sobol quasi-random number generator *[Sobol76]*, *[Bratley88]*, which works in arbitrary dimension.
niederreiter	Niederreiter quasi-random number generator *[Bratley92]*, which works in arbitrary dimension.
ars5	ARS-5 counter-based pseudorandom number generator with a period of 2^{128}, which uses instructions from the AES-NI set `ARS5` *[Salmon11]*.
nondeterministic	Non-deterministic random number generator

For some basic generators, oneMKL RNG provides two methods of creating independent states in multiprocessor computations, which are the leapfrog method and the block-splitting method. These sequence splitting methods are also useful in sequential Monte Carlo. The description of these functions can be found in the *Host Service Routines* section.

In addition, the MT2203 pseudorandom number generator is a set of 6024 generators designed to create up to 6024 independent random sequences, which might be used in parallel Monte Carlo simulations. Another generator that has the same feature is Wichmann-Hill. It allows creating up to 273 independent random streams. The properties of the generators designed for parallel computations are discussed in detail in [*Coddington94*].

Parent topic: *Random Number Generators Host Routines*

- *default_engine* The default random engine (implementation defined)
- *mrg32k3a* The combined multiple recursive pseudorandom number generator MRG32k3a [L'Ecuyer99a]
- *philox4x32x10* A Philox4x32-10 counter-based pseudorandom number generator. [Salmon11].

- *mcg31m1* The 31-bit multiplicative congruential pseudorandom number generator MCG(1132489760, 231 -1) [L'Ecuyer99]
- *mcg59* The 59-bit multiplicative congruential pseudorandom number generator MCG(1313, 259) from NAG Numerical Libraries [NAG].
- *r250* The 32-bit generalized feedback shift register pseudorandom number generator GFSR(250,103)[Kirkpatrick81].
- *wichmann_hill* Wichmann-Hill pseudorandom number generator (a set of 273 basic generators) from NAG Numerical Libraries [NAG].
- *mt19937* Mersenne Twister pseudorandom number generator MT19937 [Matsumoto98] with period length 2^{19937}-1 of the produced sequence.
- *sfmt19937* SIMD-oriented Fast Mersenne Twister pseudorandom number generator SFMT19937 [Saito08] with a period length equal to 2^{19937}-1 of the produced sequence.
- *mt2203* Set of 6024 Mersenne Twister pseudorandom number generators MT2203 [Matsumoto98], [Matsumoto00]. Each of them generates a sequence of period length equal to 2^{2203}-1. Parameters of the generators provide mutual independence of the corresponding sequences..
- *ars5* ARS-5 counter-based pseudorandom number generator with a period of 2^{128}, which uses instructions from the AES-NI set ARS5[Salmon11].
- *sobol* Sobol quasi-random number generator [Sobol76], [Bratley88], which works in arbitrary dimension.
- *niederreiter* Niederreiter quasi-random number generator [Bratley92], which works in arbitrary dimension.
- *nondeterministic* Non-deterministic random number generator.

default_engine

Default random engine.

Description

The choice of engine type named by default_engine is implementation-defined. The implementation may select this type on the basis of performance, size, quality, or any combination of such factors.

type alias default_engine

Syntax

```
using default_engine = implementation-defined;
```

Parent topic: *Host Engines (Basic Random Number Generators)*

mrg32k3a

The combined multiple recursive pseudorandom number generator MRG32k3a.

Description

MRG32k3a engine is a 32-bit combined multiple recursive generator with two components of order 3 *[L'Ecuyer99a]*. MRG32k3a combined generator meets the requirements for modern RNGs, such as good multidimensional uniformity, or a long period ($p \approx 2^{191}$).

Generation algorithm

$x_n = a_{11}x_{n-1} + a_{12}x_{n-2} + a_{13}x_{n-3}(mod\, m_1)$

$y_n = a_{21}y_{n-1} + a_{22}y_{n-2} + a_{23}(mod\, m_2)$

$z_n = x_n - y_n(mod\, m_1)$

$u_n = z_n/m_1$

$a_{11} = 0, a_{12} = 1403580, a_{13} = -810728, m_1 = 2^{32} - 209$

$a_{21} = 527612, a_{22} = 0, a_{23} = -1370589, m_2 = 2^{32} - 22853$

class mrg32k3a

Syntax

```
namespace oneapi::mkl::rng {
class mrg32k3a {
public:
    static constexpr std::uint32_t default_seed = 1;

    mrg32k3a(sycl::queue queue, std::uint32_t seed = default_seed);

    mrg32k3a(sycl::queue queue, std::initializer_list<std::uint32_t> seed);

    mrg32k3a(const mrg32k3a& other);

    mrg32k3a(mrg32k3a&& other);

    mrg32k3a& operator=(const mrg32k3a& other);

    mrg32k3a& operator=(mrg32k3a&& other);

    ~mrg32k3a();
};
}
```

Class Members

Routine	Description
mrg32k3a(sycl::queue queue, std::uint32_t seed = default_seed)	Constructor for common seed initialization of the engine
mrg32k3a(sycl::queue queue, std::initializer_list<std::uint32_t> seed)	Constructor for extended seed initialization of the engine
mrg32k3a(const mrg32k3a& other)	Copy constructor
mrg32k3a(mrg32k3a&& other)	Move constructor
mrg32k3a& operator=(const mrg32k3a& other)	Copy assignment operator
mrg32k3a& operator=(mrg32k3a&& other)	Move assignment operator

Constructors

```
mrg32k3a::mrg32k3a(sycl::queue queue, std::uint32_t seed = default_seed)
```

Input Parameters

queue

Valid `sycl::queue object`, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed

The initial conditions of the generator state, assume $x_{-3} = seed \ mod \ m_1, x_{-2} = x_{-1} = y_{-3} = y_{-2} = y_{-1} = 1$.

```
mrg32k3a::mrg32k3a(sycl::queue queue, std::initializer_list<std::uint32_t> seed)
```

Input Parameters

queue

Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed

The initial conditions of the generator state, assume if $n = 0 : x_{-3} = x_{-2} = x_{-1} = y_{-3} = y_{-2} = y_{-1} = 1$

if $n = 1 : x_{-3} = seed[0] \ mod \ m_1, x_{-2} = x_{-1} = y_{-3} = y_{-2} = y_{-1} = 1$

if $n = 2 : x_{-3} = seed[0] \ mod \ m_1, x_{-2} = seed[1] \ mod \ m_1, x_{-1} = y_{-3} = y_{-2} = y_{-1} = 1$

if $n = 3 : x_{-3} = seed[0] \ mod \ m_1, x_{-2} = seed[1] \ mod \ m_1, x_{-1} = seed[2] \ mod \ m_1$

$y_{-3} = y_{-2} = y_{-1} = 1$

if $n = 4 : x_{-3} = seed[0] \ mod \ m_1, x_{-2} = seed[1] \ mod \ m_1, x_{-1} = seed[2] \ mod \ m_1$

$y_{-3} = seed[3] \ mod \ m_2, y_{-2} = y_{-1} = 1$

if $n = 5 : x_{-3} = seed[0] \ mod \ m_1, x_{-2} = seed[1] \ mod \ m_1, x_{-1} = seed[2] \ mod \ m_1$

$y_{-3} = seed[3] \ mod \ m_2, y_{-2} = seed[4] \ mod \ m_2, y_{-1} = 1$

if $n \geqslant 6 : x_{-3} = seed[0] \ mod \ m_1, x_{-2} = seed[1] \ mod \ m_1, x_{-1} = seed[2] \ mod \ m_1$

$y_{-3} = seed[3] \ mod \ m_2, y_{-2} = seed[4] \ mod \ m_2, y_{-1} = seed[5] \ mod \ m_2$

if the values prove to be $x_{-3} = x_{-2} = x_{-1} = 0$, assume $x_{-3} = 1$

if the values prove to be $y_{-3} = y_{-2} = y_{-1} = 0$, assume $y_{-3} = 1$

```
mrg32k3a::mrg32k3a(const mrg32k3a& other)
```

Input Parameters

other
Valid `mrg32k3a` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
mrg32k3a::mrg32k3a(mrg32k3a&& other)
```

Input Parameters

other
Valid `mrg32k3a` object. The `queue` and state of the other engine is moved to the current engine.

```
mrg32k3a::mrg32k3a& operator=(const mrg32k3a& other)
```

Input Parameters

other
Valid `mrg32k3a` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
mrg32k3a::mrg32k3a& operator=(mrg32k3a&& other)
```

Input Parameters

other
Valid `mrg32k3a` r-value object. The `queue` and state of the other engine is moved to the current engine.

Parent topic: *Host Engines (Basic Random Number Generators)*

philox4x32x10

The Philox4x32x10 counter-based pseudorandom number generator.

Description

The Philox4x32x10 engine is a keyed family of generator of counter-based BRNG. The state consists of 128-bit integer counter c and two 32-bits keys k_0 and k_1.

Generation algorithm

The generator has 32-bit integer output obtained in the following way [*Salmon11*]:

1. $c_n = c_{n-1} + 1$
2. $\omega_n = f(c_n)$**, where f is a function that takes 128-bit argument and returns a 128-bit number. The returned number is obtained as follows:**

   2.1. The argument c is interpreted as four 32-bit numbers $c = \overline{L_1 R_1 L_0 R_0}$, where $\overline{ABCD} = A \cdot 2^{96} + B \cdot 2^{64} + C \cdot 2^{32} + D$, put $k_0^0 = k_0, k_1^0 = k_1$.

   2.2. The following recurrence is calculated:

   $L_1^{i+1} = mullo(R_1^i, 0xD2511F53)$

   $R_1^{i+1} = mulhi(R_0^i, 0xCD9E8D57) \oplus k_0^i \oplus L_0^i$

   $L_0^{i+1} = mullo(R_0^i, 0xCD9E8D57)$

   $R_0^{i+1} = mulhi(R_1^i, 0xD2511F53) \oplus k_1^i \oplus L_1^i$

   $k_0^{i+1} = k_0^i + 0xBB67AE85$

   $k_1^{i+1} = k_1^i + 0x9E3779B9$, where $mulhi(a, b)$ and $mullo(a, b)$ are high and low parts of the $a \cdot b$ product respectively.

   2.3. Put $f(c) = \overline{L_1^N R_1^N L_0^N R_0^N}$, where $N = 10$
3. Integer output: $r_{4n+k} = \omega_n(k)$, where $\omega_n(k)$ is the k-th 32-bit integer in quadruple $\omega_n, k = 0, 1, 2, 3$
4. Real output: $u_n = (int) r_n / 2^{32} + 1/2$

class philox4x32x10

Syntax

```
namespace oneapi::mkl::rng {
class philox4x32x10 {
public:
static constexpr std::uint64_t default_seed = 0;

philox4x32x10(sycl::queue queue, std::uint64_t seed = default_seed);

philox4x32x10(sycl::queue queue, std::initializer_list<std::uint64_t> seed);

philox4x32x10(const philox4x32x10& other);

philox4x32x10(philox4x32x10&& other);

philox4x32x10& operator=(const philox4x32x10& other);

philox4x32x10& operator=(philox4x32x10&& other);

~philox4x32x10();
};
}
```

Class Members

Routine	Description
philox4x32x10(sycl::queue queue, std::uint64_t seed = default_seed)	Constructor for common seed initialization of the engine
philox4x32x10(sycl::queue queue, std::initializer_list<std::uint64_t> seed)	Constructor for extended seed initialization of the engine
philox4x32x10(const philox4x32x10& other)	Copy constructor
philox4x32x10(philox4x32x10&& other)	Move constructor
philox4x32x10& operator=(const philox4x32x10& other)	Copy assignment operator
philox4x32x10& operator=(philox4x32x10&& other)	Move assignment operator

Constructors

```
philox4x32x10::philox4x32x10(sycl::queue queue, std::uint64_t seed = default_seed)
```

Input Parameters

queue
Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed
The initial conditions of the generator state, assume $k = seed, c = 0$, where k is a 64-bit key, c is a 128-bit counter.

```
philox4x32x10::philox4x32x10(sycl::queue queue, std::initializer_list<std::uint64_t>␣
↪seed)
```

Input Parameters

queue
Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed
The initial conditions of the generator state, assume if $n = 0 : k = 0, c = 0$

if $n = 1 : k = seed[0], c = 0$

if $n = 2 : k = seed[0], c = seed[1]$

if $n = 3 : k = seed[0], c = seed[1] + seed[2] \cdot 2^{64}$

for $n > 3$ following arguments are ignored

```
philox4x32x10::philox4x32x10(const philox4x32x10& other)
```

Input Parameters

other
Valid `philox4x32x10` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
philox4x32x10::philox4x32x10(philox4x32x10&& other)
```

Input Parameters

other
Valid `philox4x32x10` r-value object. The `queue` and state of the other engine is moved to the current engine.

```
philox4x32x10::philox4x32x10& operator=(const philox4x32x10& other)
```

Input Parameters

other
Valid `philox4x32x10` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
philox4x32x10::philox4x32x10& operator=(philox4x32x10&& other)
```

Input Parameters

other
Valid `philox4x32x10` r-value object. The `queue` and state of the other engine is moved to the current engine.

Parent topic: *Host Engines (Basic Random Number Generators)*

mcg31m1

The 31-bit multiplicative congruential pseudorandom number generator MCG(1132489760, 231 -1).

Description

The mcg31m1 engine is a 31-bit multiplicative congruential generator *[L'Ecuyer99]*. The mcg31m1 generator belongs to linear congruential generators with the period length of approximately 2^{31}. Such generators are still used as default random number generators in various software systems, mainly due to the simplicity of the portable versions implementation, speed, and compatibility with the earlier systems versions. However, their period length does not meet the requirements for modern basic generators. Still, the mcg31m1 generator possesses good statistic properties and you may successfully use it to generate random numbers of different distributions for small samplings.

Generation algorithm

$x_n = ax_{n-1}(mod\, m)$

$u_n = x_n/m$

$a = 1132489760, m = 2^{31} - 1$

class mcg31m1

Syntax

```
namespace oneapi::mkl::rng {
class mcg31m1 {
public:
    static constexpr std::uint32_t default_seed = 1;

    mcg31m1(sycl::queue queue, std::uint32_t seed = default_seed);

    mcg31m1(const mcg31m1& other);

    mcg31m1(mcg31m1&& other);

    mcg31m1& operator=(const mcg31m1& other);

    mcg31m1& operator=(mcg31m1&& other);

    ~mcg31m1();
};
}
```

Class Members

Routine	Description
mcg31m1(sycl::queue queue, std::uint32_t seed = default_seed)	Constructor for common seed initialization of the engine
mcg31m1(const mcg31m1& other)	Copy constructor
mcg31m1(mcg31m1&& other)	Move constructor
mcg31m1& operator=(const mcg31m1& other)	Copy assignment operator
mcg31m1& operator=(mcg31m1&& other)	Move assignment operator

Constructors

```
mcg31m1::mcg31m1(sycl::queue queue, std::uint32_t seed = default_seed)
```

Input Parameters

queue
Valid sycl::queue object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed
The initial conditions of the generator state, assume $x_0 = seed \ mod \ 0x7FFFFFFF$, if $x_0 = 0$, assume $x_0 = 1$.

```
mcg31m1::mcg31m1(const mcg31m1& other)
```

Input Parameters

other
Valid `mcg31m1` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
mcg31m1::mcg31m1(mcg31m1&& other)
```

Input Parameters

other
Valid `mcg31m1` object. The `queue` and state of the other engine is moved to the current engine.

```
mcg31m1::mcg31m1& operator=(const mcg31m1& other)
```

Input Parameters

other
Valid `mcg31m1` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
mcg31m1::mcg31m1& operator=(mcg31m1&& other)
```

Input Parameters

other
Valid `mcg31m1` r-value object. The `queue` and state of the other engine is moved to the current engine.

Parent topic: *Host Engines (Basic Random Number Generators)*

mcg59

The 59-bit multiplicative congruential pseudorandom number generator.

Description

The mcg59 engine is a 59-bit multiplicative congruential generator from NAG Numerical Libraries *NAG*. The mcg59 generator belongs to linear congruential generators with the period length of approximately 2^{57}.

Generation algorithm

$x_n = ax_{n-1}(mod\ m)$

$u_n = x_n/m$

$a = 13^{13}, m = 2^{59}$

class mcg59

Syntax

```
namespace oneapi::mkl::rng {
class mcg59 {
public:
    static constexpr std::uint64_t default_seed = 1;

    mcg59(sycl::queue queue, std::uint64_t seed = default_seed);

    mcg59(const mcg59& other);

    mcg59(mcg59&& other);

    mcg59& operator=(const mcg59& other);

    mcg59& operator=(mcg59&& other);

    ~mcg59();
};
}
```

Class Members

Routine	Description
mcg59(sycl::queue queue, std::uint64_t seed = default_seed)	Constructor for common seed initialization of the engine
mcg59(const mcg59& other)	Copy constructor
mcg59(mcg59&& other)	Move constructor
mcg59& operator=(const mcg59& other)	Copy assignment operator
mcg59& operator=(mcg59&& other)	Move assignment operator

Constructors

```
mcg59::mcg59(sycl::queue queue, std::uint64_t seed = default_seed)
```

Input Parameters

queue
Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed
The initial conditions of the generator state, assume $x_0 = seed \ mod \ 2^{59}$, if $x_0 = 0$, assume $x_0 = 1$.

```
mcg59::mcg59(const mcg59& other)
```

Input Parameters

other
Valid `mcg59` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
mcg59::mcg59(mcg59&& other)
```

Input Parameters

other
Valid `mcg59` object. The `queue` and state of the other engine is moved to the current engine.

```
mcg59::mcg59& operator=(const mcg59& other)
```

Input Parameters

other
Valid `mcg59` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
mcg59::mcg59& operator=(mcg59&& other)
```

Input Parameters

other
Valid `mcg59` r-value object. The `queue` and state of the other engine is moved to the current engine.

Parent topic: *Host Engines (Basic Random Number Generators)*

r250

The 32-bit generalized feedback shift register pseudorandom number generator GFSR(250,103) *[Kirkpatrick81]*.

Description

Feedback shift register generators possess ample theoretical foundation and were initially intended for cryptographic and communication applications. The stream state is the array of 250 32-bit integers.

Generation algorithm

$x_n = x_{n-103} \oplus x_{n-250}$

$u_n = x_n / 2^{32}$

class r250

Syntax

```
namespace oneapi::mkl::rng {
class r250 {
public:
    static constexpr std::uint32_t default_seed = 1;

    r250(sycl::queue queue, std::uint32_t seed = default_seed);

    r250(sycl::queue queue, std::vector<std::uint32_t> seed);

    r250(const r250& other);

    r250(r250&& other);

    r250& operator=(const r250& other);

    r250& operator=(r250&& other);

    ~r250();
};
}
```

Class Members

Routine	Description
r250(sycl::queue queue, std::uint32_t seed = default_seed)	Constructor for common seed initialization of the engine
r250(sycl::queue queue, std::vector<std::uint32_t> seed)	Constructor for extended seed initialization of the engine
r250(const r250& other)	Copy constructor
r250(r250&& other)	Move constructor
r250& operator=(const r250& other)	Copy assignment operator
r250& operator=(r250&& other)	Move assignment operator

Constructors

```
r250::r250(sycl::queue queue, std::uint32_t seed = default_seed)
```

Input Parameters

queue

Valid sycl::queue object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed

The initial conditions of the generator state, assume $x_{-250} = seed$. If $seed = 0$, assume $seed = 1$. Other values in state are initialized according to recurrent correlation $x_{n+1} = 69069 x_n (mod\ 2^{32})$. Then the values $x_{7k-247}, k = 0, 1, ..., 31$ are interpreted as a binary matrix of size 32 x 32 and diagonal bits are set to 0, the under-diagonal bits to 0.

```
r250::r250(sycl::queue queue, std::vector<std::uint32_t> seed)
```

Input Parameters

queue

Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed

The initial conditions of the generator state if $n \geqslant 0 : x_{k-250} = seed[k], k = 0, 1, ..., 249$

```
r250::r250(const r250& other)
```

Input Parameters

other
Valid `r250` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
r250::r250(r250&& other)
```

Input Parameters

other
Valid `r250` object. The `queue` and state of the other engine is moved to the current engine.

```
r250::r250& operator=(const r250& other)
```

Input Parameters

other
Valid `r250` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
r250::r250& operator=(r250&& other)
```

Input Parameters

other
Valid `r250` r-value object. The `queue` and state of the other engine is moved to the current engine.

Parent topic: *Host Engines (Basic Random Number Generators)*

wichmann_hill

The wichmann_hill engine is the set of 273 Wichmann-Hill's combined multiplicative congruential generators from NAG Numerical Libraries *[NAG]*.

Description

The set of 372 different basic pseudorandom number generators wichmann_hill is the second basic generator in the NAG libraries.

Generation algorithm

$x_n = a_{1,j} x_{n-1} (mod\ m_{1,j})$

$y_n = a_{2,j} y_{n-1} (mod\ m_{2,j})$

$z_n = a_{3,j} z_{n-1} (mod\ m_{3,j})$

$w_n = a_{4,j} w_{n-1} (mod\ m_{4,j})$

$u_n = (x_n/m_{1,j} + y_n/m_{2,j} + z_n/m_{3,j} + w_n/m_{4,j}) mod\ 1$

The constants $a_{i,j}$ range from 112 to 127, the constants $m_{i,j}$ are prime numbers ranging from 16718909 to 16776917, close to 2^{24}.

class wichmann_hill

Syntax

```
namespace oneapi::mkl::rng {
class wichmann_hill {
public:
    static constexpr std::uint32_t default_seed = 1;

    wichmann_hill(sycl::queue queue, std::uint32_t seed = default_seed);

    wichmann_hill(sycl::queue queue, std::uint32_t seed, std::uint32_t engine_idx);

    wichmann_hill(sycl::queue queue, std::initializer_list<std::uint32_t> seed);

    wichmann_hill(sycl::queue queue, std::initializer_list<std::uint32_t> seed,␣
↪std::uint32_t engine_idx);

    wichmann_hill(const wichmann_hill& other);

    wichmann_hill(wichmann_hill&& other);

    wichmann_hill& operator=(const wichmann_hill& other);

    wichmann_hill& operator=(wichmann_hill&& other);

    ~wichmann_hill();
};
}
```

Class Members

Routine	Description
wichmann_hill(sycl::queue queue, std::uint32_t seed = default_seed)	Constructor for common seed initialization of the engine (for this case multiple generators of the set would be used)
wichmann_hill(sycl::queue queue, std::uint32_t seed, std::uint32_t engine_idx)	Constructor for common seed initialization of the engine (for this case single generator of the set would be used)
wichmann_hill(sycl::queue& queue, std::initializer_list<std::uint32_t> seed)	Constructor for extended seed initialization of the engine (for this case multiple generators of the set would be used)
wichmann_hill(sycl::queue& queue, std::initializer_list<std::uint32_t> seed, std::uint32_t engine_idx)	Constructor for extended seed initialization of the engine (for this case single generator of the set would be used)
wichmann_hill(const wichmann_hill& other)	Copy constructor
wichmann_hill(wichmann_hill&& other)	Move constructor
wichmann_hill& operator=(const wichmann_hill& other)	Copy assignment operator
wichmann_hill& operator=(wichmann_hill&& other)	Move assignment operator

Constructors

```
wichmann_hill::wichmann_hill(sycl::queue queue, std::uint32_t seed = default_seed)
```

Input Parameters

queue
Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed
The initial conditions of the generator state. Assume $x_0 = seed \, mod \, m_1, y_0 = z_0 = w_0 = 1$. If $x_0 = 0$, assume $x_0 = 1$.

```
wichmann_hill::wichmann_hill(sycl::queue queue, std::uint32_t seed, std::uint32_t engine_
↪idx)
```

Input Parameters

queue
Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed
The initial conditions of the generator state. Assume $x_0 = seed \, mod \, m_1, y_0 = z_0 = w_0 = 1$. If $x_0 = 0$, assume $x_0 = 1$.

engine_idx
The index of the set $1, \ldots, 273$.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $idx > 273$

```
wichmann_hill::wichmann_hill(sycl::queue& queue, std::initializer_list<std::uint32_t>␣
↪seed)
```

Input Parameters

queue
Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed
The initial conditions of the generator state, assume: if $n = 0 : x_0 = y_0 = z_0 = w_0 = 1$

if $n = 1 : x_0 = seed[0]\ mod\ m_1, y_0 = z_0 = w_0 = 1$. If $x_0 = 0$, assume $x_0 = 1$.

if $n = 2 : x_0 = seed[0]\ mod\ m_1, y_0 = seed[1]\ mod\ m_2, z_0 = w_0 = 1$.

if $n = 3 : x_0 = seed[0]\ mod\ m_1, y_0 = seed[1]\ mod\ m_2, z_0 = seed[2]\ mod\ m_3, w_0 = 1$.

if $n \geqslant 4 : x_0 = seed[0]\ mod\ m_1, y_0 = seed[1]\ mod\ m_2$

$z_0 = seed[2]\ mod\ m_3, w_0 = seed[3]\ mod\ m_4$.

```
wichmann_hill::wichmann_hill(sycl::queue& queue, std::initializer_list<std::uint32_t>␣
↪seed, std::uint32_t engine_idx)
```

Input Parameters

queue
Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed
The initial conditions of the generator state, assume: if $n = 0 : x_0 = y_0 = z_0 = w_0 = 1$

if $n = 1 : x_0 = seed[0]\ mod\ m_1, y_0 = z_0 = w_0 = 1$. If $x_0 = 0$, assume $x_0 = 1$.

if $n = 2 : x_0 = seed[0]\ mod\ m_1, y_0 = seed[1]\ mod\ m_2, z_0 = w_0 = 1$.

if $n = 3 : x_0 = seed[0]\ mod\ m_1, y_0 = seed[1]\ mod\ m_2, z_0 = seed[2]\ mod\ m_3, w_0 = 1$.

if $n \geqslant 4 : x_0 = seed[0]\ mod\ m_1, y_0 = seed[1]\ mod\ m_2$

$z_0 = seed[2]\ mod\ m_3, w_0 = seed[3]\ mod\ m_4$.

engine_idx
The index of the set $1, \ldots, 273$.

```
wichmann_hill::wichmann_hill(const wichmann_hill& other)
```

Input Parameters

other
Valid `wichmann_hill` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
wichmann_hill::wichmann_hill(wichmann_hill&& other)
```

Input Parameters

other
Valid `wichmann_hill` object. The `queue` and state of the other engine is moved to the current engine.

```
wichmann_hill::wichmann_hill& operator=(const wichmann_hill& other)
```

Input Parameters

other
Valid `wichmann_hill` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
wichmann_hill::wichmann_hill& operator=(wichmann_hill&& other)
```

Input Parameters

other
Valid `wichmann_hill` r-value object. The `queue` and state of the other engine is moved to the current engine.

Parent topic: *Host Engines (Basic Random Number Generators)*

mt19937

Mersenne Twister pseudorandom number generator.

Description

The Mersenne Twister pseudorandom number generator, mt19937, is a modification of twisted generalized feedback shift register generator [*Matsumoto98*]. MT19937 has the period length of $2^{19937}-1$ and is 623-dimensionally equidistributed with up to 32-bit accuracy. These properties make the generator applicable for simulations in various fields of science and engineering. The state of the generator is represented by 624 32-bit unsigned integer numbers.

Generation algorithm

$x_n = x_{n-(624-397)} \oplus ((x_{n-624} \& 0x80000000) | (x_{n-624+1} \& 0x7FFFFFFF))A$

$y_n = x_n$

$y_n = y_n \oplus (y_n >> 11)$

$y_n = y_n \oplus ((y_n << 7) \& 0x9D2C5680)$

$y_n = y_n \oplus ((y_n << 15) \& 0xEFC60000)$

$y_n = y_n \oplus (y_n >> 18)$

$u_n = y_n / 2^{32}$

Matrix $A_j(32x32)$ has the following format:

$$\begin{bmatrix} 0 & 1 & 0 & ... & ... \\ 0 & 0 & ... & 0 & ... \\ ... & ... & ... & ... & ... \\ 0 & ... & 0 & 0 & 1 \\ a_{31} & a_{30} & ... & ... & a_0 \end{bmatrix}$$

Where the 32-bit vector $a = a_{31}..a_0$ has the value $a = 0x9908B0DF$.

class mt19937

Syntax

```
namespace oneapi::mkl::rng {
class mt19937 {
public:
    static constexpr std::uint32_t default_seed = 1;

    mt19937(sycl::queue queue, std::uint32_t seed = default_seed);

    mt19937(sycl::queue queue, std::initializer_list<std::uint32_t> seed);

    mt19937(const mt19937& other);

    mt19937(mt19937&& other);

    mt19937& operator=(const mt19937& other);

    mt19937& operator=(mt19937&& other);

    ~mt19937();
};
}
```

Class Members

Routine	Description
mt19937(sycl::queue queue, std::uint32_t seed = default_seed)	Constructor for common seed initialization of the engine
mt19937(sycl::queue queue, std::initializer_list<std::uint32_t> seed)	Constructor for extended seed initialization of the engine
mt19937(const mt19937& other)	Copy constructor
mt19937(mt19937&& other)	Move constructor
mt19937& operator=(const mt19937& other)	Copy assignment operator
mt19937& operator=(mt19937&& other)	Move assignment operator

Constructors

```
mt19937::mt19937(sycl::queue queue, std::uint32_t seed = default_seed)
```

Input Parameters

queue
Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed
The initial conditions of the generator state. The initialization algorithm described in [*MT2203*].

```
mt19937::mt19937(sycl::queue queue, std::initializer_list<std::uint32_t> seed)
```

Input Parameters

queue
Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed
The initial conditions of the generator state. The initialization algorithm described in [*MT2203*].

```
mt19937::mt19937(const mt19937& other)
```

Input Parameters

other
Valid `mt19937` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
mt19937::mt19937(mt19937&& other)
```

Input Parameters

other
Valid `mt19937` object. The queue and state of the other engine is moved to the current engine.

```
mt19937::mt19937& operator=(const mt19937& other)
```

Input Parameters

other
Valid `mt19937` object. The queue and state of the other engine is copied and applied to the current engine.

```
mt19937::mt19937& operator=(mt19937&& other)
```

Input Parameters

other
Valid `mt19937` r-value object. The queue and state of the other engine is moved to the current engine.

Parent topic: *Host Engines (Basic Random Number Generators)*

sfmt19937

The SIMD-oriented Mersenne Twister pseudorandom number generator.

Description

SIMD-oriented Fast Mersenne Twister pseudorandom number generator SFMT19937 [*Saito08*] with a period length equal to $2^{19937} - 1$ of the produced sequence. The state of the engine contains the array of 156 128-bit integers.

Generation algorithm

$w_n = w_0 A \oplus w_M B \oplus w_{n-2} C \oplus w_{n-1} D$

Where $w_0, w_M, w_{n-2}, ...$ are the 128-bit integers, and wA, wB, wC, wD operations are defined as follows:

$wA = (w << 8) \oplus w$, left shift of 128-bit integer w by a followed by exclusive-or operation

$wB = (w >> 8) \& mask$, right shift of each 32-bit integer in quadruple w by and-operator with quadruple of 32-bit masks $mask = (0xBFFFFFF6, 0xDFFAFFFF, 0xDDFECB7F, 0xDFFFFFEF)$

$wC = (w >> 8) \oplus w$, right shift of 128-bit integer w

$wD = (w << 8)$, left shift of each 32-bit integer in quadruple w

Integer output: $r_{4n+k} = w_n(k)$, where $w_n(k)$ is the k-th 32-bit integer in quadruple $w_n, k = 0, 1, 2, 3$

$u_n = (int) r_n / 2^{32} + 1/2$

class sfmt19937

Syntax

```
namespace oneapi::mkl::rng {
class sfmt19937 {
public:
    static constexpr std::uint32_t default_seed = 1;

    sfmt19937(sycl::queue queue, std::uint32_t seed = default_seed);

    sfmt19937(sycl::queue queue, std::initializer_list<std::uint32_t> seed);

    sfmt19937(const sfmt19937& other);

    sfmt19937(sfmt19937&& other);

    sfmt19937& operator=(const sfmt19937& other);

    sfmt19937& operator=(sfmt19937&& other);

    ~sfmt19937();
};
}
```

Class Members

Routine	Description
sfmt19937(sycl::queue queue, std::uint32_t seed = default_seed)	Constructor for common seed initialization of the engine
sfmt19937(sycl::queue queue, std::initializer_list<std::uint32_t> seed)	Constructor for extended seed initialization of the engine
sfmt19937(const sfmt19937& other)	Copy constructor
sfmt19937(sfmt19937&& other)	Move constructor
sfmt19937& operator=(const sfmt19937& other)	Copy assignment operator
sfmt19937& operator=(sfmt19937&& other)	Move assignment operator

Constructors

```
sfmt19937::sfmt19937(sycl::queue queue, std::uint32_t seed = default_seed)
```

Input Parameters

queue
Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed
The initial conditions of the generator state. The initialization algorithm described in [*Saito08*].

```
sfmt19937::sfmt19937(sycl::queue queue, std::initializer_list<std::uint32_t> seed)
```

Input Parameters

queue
Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed
The initial conditions of the generator state. The initialization algorithm described in [*Saito08*].

```
sfmt19937::sfmt19937(const sfmt19937& other)
```

Input Parameters

other
Valid `sfmt19937` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
sfmt19937::sfmt19937(sfmt19937&& other)
```

Input Parameters

other
Valid `sfmt19937` object. The `queue` and state of the other engine is moved to the current engine.

```
sfmt19937::sfmt19937& operator=(const sfmt19937& other)
```

Input Parameters

other
Valid `sfmt19937` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
sfmt19937::sfmt19937& operator=(sfmt19937&& other)
```

Input Parameters

other
Valid `sfmt19937` r-value object. The `queue` and state of the other engine is moved to the current engine.

Parent topic: *Host Engines (Basic Random Number Generators)*

mt2203

The mt2203 engine is the set of 6024 Mersenne Twister pseudorandom number generators MT2203 [*Matsumoto98*], [*Matsumoto00*].

Description

The set of 6024 basic pseudorandom number generators MT2203 is a natural addition to the MT19937 generator. MT2203 generators are intended for use in large scale Monte Carlo simulations performed on multi-processor computer systems.

Generation algorithm

For $j = 1, ..., 6024$:

$x_{n,j} = x_{n-(69-34),j} \oplus ((x_{n-69,j}\&0xFFFFFFFE0)|(x_{n+69+1,j}\&0x1F))A_j$

$y_{n,j} = x_{n,j}$

$y_{n,j} = y_{n,j} \oplus (y_{n,j} >> 12)$

$y_{n,j} = y_{n,j} \oplus ((y_{n,j} << 7)\&b_j)$

$y_{n,j} = y_{n,j} \oplus ((y_{n,j} << 15)\&c_j)$

$y_{n,j} = y_{n,j} \oplus (y_{n,j} >> 18)$

Matrix $A_j(32x32)$ has the following format:

$$\begin{bmatrix} 0 & 1 & 0 & ... & ... \\ 0 & 0 & ... & 0 & ... \\ ... & ... & ... & ... & ... \\ 0 & ... & 0 & 0 & 1 \\ a_{31,j} & a_{30,j} & ... & ... & a_{0,1} \end{bmatrix}$$

class mt2203

Syntax

```
namespace oneapi::mkl::rng {
class mt2203 {
public:
    static constexpr std::uint32_t default_seed = 1;

    mt2203(sycl::queue queue, std::uint32_t seed = default_seed);
```

(continues on next page)

(continued from previous page)

```
    mt2203(sycl::queue queue, std::uint32_t seed, std::uint32_t engine_idx);

    mt2203(sycl::queue queue, std::initializer_list<std::uint32_t> seed);

    mt2203(sycl::queue queue, std::initializer_list<std::uint32_t> seed, std::uint32_t␣
↪engine_idx);

    mt2203(const mt2203& other);

    mt2203(mt2203&& other);

    mt2203& operator=(const mt2203& other);

    mt2203& operator=(mt2203&& other);

    ~mt2203();
};
}
```

Class Members

Routine	Description
mt2203(sycl::queue queue, std::uint32_t seed = default_seed)	Constructor for common seed initialization of the engine (for this case multiple generators of the set would be used)
mt2203(sycl::queue queue, std::uint32_t seed, std::uint32_t engine_idx)	Constructor for common seed initialization of the engine (for this case single generator of the set would be used)
mt2203(sycl::queue queue, std::initializer_list<std::uint32_t> seed)	Constructor for extended seed initialization of the engine (for this case multiple generators of the set would be used)
mt2203(sycl::queue queue, std::initializer_list<std::uint32_t> seed, std::uint32_t engine_idx)	Constructor for extended seed initialization of the engine (for this case single generator of the set would be used)
mt2203(const mt2203& other)	Copy constructor
mt2203(mt2203&& other)	Move constructor
mt2203& operator=(const mt2203& other)	Copy assignment operator
mt2203& operator=(mt2203&& other)	Move assignment operator

Constructors

```
mt2203::mt2203(sycl::queue queue, std::uint32_t seed = default_seed)
```

Input Parameters

queue
Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed
The initial conditions of the generator state. The initialization algorithm described in [*MT2203*].

```
mt2203::mt2203(sycl::queue queue, std::uint32_t seed, std::uint32_t engine_idx)
```

Input Parameters

queue
Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed
The initial conditions of the generator state. The initialization algorithm described in [*MT2203*].

engine_idx
The index of the set 1, . . . , 6024.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $idx > 6024$

```
mt2203::mt2203(sycl::queue queue, std::initializer_list<std::uint32_t> seed)
```

Input Parameters

queue
Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed
The initial conditions of the generator state. The initialization algorithm described in [*MT2203*].

```
mt2203::mt2203(sycl::queue queue, std::initializer_list<std::uint32_t> seed, std::uint32_
↪t engine_idx)
```

Input Parameters

queue
Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed
The initial conditions of the generator state. The initialization algorithm described in [*MT2203*].

engine_idx
The index of the set 1, . . . , 6024.

```
mt2203::mt2203(const mt2203& other)
```

Input Parameters

other
Valid `mt2203` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
mt2203::mt2203(mt2203&& other)
```

Input Parameters

other
Valid `mt2203` object. The `queue` and state of the other engine is moved to the current engine.

```
mt2203::mt2203& operator=(const mt2203& other)
```

Input Parameters

other
Valid `mt2203` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
mt2203::mt2203& operator=(mt2203&& other)
```

Input Parameters

other
Valid `mt2203` r-value object. The `queue` and state of the other engine is moved to the current engine.

Parent topic: *Host Engines (Basic Random Number Generators)*

ars5

The ars5 counter-based pseudorandom number generator.

Description

The ars5 engine is a keyed family of counter-based BRNG. The state consists of a 128-bit integer counter c and a 128-bit key k. The BRNG is based on the AES encryption algorithm [*FIPS-197*].

Generation algorithm

The generator has a 32-bit integer output obtained in the following way [*Salmon11*]:

1. The i-th number is defined by the following formula $r_i = (f(i/4) >> ((i \ mod \ 4) * 32)) \ mod \ 2^{32}$
2. **Function $f(c)$ takes a 128-bit argument and returns a 128-bit number. The returned number is obtained as follows:**
   2.1. $c_0 = c \oplus k$ and $k_0 = k$.

   2.2. The following recurrence is calculated N = 5 times:

   $c_{i+1} = SubBytes(c)$

   $c_{i+1} = ShiftRows(c_{i+1})$

   $c_{i+1} = MixColumns(c_{i+1})$, this step is omitted if $i + 1 = N$

   $c_{i+1} = AddRoundKey(c_{i+1}, k_j)$

   $Lo(k_{i+1}) = Lo(k) + 0x9E3779B97F4A7C15$

   $Hi(k_{i+1}) = Hi(k) + 0xBB67AE8584CAA73B$

   Specification for $SubBytes, ShiftRows, MixColumns, AddRoundKey$ functions can be found in [*FIPS-197*].

   2.3. Put $f(c) = c_N$, where $N = 10$
3. Real output: $u_n = (int)r_n/2^{32} + 1/2$

class ars5

Syntax

```
namespace oneapi::mkl::rng {
class ars5 {
public:
        static constexpr std::uint64_t default_seed = 0;

        ars5(sycl::queue queue, std::uint64_t seed = default_seed);

        ars5(sycl::queue queue, std::initializer_list<std::uint64_t> seed);

        ars5(const ars5& other);

        ars5(ars5&& other);
```

(continues on next page)

(continued from previous page)

```
        ars5& operator=(const ars5& other);

        ars5& operator=(ars5&& other);

        ~ars5();
};
}
```

Class Members

Routine	Description
ars5(sycl::queue queue, std::uint64_t seed)	Constructor for common seed initialization of the engine
ars5(sycl::queue queue, std::initializer_list<std::uint64_t> seed)	Constructor for extended seed initialization of the engine
ars5(const ars5& other)	Copy constructor
ars5(ars5&& other)	Move constructor
ars5& operator=(const ars5& other)	Copy assignment operator
ars5& operator=(ars5&& other)	Move assignment operator

Constructors

```
ars5::ars5(sycl::queue queue, std::uint64_t seed)
```

Input Parameters

queue
Valid sycl::queue object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed
The initial conditions of the generator state, assume $k = seed, c = 0$, where k is 128-bit key, c is 128-bit counter.

```
ars5::ars5(sycl::queue queue, std::initializer_list<std::uint64_t> seed)
```

Input Parameters

queue
Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed
The initial conditions of the generator state, assume if $n = 0 : k = 0, c = 0$

if $n = 1 : k = seed[0], c = 0$

if $n = 2 : k = seed[0] + seed[1] \cdot 2^{64}, c = 0$

if $n = 3 : k = seed[0] + seed[1] \cdot 2^{64}, c = seed[2]$

if $n = 4 : k = seed[0] + seed[1] \cdot 2^{64}, c = seed[2] + seed[3] \cdot 2^{64}$

for $n > 4$ following arguments are ignored

```
ars5::ars5(const ars5& other)
```

Input Parameters

other
Valid `ars5` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
ars5::ars5(ars5&& other)
```

Input Parameters

other
Valid `ars5` r-value object. The `queue` and state of the other engine is moved to the current engine.

```
ars5::ars5& operator=(const ars5& other)
```

Input Parameters

other
Valid `ars5` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
ars5::ars5& operator=(ars5&& other)
```

Input Parameters

other
Valid `ars5` r-value object. The `queue` and state of the other engine is moved to the current engine.

Parent topic: *Host Engines (Basic Random Number Generators)*

sobol

The sobol is a 32-bit Gray code-based quasi-random number generator.

Description

Bratley and Fox *[Bratley88]* provide an implementation of the SOBOL quasi-random number generator. The default dimensions of quasi-random vectors can vary from 1 to 40 inclusive. It is also allowed to register user-defined parameters (direction numbers).

Generation algorithm

$x_n = x_{n_1} \oplus v_c$

$u_n = x_n / 2^{32}$

The value c is the right-most zero bit in $n-1$; x_n is s-dimensional vector of 32-bit values. The s-dimensional vectors (calculated during engine initialization) $v_i, i = 1, 32$ are called direction numbers. The vector u_n is the generator output normalized to the unit hypercube $(0, 1)^s$.

class sobol

Syntax

```
namespace oneapi::mkl::rng {
class sobol {
public:
    static constexpr std::uint32_t default_dimensions_number = 1;

    sobol(sycl::queue queue, std::uint32_t dimensions = default_dimensions_number);

    sobol(sycl::queue queue, std::vector<std::uint32_t>& direction_numbers);

    sobol(const sobol& other);

    sobol(sobol&& other);

    sobol& operator=(const sobol& other);

    sobol& operator=(sobol&& other);

    ~sobol();
};
}
```

Class Members

Routine	Description
sobol(sycl::queue queue, std::uint32_t dimensions = default_dimensions_number)	Constructor with specified number of dimensions. The value should be 1..40.
sobol(sycl::queue queue, std::vector<std::uint32_t>& direction_numbers)	Constructor for extended use-case, when it's needed to use the number of dimensions greater than 40 or obtain another sequence.
sobol(const sobol& other)	Copy constructor
sobol(sobol&& other)	Move constructor
sobol& operator=(const sobol& other)	Copy assignment operator
sobol& operator=(sobol&& other)	Move assignment operator

Constructors

```
sobol::sobol(sycl::queue queue, std::uint32_t dimensions = default_dimensions_number)
```

Input Parameters

queue
Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

dimensions
Number of dimensions. If $dimen < 1$ or $dimen > 40$, assume $dimen = 1$.

```
sobol::sobol(sycl::queue queue, std::vector<std::uint32_t>& direction_numbers)
```

Input Parameters

queue
Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

direction_numbers
If you want to generate quasi-random vectors of greater dimension or obtain another sequence, you can register a set of your own direction_numbers. The number of dimensions corresponds to direction_numbers.size() / 32.

```
sobol::sobol(const sobol& other)
```

Input Parameters

other
Valid `sobol` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
sobol::sobol(sobol&& other)
```

Input Parameters

other
Valid `sobol` object. The `queue` and state of the other engine is moved to the current engine.

```
sobol::sobol& operator=(const sobol& other)
```

Input Parameters

other
Valid `sobol` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
sobol::sobol& operator=(sobol&& other)
```

Input Parameters

other
Valid `sobol` r-value object. The `queue` and state of the other engine is moved to the current engine.

Parent topic: *Host Engines (Basic Random Number Generators)*

niederreiter

The niederreiter generator is a 32-bit Gray code-based quasi-random number generator.

Description

According to results of Bratley, Fox and Niederreiter *[Bratley92]* Niederreiter sequences have the best known theoretical asymptotic properties. The default dimension of quasi-random vectors can vary from 1 to 318 inclusive. It is also allowed to register user-defined parameters (irreducible polynomials).

Generation algorithm

$x_n = x_{n_1} \oplus v_c$

$u_n = x_n / 2^{32}$

The value c is the right-most zero bit in $n-1$; x_n is s-dimensional vector of 32-bit values. The s-dimensional vectors (calculated during engine initialization) $v_i, i = 1, 32$ are called direction numbers. The vector u_n is the generator output normalized to the unit hypercube $(0, 1)^s$.

class niederreiter

Syntax

```
namespace oneapi::mkl::rng {
class niederreiter {
public:
    static constexpr std::uint32_t default_dimensions_number = 1;

    niederreiter(sycl::queue queue, std::uint32_t dimensions = default_dimensions_
↪number);
```

(continues on next page)

(continued from previous page)

```
    niederreiter(sycl::queue queue, std::vector<std::uint32_t>& irred_polynomials);

    niederreiter(const niederreiter& other);

    niederreiter(niederreiter&& other);

    niederreiter& operator=(const niederreiter& other);

    niederreiter& operator=(niederreiter&& other);

    ~niederreiter();
};
}
```

Class Members

Routine	Description
niederreiter(sycl::queue queue, std::uint32_t dimensions = default_dimensions_number)	Constructor with specified number of dimensions. The value should be 1..318.
niederreiter(sycl::queue queue, std::vector<std::uint32_t>& irred_polynomials)	Constructor for extended use-case, when it's needed to use the number of dimensions greater than 318 or obtain another sequence.
niederreiter(const niederreiter& other)	Copy constructor
niederreiter(niederreiter&& other)	Move constructor
niederreiter& operator=(const niederreiter& other)	Copy assignment operator
niederreiter& operator=(niederreiter&& other)	Move assignment operator

Constructors

```
niederreiter::niederreiter(sycl::queue queue, std::uint32_t dimensions = default_
↪dimensions_number)
```

Input Parameters

queue
: Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

dimensions
: Number of dimensions. If $dimen < 1$ or $dimen > 318$, assume $dimen = 1$.

```
niederreiter::niederreiter(sycl::queue queue, std::vector<std::uint32_t>& irred_
↪polynomials)
```

Input Parameters

queue
Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

irred_polynomials
If you want to generate quasi-random vectors of greater dimension or obtain another sequence, you can register a set of your own irreducible polynomials. The number of dimensions corresponds to the length of the vector.

```
niederreiter::niederreiter(const niederreiter& other)
```

Input Parameters

other
Valid `niederreiter` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
niederreiter::niederreiter(niederreiter&& other)
```

Input Parameters

other
Valid `niederreiter` object. The `queue` and state of the other engine is moved to the current engine.

```
niederreiter::niederreiter& operator=(const niederreiter& other)
```

Input Parameters

other
Valid `niederreiter` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
niederreiter::niederreiter& operator=(niederreiter&& other)
```

Input Parameters

other
Valid `niederreiter` r-value object. The `queue` and state of the other engine is moved to the current engine.

Parent topic: *Host Engines (Basic Random Number Generators)*

nondeterministic

Non-deterministic random number generator.

Description

Implementation defined generator with non-deterministic source of randomness (for example, a hardware device).

class nondeterministic

Syntax

```
namespace oneapi::mkl::rng {
class nondeterministic {
public:
    nondeterministic(sycl::queue queue);

    nondeterministic(const nondeterministic& other);

    nondeterministic(nondeterministic&& other);

    nondeterministic& operator=(const nondeterministic& other);

    nondeterministic& operator=(nondeterministic&& other);

    ~nondeterministic();
};
}
```

Class Members

Routine	Description
nondeterministic(sycl::queue queue)	Constructor for the particular device
nondeterministic(const nondeterministic& other)	Copy constructor
nondeterministic(nondeterministic&& other)	Move constructor
nondeterministic& operator=(const nondeterministic& other)	Copy assignment operator
nondeterministic& operator=(nondeterministic&& other)	Move assignment operator

Constructors

```
nondeterministic::nondeterministic(sycl::queue queue)
```

Input Parameters

queue
Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

```
nondeterministic::nondeterministic(const nondeterministic& other)
```

Input Parameters

other
Valid `nondeterministic` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
nondeterministic::nondeterministic(nondeterministic&& other)
```

Input Parameters

other
Valid `nondeterministic` object. The `queue` and state of the other engine is moved to the current engine.

```
nondeterministic::nondeterministic& operator=(const nondeterministic& other)
```

Input Parameters

other
Valid `nondeterministic` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
nondeterministic::nondeterministic& operator=(nondeterministic&& other)
```

Input Parameters

other
Valid `nondeterministic` r-value object. The `queue` and state of the other engine is moved to the current engine.

Parent topic: *Host Engines (Basic Random Number Generators)*

Host Service Routines

Routine	Description
leapfrog	Proceed state of engine by the leapfrog method to generate a subsequence of the original sequence
skip_ahead	Proceed state of engine by the skip-ahead method to skip a given number of elements from the original sequence

Parent topic: *Random Number Generators Host Routines*

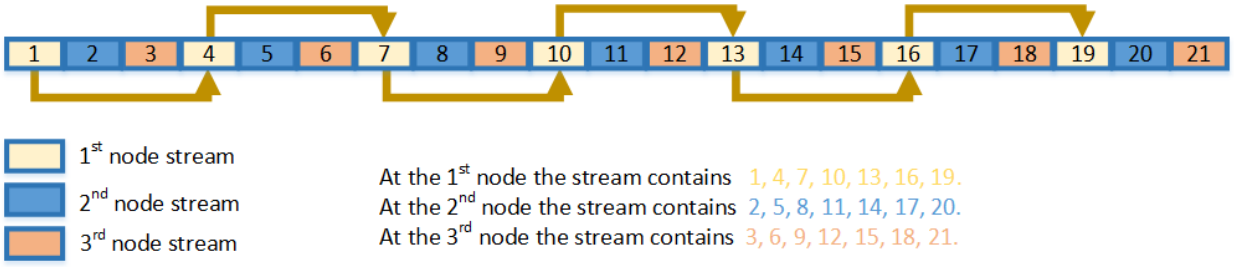
leapfrog

Proceed state of engine by the leapfrog method.

Description and Assumptions

oneapi::mkl::rng::leapfrog function generates random numbers in an engine with non-unit stride. This feature is particularly useful in distributing random numbers from the original stream across the stride buffers without generating the original random sequence with subsequent manual distribution. see *Figure "Leapfrog Method"*.

Leapfrog Method



leapfrog

Syntax

```
namespace oneapi::mkl::rng {
template<typename EngineType>
void leapfrog(EngineType& engine, std::uint64_t idx, std::uint64_t stride);
}
```

Template Parameters

EngineType
Type of engine. Note: may not be supported by all available engine classes.

Input Parameters

engine
Engine which state would be skipped.

idx
Index of the computational node.

stride
Largest number of computational nodes, or stride.

Example

```
// Creating 3 identical engines
oneapi::mkl::rng::mcg31m1 engine_1(queue, seed);

oneapi::mkl::rng::mcg31m1 engine_2(engine_1);
oneapi::mkl::rng::mcg31m1 engine_3(engine_1);


// Leapfrogging the states of engines
oneapi::mkl::rng::leapfrog(engine_1, 0 , 3);
oneapi::mkl::rng::leapfrog(engine_2, 1 , 3);
oneapi::mkl::rng::leapfrog(engine_3, 2 , 3);
// Generating random numbers
```

Parent topic: *Host Service Routines*

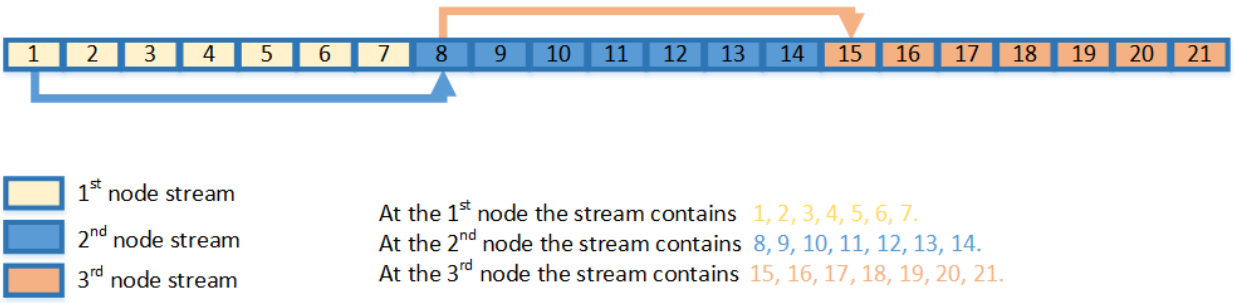
skip_ahead

Proceed state of engine by the skip-ahead method.

Description and Assumptions

oneapi::mkl::rng::skip_ahead function changes the current state of the engine so that with the further call of the generator the output subsequence begins with the specified offset see *Figure "Block-Splitting Method"*.

Block-Splitting Method



skip_ahead

Syntax

```
namespace oneapi::mkl::rng {
template<typename EngineType>
void skip_ahead(EngineType& engine, std::uint64_t num_to_skip);
}
```

Template Parameters

EngineType
Type of engine. Note: may not be supported by all available engine classes.

Input Parameters

engine
Engine which state would be skipped.

num_to_skip
Number of elements to skip in the engine's sequence.

Example

```
// Creating 3 identical engines
oneapi::mkl::rng::mcg31m1 engine_1(queue, seed);
oneapi::mkl::rng::mcg31m1 engine_2(engine_1);
oneapi::mkl::rng::mcg31m1 engine_3(engine_2);

// Skipping ahead by 7 elements the 2nd engine
oneapi::mkl::rng::skip_ahead(engine_2, 7);

// Skipping ahead by 14 elements the 3rd engine
oneapi::mkl::rng::skip_ahead(engine_3, 14);
```

skip_ahead (Interface with a partitioned number of skipped elements)

Syntax

```
namespace oneapi::mkl::rng {
template<typename EngineType>
void oneapi::mkl::rng::skip_ahead(EngineType& engine, std::initializer_list<std::uint64_
↪t> num_to_skip);
}
```

Template Parameters

EngineType
Type of engine. Note: may not be supported by all available engine classes.

Input Parameters

engine
Engine which state would be skipped.

num_to_skip
Partitioned number of elements to skip in the engine's sequence. The total number of skipped elements would be: $num\_to\_skip[0] + num\_to\_skip[1] \cdot 2^{64} + ... + num\_to\_skip[1] \cdot 2^{64(n-1)}$, where n is a number of elements in num_to_skip list.

Example with Partitioned Numer of Elements

```
// Creating the first engine
oneapi::mkl::rng::mrg32k3a engine_1(queue, seed);

// To skip 2^64 elements in the random stream number of skipped elements should be
// represented as num_to_skip = 2^64 = 0 + 1 * 2^64
std::initializer_list<std::uint64_t> num_to_skip = {0, 1};

// Creating the 2nd engine based on 1st. Skipping by 2^64
oneapi::mkl::rng::mrg32k3a engine_2(engine_1);
oneapi::mkl::rng::skip_ahead(engine_2, num_to_skip);
```

Parent topic: *Host Service Routines*

Host Distributions

oneMKL RNG routines are used to generate random numbers with different types of distribution. Each function group is introduced below by the type of underlying distribution and contains a short description of its functionality, as well as specifications of the call sequence and the explanation of input and output parameters. *Table Continuous Distribution Generators* and *Table Discrete Distribution Generators* list the random number generator routines with data types and output distributions, and sets correspondence between data types of the generator routines and the basic random number generators.

Table Continuous Distribution Generators

Routine	Description
uniform (continuous)	Uniform continuous distribution on the interval [a,b)
gaussian	Normal (Gaussian) distribution
exponential	Exponential distribution
laplace	Laplace distribution (double exponential distribution)
weibull	Weibull distribution
cauchy	Cauchy distribution
rayleigh	Rayleigh distribution
lognormal	Lognormal distribution
gumbel	Gumbel (extreme value) distribution
gamma	Gamma distribution
beta	Beta distribution
chi_square	Chi-Square distribution
gaussian_mv	Normal Multivariate (Gaussian Multivariate) distribution

Table Discrete Distribution Generators

Type of Distribution	Description
uniform (discrete)	Uniform discrete distribution on the interval [a,b)
uniform_bits	Uniformly distributed bits in 32/64-bit chunks
bits	Bits of underlying BRNG integer recurrence
bernoulli	Bernoulli distribution
geometric	Geometric distribution
binomial	Binomial distribution
hypergeometric	Hypergeometric distribution
poisson	Poisson distribution
poisson_v	Poisson distribution with varying mean
negative_binomial	Negative binomial distribution, or Pascal distribution
multinomial	Multinomial distribution

Modes of random number generation

The library provides two modes of random number generation, accurate and fast. Accurate generation mode is intended for applications that are highly demanding to accuracy of calculations. When used in this mode, the generators produce random numbers lying completely within the definitional domain for all values of the distribution parameters. For example, random numbers obtained from the generator of continuous distribution that is uniform on interval [a,b] belong to this interval irrespective of what a and b values may be. Fast mode provides high performance generation and also guarantees that generated random numbers belong to the definitional domain except for some specific values of distribution parameters. The generation mode is set by specifying the relevant value of the method parameter in generator routines. The list of distributions that support accurate mode of generation is given in the table below.

Table Distribution Generators with Accurate Method

Distribution	Method
uniform (continuous)	*oneapi::mkl::rng::unform_method::accurate*
exponential	*oneapi::mkl::rng::exponential_method::icdf_accurate*
weibull	*oneapi::mkl::rng::weibull_method::icdf_accurate*
rayleigh	*oneapi::mkl::rng::rayleigh_method::icdf_accurate*
lognormal	*oneapi::mkl::rng::lognormal_method::box_muller2_accurate*, *oneapi::mkl::rng::lognormal_method::icdf_accurate*
gamma	*oneapi::mkl::rng::gamma_method::marsaglia_accurate*
beta	*oneapi::mkl::rng::beta_method::cja_accurate*

Parent topic: *Random Number Generators Host Routines*

Distributions Template Parameter Method

Method	Distributions	Math Description
uniform_metho uniform_metho	unifo d) unifo	Standard method. `uniform_method::standard_accurate` supported for `uniform(s, d)` only.
gaussian_metl	gauss	Generates normally distributed random number x thru the pair of uniformly distributed numbers u_1 and u_2 according to the formula: $x = \sqrt{-2lnu_1}\sin(2\pi u_2)$
gaussian_metl	gauss logno	Generates normally distributed random numbers x_1 and x_2 thru the pair of uniformly distributed numbers u_1 and u_2 according to the formulas: $x_1 = \sqrt{-2lnu_1}\sin 2\pi u_2$ $x_2 = \sqrt{-2lnu_1}\cos 2\pi u_2$
gaussian_metl	gauss	Inverse cumulative distribution function (ICDF) method.
exponential_r exponential_r	expon	Inverse cumulative distribution function (ICDF) method.
weibull_metho weibull_metho	weibu	Inverse cumulative distribution function (ICDF) method.
cauchy_metho	cauch	Inverse cumulative distribution function (ICDF) method.
rayleigh_metl rayleigh_metl	rayle	Inverse cumulative distribution function (ICDF) method.
bernoulli_me	berno	Inverse cumulative distribution function (ICDF) method.
geometric_me	geome	Inverse cumulative distribution function (ICDF) method.
gumbel_metho	gumbe	Inverse cumulative distribution function (ICDF) method.
lognormal_me lognormal_me	logno	Inverse cumulative distribution function (ICDF) method.
lognormal_me lognormal_me	logno	Generated normally distributed random numbers x_1 and x_2 by box_muller2 method are converted to lognormal distribution.
gamma_method gamma_method	gamma	For $\alpha > 1$, a gamma distributed random number is generated as a cube of properly scaled normal random number; for $0.6 \leq \alpha < 1$, a gamma distributed random number is generated using rejection from Weibull distribution; for $\alpha < 0.6$, a gamma distributed random number is obtained using transformation of exponential power distribution; for $\alpha = 1$, gamma distribution is reduced to exponential distribution.
beta_method: beta_method:	beta	Cheng-Jonhnk-Atkinson method. For $min(p,q) > 1$, Cheng method is used; for $min(p,q) < 1$, Johnk method is used, if $q + K * p2 + C \leq 0(K = 0.852..., C = -0.956...)$ otherwise, Atkinson switching algorithm is used; for $max(p,q) < 1$, method of Johnk is used; for $min(p,q) < 1, max(p,q) > 1$, Atkinson switching algorithm is used (CJA stands for Cheng, Johnk, Atkinson); for $p = 1$ or $q = 1$, inverse cumulative distribution function method is used; for $p = 1$ and $q = 1$, beta distribution is reduced to uniform distribution.
chi_square_m	chi_s	(most common): If $\nu \geq 17$ or ν is odd and $5 \leq \nu \leq 15$, a chi-square distribution is reduced to a Gamma distribution with these parameters: Shape $\alpha = \nu/2$Offset $a = 0$ Scale factor $\beta = 2$ The random numbers of the Gamma distribution are generated.
binomial_metl	binom	Acceptance/rejection method for $ntrial * min(p, 1-p) \geq 30$ with decomposition into four regions: two parallelograms, triangle, left exponential tail, right exponential tail.
poisson_metho	poiss	Acceptance/rejection method for $\lambda \geq 27$ with decomposition into four regions: two parallelograms, triangle, left exponential tail, right exponential tail.
poisson_metho	poiss	for $\lambda \geq 1$, method based on Poisson inverse CDF approximation by Gaussian inverse CDF; for $\lambda < 1$, table lookup method is used.
poisson_v_me	poiss	for $\lambda \geq 1$, method based on Poisson inverse CDF approximation by Gaussian inverse CDF; for $\lambda < 1$, table lookup method is used.
hypergeometr	hyper	Acceptance/rejection method for large mode of distribution with decomposition into three regions: rectangular, left exponential tail, right exponential tail.
negative_bino	negbi	Acceptance/rejection method for: $\frac{(a-1)(1-p)}{p} \geq 100$ with decomposition into five regions: rectangular, 2 trapezoid, left exponential tail, right exponential tail.
multinomial_r	multi	Multinomial distribution with parameters m, k, and a probability vector p. Random numbers of the multinomial distribution are generated by Poisson Approximation method.

Parent topic: *Host Distributions*

uniform (continuous)

Class is used for generation of uniformly distributed real types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers uniformly distributed over the interval $[a, b)$, where a, b are the left and right bounds of the interval, respectively, and $a, b \in R; a < b$

The probability distribution is given by:

$$f_{a,b}(x) = \begin{cases} \frac{1}{b-a}, x \in [a, b) \\ 0, x \notin [a, b) \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,b}(x) = \begin{cases} 0, x < a \\ \frac{x-a}{b-a}, a \leq x < b, x \in R \\ 1, x \geq b \end{cases}$$

class uniform

Syntax

```
namespace oneapi::mkl::rng {
template<typename RealType = float, typename Method = uniform_method::by_default>
class uniform {
public:
    using method_type = Method;
    using result_type = RealType;
    uniform();
    explicit uniform(RealType a, RealType b);
    RealType a() const;
    RealType b() const;
};
}
```

Template parameters

typename RealType

Type of the produced values. Supported types:

- `float`
- `double`

typename Method = oneapi::mkl::rng::uniform_method::by_default

Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::uniform_method::by_default`

- `oneapi::mkl::rng::uniform_method::standard`
- `oneapi::mkl::rng::uniform_method::accurate`

See description of the methods in *Distributions methods template parameter*

Class Members

Routine	Description
uniform()	Default constructor
explicit uniform(RealType a, RealType b)	Constructor with parameters
RealType a() const	Method to obtain left bound a
RealType b() const	Method to obtain right bound b

Member types

```
uniform::method_type = Method
```

Description

The type which defines transformation method for generation.

```
uniform::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
uniform::uniform()
```

Description

Default constructor for distribution, parameters set as $a = 0.0$, $b = 1.0$.

```
explicit uniform::uniform(RealType a, RealType b)
```

Description

Constructor with parameters. a is a left bound, b is a right bound, assume $a < b$.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $a \geq b$

Characteristics

```
RealType uniform::a() const
```

Return Value

Returns the distribution parameter a - left bound.

```
RealType uniform::b() const
```

Return Value

Returns the distribution parameter b - right bound.

Parent topic: *Host Distributions*

gaussian

Class is used for generation of normally distributed real types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers normally distributed with mean $(mean, a)$ and standard deviation $(stddev, \sigma)$, where $a, \sigma \in R; \sigma > 0$.

The probability distribution is given by:

$$f_{a,\sigma}(x) = \frac{1}{\sigma\sqrt{2\pi}} exp(-\frac{(x-a)^2}{2*\sigma^2}), x \in R.$$

The cumulative distribution function is as follows:

$$F_{a,\sigma}(x) = \int_{-\infty}^{x} \frac{1}{\sigma\sqrt{2\pi}} exp(-\frac{(y-a)^2}{2*\sigma^2}) dy, x \in R.$$

class gaussian

Syntax

```
namespace oneapi::mkl::rng {
template<typename RealType = float, typename Method = gaussian_method::by_default>
class gaussian {
public:
    using method_type = Method;
    using result_type = RealType;
    gaussian();
    explicit gaussian(RealType mean, RealType stddev);
    RealType mean() const;
    RealType stddev() const;
};
}
```

Template parameters

typename RealType

Type of the produced values. Supported types:

- `float`
- `double`

typename Method = oneapi::mkl::rng::gaussian_method::by_default

Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::gaussian_method::by_default`
- `oneapi::mkl::rng::gaussian_method::box_muller`
- `oneapi::mkl::rng::gaussian_method::box_muller2`
- `oneapi::mkl::rng::gaussian_method::icdf`

See description of the methods in *Distributions methods template parameter*

Class Members

Routine	Description
gaussian()	Default constructor
explicit gaussian(RealType mean, RealType stddev)	Constructor with parameters
RealType mean() const	Method to obtain mean value
RealType stddev() const	Method to obtain standard deviation value

Member types

```
gaussian::method_type = Method
```

Description

The type which defines transformation method for generation.

```
gaussian::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
gaussian::gaussian()
```

Description

Default constructor for distribution, parameters set as *mean* = 0.0, *stddev* = 1.0.

```
explicit gaussian::gaussian(RealType mean, RealType stddev)
```

Description

Constructor with parameters. *mean* is a mean value, *stddev* is a standard deviation value.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when *stddev* $\leq$ *static_cast<RealType>(0.0)*

Characteristics

```
RealType gaussian::mean() const
```

Return Value

Returns the distribution parameter *mean* - mean value.

```
RealType gaussian::stddev() const
```

Return Value

Returns the distribution parameter *stddev* - standard deviation value.

Parent topic: *Host Distributions*

exponential

Class is used for generation of exponentially distributed real types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers exponentially distributed with displacement a and scalefactor β, where $a, \beta \in R; \beta > 0$.

The probability distribution is given by:

$$f_{a,\beta}(x) = \begin{cases} \frac{1}{\beta} exp(-\frac{x-a}{\beta}), x \geq a \\ 0, x < a \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \begin{cases} 1 - exp(-\frac{x-a}{\beta}), x \geq a \\ 0, x < a \end{cases}$$

class exponential

Syntax

```
namespace oneapi::mkl::rng {
template<typename RealType = float, typename Method = exponential_method::by_default>
class exponential {
public:
    using method_type = Method;
    using result_type = RealType;
    exponential();
    explicit exponential(RealType a, RealType beta);
    RealType a() const;
    RealType beta() const;
};
}
```

Template parameters

typename RealType

Type of the produced values. Supported types:

- `float`
- `double`

typename Method = oneapi::mkl::rng::exponential_method::by_default

Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::exponential_method::by_default`
- `oneapi::mkl::rng::exponential_method::icdf`
- `oneapi::mkl::rng::exponential_method::icdf_accurate`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
exponential()	Default constructor
explicit exponential(RealType a, RealType beta)	Constructor with parameters
RealType a() const	Method to obtain displacement value
RealType beta() const	Method to obtain scalefactor

Member types

```
exponential::method_type = Method
```

Description

The type which defines transformation method for generation.

```
exponential::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
exponential::exponential()
```

Description

Default constructor for distribution, parameters set as *a* = 0.0, *beta* = 1.0.

```
explicit exponential::exponential(RealType a, RealType beta)
```

Description

Constructor with parameters. *a* is a displacement, *beta* is a scalefactor.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $beta \leq$ static_cast<RealType>(0.0)

Characteristics

```
RealType exponential::a() const
```

Return Value

Returns the distribution parameter *a* - displacement.

```
RealType exponential::beta() const
```

Return Value

Returns the distribution parameter *beta* - scalefactor value.

Parent topic: *Host Distributions*

laplace

Class is used for generation of Laplace distributed real types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers Laplace distributed with mean value (or average) a, and scalefactor (b, β), where $a, \beta \in R; \beta > 0$. The scalefactor value determines the standard deviation as $\sigma = \beta\sqrt{2}$.

The probability distribution is given by:

$$f_{a,\beta}(x) = \frac{1}{\sqrt{2\beta}} exp(-\frac{|x-a|}{\beta}), x \in R.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \begin{cases} \frac{1}{2} exp(-\frac{|x-a|}{\beta}), x \geq a \\ 1 - \frac{1}{2} exp(-\frac{|x-a|}{\beta}), x < a \end{cases}$$

class laplace

Syntax

```
template<typename RealType = float, typename Method = laplace_method::by_default>
class laplace {
public:
    using method_type = Method;
    using result_type = RealType;
    laplace();
    explicit laplace(RealType a, RealType b);
    RealType a() const;
    RealType b() const;
};
```

Template parameters

typename RealType

Type of the produced values. Supported types:

- `float`
- `double`

typename Method = oneapi::mkl::rng::laplace_method::by_default

Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::laplace_method::by_default`
- `oneapi::mkl::rng::laplace_method::icdf`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
laplace()	Default constructor
explicit laplace(RealType a, RealType b)	Constructor with parameters
RealType a() const	Method to obtain mean value
RealType b() const	Method to obtain scalefactor value

Member types

```
laplace::method_type = Method
```

Description

The type which defines transformation method for generation.

```
laplace::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
laplace::laplace()
```

Description

Default constructor for distribution, parameters set as $a = 0.0$, and $beta = 1.0$.

```
explicit laplace::laplace(RealType a, RealType b)
```

Description

Constructor with parameters. a is a mean value, $beta$ is a scalefactor value.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $b \leq$ static_cast<RealType>(0.0)

Characteristics

```
RealType laplace::a() const
```

Return Value

Returns the distribution parameter a - mean value.

```
RealType laplace::b() const
```

Return Value

Returns the distribution parameter b - scalefactor value.

Parent topic: *Host Distributions*

weibull

Class is used for generation of Weibull distributed real types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers Weibull distributed with displacement a, scalefactor β, and shape α, where $a, \beta, \alpha \in R; \alpha > 0; \beta > 0$.

The probability distribution is given by:

$$f_{a,\alpha,\beta}(x) = \begin{cases} \frac{\alpha}{\beta^{\alpha}}(x-a)^{\alpha-1} exp((-\frac{x-a}{\beta})^{\alpha}), x \geq a \\ 0, x < a \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,\alpha,\beta}(x) = \begin{cases} 1 - exp((-\frac{x-a}{\beta})^{\alpha}), x \geq a \\ 0, x < a \end{cases}$$

class weibull

Syntax

```
namespace oneapi::mkl::rng {
template<typename RealType = float, typename Method = weibull_method::by_default>
class weibull {
public:
    using method_type = Method;
```

(continues on next page)

(continued from previous page)

```
    using result_type = RealType;
    weibull();
    explicit weibull(RealType alpha, RealType a, RealType b);
    RealType alpha() const;
    RealType a() const;
    RealType beta() const;
};
}
```

Template parameters

typename RealType

Type of the produced values. Supported types:

- `float`
- `double`

typename Method = oneapi::mkl::rng::weibull_method::by_default

Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::weibull_method::by_default`
- `oneapi::mkl::rng::weibull_method::icdf`
- `oneapi::mkl::rng::weibull_method::icdf_accurate`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
weibull()	Default constructor
explicit weibull(RealType alpha, RealType a, RealType beta)	Constructor with parameters
RealType alpha() const	Method to obtain shape value
RealType a() const	Method to obtain displacement value
RealType beta() const	Method to obtain scalefactor value

Member types

```
weibull::method_type = Method
```

Description

The type which defines transformation method for generation.

```
weibull::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
weibull::weibull()
```

Description

Default constructor for distribution, parameters set as *alpha* = 1.0, *a* = 0.0, and *b* = 1.0.

```
explicit weibull::weibull(RealType alpha, RealType a, RealType beta)
```

Description

Constructor with parameters. *alpha* is a shape value, *a* is a displacement value, *beta* is a scalefactor value.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $alpha \leq$ static_cast<RealType>(0.0), or $beta \leq$ static_cast<RealType>(0.0)

Characteristics

```
RealType weibull::alpha() const
```

Return Value

Returns the distribution parameter *alpha* - shape value.

```
RealType weibull::a() const
```

Return Value

Returns the distribution parameter a - displacement value.

```
RealType weibull::beta() const
```

Return Value

Returns the distribution parameter *beta* - scalefactor value.

Parent topic: *Host Distributions*

cauchy

Class is used for generation of Cauchy distributed real types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers Cauchy distributed with displacement a, and scale parameter (b, β), where $a, \beta \in R; \beta > 0$.

The probability distribution is given by:

$$f_{a,\beta}(x) = \frac{1}{\pi\beta(1 + (\frac{x-a}{\beta})^2)}, x \in R.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \frac{1}{2} + \frac{1}{\pi}\arctan(\frac{x-a}{\beta}), x \in R.$$

class cauchy

Syntax

```
namespace oneapi::mkl::rng {
template<typename RealType = float, typename Method = cauchy_method::by_default>
class cauchy {
public:
    using method_type = Method;
    using result_type = RealType;
    cauchy();
    explicit cauchy(RealType a, RealType b);
    RealType a() const;
    RealType b() const;
};
}
```

Template parameters

typename RealType

Type of the produced values. Supported types:

- `float`
- `double`

typename Method = oneapi::mkl::rng::cauchy_method::by_default

Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::cauchy_method::by_default`
- `oneapi::mkl::rng::cauchy_method::icdf`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
cauchy()	Default constructor
explicit cauchy(RealType a, RealType b)	Constructor with parameters
RealType a() const	Method to obtain displacement value
RealType b() const	Method to obtain scalefactor value

Member types

```
cauchy::method_type = Method
```

Description

The type which defines transformation method for generation.

```
cauchy::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
cauchy::cauchy()
```

Description

Default constructor for distribution, parameters set as a = 0.0, and b = 1.0.

```
explicit cauchy::cauchy(RealType a, RealType b)
```

Description

Constructor with parameters. a is a displacement value, b is a scalefactor value.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $b \leq$ static_cast<RealType>(0.0)

Characteristics

```
RealType cauchy::a() const
```

Return Value

Returns the distribution parameter a - displacement value.

```
RealType cauchy::b() const
```

Return Value

Returns the distribution parameter b - scalefactor value.

Parent topic: *Host Distributions*

rayleigh

Class is used for generation of Rayleigh distributed real types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers Rayleigh distributed with displacement a, and scalefactor (b, β), where $a, \beta \in R; \beta > 0$.

The Rayleigh distribution is a special case of the *weibull* distribution, where the shape parameter *alpha* = 2 .

The probability distribution is given by:

$$f_{a,\beta}(x) = \begin{cases} \frac{2(x-a)}{\beta^2} exp(-\frac{(x-a)^2)}{\beta^2}), x \geq a \\ 0, x < a \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \begin{cases} 1 - exp(-\frac{(x-a)^2)}{\beta^2}), x \geq a \\ 0, x < a \end{cases}$$

class rayleigh

Syntax

```
namespace oneapi::mkl::rng {
template<typename RealType = float, typename Method = rayleigh_method::by_default>
class rayleigh {
public:
    using method_type = Method;
    using result_type = RealType;
    rayleigh();
    explicit rayleigh(RealType a, RealType b);
    RealType a() const;
    RealType b() const;
};
}
```

Template parameters

typename RealType

Type of the produced values. Supported types:

- `float`
- `double`

typename Method = oneapi::mkl::rng::rayleigh_method::by_default

Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::rayleigh_method::by_default`
- `oneapi::mkl::rng::rayleigh_method::icdf`
- `oneapi::mkl::rng::rayleigh_method::icdf_accurate`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
rayleigh()	Default constructor
explicit rayleigh(RealType a, RealType b)	Constructor with parameters
RealType a() const	Method to obtain displacement value
RealType b() const	Method to obtain scalefactor value

Member types

```
rayleigh::method_type = Method
```

Description

The type which defines transformation method for generation.

```
rayleigh::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
rayleigh::rayleigh()
```

Description

Default constructor for distribution, parameters set as $a = 0.0$, and $b = 1.0$.

```
explicit rayleigh::rayleigh(RealType a, RealType b)
```

Description

Constructor with parameters. a is a displacement value, b is a scalefactor value.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $b \leq$ static_cast<RealType>(0.0)

Characteristics

```
RealType rayleigh::a() const
```

Return Value

Returns the distribution parameter a - displacement value.

```
RealType rayleigh::b() const
```

Return Value

Returns the distribution parameter b - scalefactor value.

Parent topic: *Host Distributions*

lognormal

Class is used for generation of lognormally distributed real types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers lognormally distributed with mean (m, a) and standard deviation (s, σ) of subject normal distribution, displacement $(displ, b)$, and scalefactor $(scale, \beta)$, where $a, \sigma, b, \beta \in R; \sigma > 0; \beta > 0$.

The probability distribution is given by:

$$f_{a,\sigma,b,\beta}(x) = \begin{cases} \frac{1}{\sigma(x-b)\sqrt{2\pi}} exp(-\frac{[ln((x-b)/\beta)-a]^2}{2*\sigma^2}), x > b \\ 0, x \leq b \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,\sigma,b,\beta}(x) = \begin{cases} \Phi(\frac{(ln((x-b)/\beta)-a)}{\sigma}), x > b \\ 0, x \leq b \end{cases}$$

class lognormal

Syntax

```
namespace oneapi::mkl::rng {
template<typename RealType = float, typename Method = lognormal_method::by_default>
class lognormal {
public:
    using method_type = Method;
    using result_type = RealType;
    lognormal();
    explicit lognormal(RealType m, RealType s, RealType displ = static_cast<RealType>(0.
↪0), RealType scale = static_cast<RealType>(1.0));
    RealType m() const;
    RealType s() const;
    RealType displ() const;
    RealType scale() const;
};
}
```

Template parameters

typename RealType

Type of the produced values. Supported types:

- `float`
- `double`

typename Method = oneapi::mkl::rng::lognormal_method::by_default

Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::lognormal_method::by_default`
- `oneapi::mkl::rng::lognormal_method::box_muller2`
- `oneapi::mkl::rng::lognormal_method::icdf`
- `oneapi::mkl::rng::lognormal_method::box_muller2_accurate`
- `oneapi::mkl::rng::lognormal_method::icdf_accurate`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
lognormal()	Default constructor
explicit lognormal(RealType m, RealType s, RealType displ = static_cast<RealType>(0.0), RealType scale = static_cast<RealType>(1.0))	Constructor with parameters
RealType m() const	Method to obtain mean value
RealType s() const	Method to obtain standard deviation value
RealType displ() const	Method to obtain displacement value
RealType scale() const	Method to obtain scalefactor value

Member types

```
lognormal::method_type = Method
```

Description

The type which defines transformation method for generation.

```
lognormal::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
lognormal::lognormal()
```

Description

Default constructor for distribution, parameters set as m = 0.0, s = 1.0, *displ* = 0.0, *scale* = 1.0.

```
explicit lognormal::lognormal(RealType m, RealType s, RealType displ = static_cast
↪<RealType>(0.0), RealType scale = static_cast<RealType>(1.0))
```

Description

Constructor with parameters. m is a mean value, s is a standard deviation value, *displ* is a displacement value, *scale* is a scalefactor value.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $s \leq$ static_cast<RealType>(0.0), or $scale \leq$ static_cast<RealType>(0.0)

Characteristics

```
RealType lognormal::m() const
```

Return Value

Returns the distribution parameter m - mean value.

```
RealType lognormal::s() const
```

Return Value

Returns the distribution parameter s - standard deviation value.

```
RealType lognormal::displ() const
```

Return Value

Returns the distribution parameter *displ* - displacement value.

```
RealType lognormal::scale() const
```

Return Value

Returns the distribution parameter *scale* - scalefactor value.

Parent topic: *Host Distributions*

gumbel

Class is used for generation of Gumbel distributed real types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers Gumbel distributed with displacement a, and scalefactor (b, β), where $a, \beta \in R; \beta > 0$.

The probability distribution is given by:

$$f_{a,\beta}(x) = \frac{1}{\beta} exp(-\frac{x-a}{\beta}) exp(-exp(\frac{x-a}{\beta})), x \in R.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = 1 - exp(-exp(\frac{x-a}{\beta})), x \in R.$$

class gumbel

Syntax

```
namespace oneapi::mkl::rng {
template<typename RealType = float, typename Method = gumbel_method::by_default>
class gumbel {
public:
    using method_type = Method;
    using result_type = RealType;
    gumbel();
    explicit gumbel(RealType a, RealType b);
    RealType a() const;
    RealType b() const;
};
}
```

Template parameters

typename RealType

Type of the produced values. Supported types:

- `float`
- `double`

typename Method = oneapi::mkl::rng::gumbel_method::by_default

Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::gumbel_method::by_default`
- `oneapi::mkl::rng::gumbel_method::icdf`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
gumbel()	Default constructor
explicit gumbel(RealType a, RealType b)	Constructor with parameters
RealType a() const	Method to obtain displacement value
RealType b() const	Method to obtain scalefactor value

Member types

```
gumbel::method_type = Method
```

Description

The type which defines transformation method for generation.

```
gumbel::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
gumbel::gumbel()
```

Description

Default constructor for distribution, parameters set as *a* = 0.0, and *beta* = 1.0.

```
explicit gumbel::gumbel(RealType a, RealType b)
```

Description

Constructor with parameters. *a* is a displacement value, *beta* is a scalefactor value.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $b \leq$ static_cast<RealType>(0.0)

Characteristics

```
RealType gumbel::a() const
```

Return Value

Returns the distribution parameter *a* - displacement value.

```
RealType gumbel::b() const
```

Return Value

Returns the distribution parameter *b* - scalefactor value.

Parent topic: *Host Distributions*

gamma

Class is used for generation of gamma distributed real types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers gamma distributed with shape α, displacement a, and scale parameter β, where $a, \alpha, \beta \in R; \alpha > 0; \beta > 0$.

The probability distribution is given by:

$$f_{a,\alpha,\beta}(x) = \begin{cases} \frac{1}{\Gamma(\alpha)\beta^{\alpha}}(x-a)^{\alpha-1}e^{-(x-a)/\beta}, x \geq a \\ 0, x < a \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,\alpha,\beta}(x) = \begin{cases} \int_a^x \frac{1}{\Gamma(\alpha)\beta^{\alpha}}(y-a)^{\alpha-1}e^{-(y-a)/\beta}dy, x \geq a \\ 0, x < a \end{cases}$$

class gamma

Syntax

```
namespace oneapi::mkl::rng {
template<typename RealType = float, typename Method = gamma_method::by_default>
class gamma {
public:
    using method_type = Method;
    using result_type = RealType;
    gamma();
    explicit gamma(RealType alpha, RealType a, RealType beta);
    RealType alpha() const;
    RealType a() const;
    RealType beta() const;
};
}
```

Template parameters

typename RealType

Type of the produced values. Supported types:

- `float`
- `double`

typename Method = oneapi::mkl::rng::gamma_method::by_default

Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::gamma_method::by_default`
- `oneapi::mkl::rng::gamma_method::marsaglia`
- `oneapi::mkl::rng::gamma_method::marsaglia_accurate`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
gamma()	Default constructor
explicit gamma(RealType alpha, RealType a, RealType beta)	Constructor with parameters
RealType alpha() const	Method to obtain shape value
RealType a() const	Method to obtain displacement value
RealType beta() const	Method to obtain scale value

Member types

```
gamma::method_type = Method
```

Description

The type which defines transformation method for generation.

```
gamma::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
gamma::gamma()
```

Description

Default constructor for distribution, parameters set as *alpha* = 1.0, *a* = 0.0, and *beta* = 1.0.

```
explicit gamma::gamma(RealType alpha, RealType a, RealType beta)
```

Description

Constructor with parameters. *alpha* is a shape value, *a* is a displacement value, *beta* is a scale parameter.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $alpha \leq$ static_cast<RealType>(0.0), or $beta \leq$ static_cast<RealType>(0.0)

Characteristics

```
RealType gamma::alpha() const
```

Return Value

Returns the distribution parameter *alpha* - shape value.

```
RealType gamma::a() const
```

Return Value

Returns the distribution parameter *a* - displacement value.

```
RealType gamma::beta() const
```

Return Value

Returns the distribution parameter *beta* - scale parameter.

Parent topic: *Host Distributions*

beta

Class is used for generation of beta distributed real types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers beta distributed with shape parameters p and q, displacement α and scale parameter (b, β), where p, q. α, $\beta \in R; p > 0; q > 0; \beta > 0$.

The probability distribution is given by:

$$f_{p,q,\alpha,\beta}(x) = \begin{cases} \frac{1}{B(p,q)*\beta^{p+q-1}}(x-a)^{p-1} * (\beta+\alpha-x)^{q-1}, \alpha \leq x < \alpha+\beta \\ 0, x < \alpha, x \geq \alpha+\beta \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,b}(x) = \begin{cases} 0, x < \alpha \\ \int_{\alpha}^{x} \frac{1}{B(p,q)*\beta^{p+q-1}}(y-\alpha)^{p-1} * (\beta+\alpha-y)^{q-1} dy, \alpha \leq x < \alpha+\beta, x \in R \\ 1, x \geq \alpha+\beta \end{cases}$$

Where $B(p,1)$ is the complete beta function.

class beta

Syntax

```
namespace oneapi::mkl::rng {
template<typename RealType = float, typename Method = beta_method::by_default>
class beta {
public:
    using method_type = Method;
    using result_type = RealType;
```

(continues on next page)

(continued from previous page)

```
    beta();
    explicit beta(RealType p, RealType q, RealType a, RealType b);
    RealType p() const;
    RealType q() const;
    RealType a() const;
    RealType b() const;
};
}
```

Template parameters

typename RealType

Type of the produced values. Supported types:

- `float`
- `double`

typename Method = oneapi::mkl::rng::beta_method::by_default

Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::beta_method::by_default`
- `oneapi::mkl::rng::beta_method::cja`
- `oneapi::mkl::rng::beta_method::cja_accurate`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
beta()	Default constructor
explicit beta(RealType p, RealType q, RealType a, RealType b)	Constructor with parameters
RealType p() const	Method to obtain shape `p`
RealType q() const	Method to obtain shape `q`
RealType a() const	Method to obtain displacement α
RealType b() const	Method to obtain scalefactor β

Member types

```
beta::method_type = Method
```

Description

The type which defines transformation method for generation.

```
beta::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
beta::beta()
```

Description

Default constructor for distribution, parameters set as p = 1.0, q = 0.0, α = 1.0, β = 1.0.

```
explicit beta::beta(RealType p, RealType q, RealType a, RealType b)
```

Description

Constructor with parameters. p and q are shapes, α is a displacement, β is a scalefactor.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $p \leq 0.0f$, or $q \leq 0.0f$, or $\beta \leq 0.0f$

Characteristics

```
RealType beta::p() const
```

Return Value

Returns the distribution parameter p - shape.

```
RealType beta::q() const
```

Return Value

Returns the distribution parameter `q` - shape.

```
RealType beta::a() const
```

Return Value

Returns the distribution parameter α - displacement.

```
RealType beta::b() const
```

Return Value

Returns the distribution parameter β - scalefactor.

Parent topic: *Host Distributions*

chi_square

Class is used for generation of chi-square distributed real types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers chi-square distributed with n degrees of freedom, $n \in N; n > 0$.

The probability distribution is given by:

$$f_n(x) = \begin{cases} \frac{x^{\frac{n-2}{2}} e^{-\frac{x}{2}}}{2^{n/2}\Gamma(n/2)}, x \geq 0 \\ 0, x < 0 \end{cases}$$

The cumulative distribution function is as follows:

$$F_n(x) = \begin{cases} \int_0^x \frac{y^{\frac{n-2}{2}} e^{-\frac{x}{2}}}{2^{n/2}\Gamma(n/2)} dy, x \geq 0 \\ 0, x < 0 \end{cases}$$

class chi_square

Syntax

```
namespace oneapi::mkl::rng {
template<typename RealType = float, typename Method = chi_square_method::by_default>
class chi_square {
public:
    using method_type = Method;
    using result_type = RealType;
    chi_square();
    explicit chi_square(std::int32_t n);
```

(continues on next page)

(continued from previous page)

```
    std::int32_t n() const;
};
}
```

Template parameters

typename RealType

Type of the produced values. Supported types:

- `float`
- `double`

typename Method = oneapi::mkl::rng::chi_square_method::by_default

Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::chi_square_method::by_default`
- `oneapi::mkl::rng::chi_square_method::gamma_based`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
chi_square()	Default constructor
explicit chi_square(std::int32_t n)	Constructor with parameters
std::int32_t n() const	Method to obtain number of degrees of freedom *n*

Member types

```
chi_square::method_type = Method
```

Description

The type which defines transformation method for generation.

```
chi_square::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
chi_square::chi_square()
```

Description

Default constructor for distribution, parameters set as n = 5.

```
explicit chi_square::chi_square(std::int32_t n)
```

Description

Constructor with parameters. n is the number of degrees of freedom.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $n < 1$

Characteristics

```
std::int32_t chi_square::n() const
```

Return Value

Returns the distribution parameter n - number of degrees of freedom.

Parent topic: *Host Distributions*

gaussian_mv

Class is used for generation of multivariate normally distributed real types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide n random numbers d-variate normally distributed, with mean a and variance-covariance matrix C, where $a \in R^d$; C is dxd symmetric positive matrix.

The probability density function is given by:

$$f_{a,C}(x) = \frac{1}{\sqrt{det(2\pi C)}} exp(-1/2(x-a)^T C^{-1}(x-a)).$$

class gaussian_mv

Let `SequenceContainerOrView` be a type that can be one of C++ Sequence containers or C++ Views (`span`, `mdspan`). It's implementation defined which type `SequenceContainerOrView` represents.

Syntax

```
namespace oneapi::mkl::rng {
template<typename RealType = std::int32_t, layout Layout = layout::packed, typename␣
↪Method = gaussian_mv_method::by_default>
class gaussian_mv {
public:
    using method_type = Method;
    using result_type = RealType;
    explicit gaussian_mv(std::uint32_t dimen, SequenceContainerOrView<RealType> mean,␣
↪SequenceContainerOrView<RealType> matrix);
    std::int32_t dimen() const;
    SequenceContainerOrView<RealType> mean() const;
    SequenceContainerOrView<RealType> matrix() const;
};
}
```

Template parameters

typename RealType

Type of the produced values. Supported types:

- `float`
- `double`

Template parameters

oneapi::mkl::rng::layout Layout

Matrix layout:

- `oneapi::mkl::rng::layout::full`
- `oneapi::mkl::rng::layout::packed`
- `oneapi::mkl::rng::layout::diagonal`

typename Method = oneapi::mkl::rng::gaussian_mv_method::by_default

Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::gaussian_mv_method::by_default`
- `oneapi::mkl::rng::gaussian_mv_method::box_muller`
- `oneapi::mkl::rng::gaussian_mv_method::box_muller2`
- `oneapi::mkl::rng::gaussian_mv_method::icdf`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
explicit gaussian_mv(std::uint32_t dimen, SequenceContainerOrView<RealType> mean, SequenceContainerOrView<RealType> matrix)	Constructor with parameters
std::int32_t dimen() const	Method to obtain number of dimensions in output random vectors
SequenceContainerOrView<double> mean() const	Method to obtain mean vector a of dimension d.
SequenceContainerOrView<double> matrix() const	Method to obtain variance-covariance matrix C

Member types

```
gaussian_mv::method_type = Method
```

Description

The type which defines transformation method for generation.

```
gaussian_mv::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
explicit gaussian_mv::gaussian_mv(std::uint32_t dimen, SequenceContainerOrView<RealType>␣
↪mean, SequenceContainerOrView<RealType> matrix)
```

Description

Constructor with parameters. `dimen` is the number of dimensions, `mean` is a mean vector, `matrix` is a variance-covariance matrix.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $mean.size() \leq 0$, or $matrix.size() \leq 0$

Characteristics

```
std::int32_t gaussian_mv::dimen() const
```

Return Value

Returns the distribution parameter *dimen*.

```
SequenceContainerOrView<double> gaussian_mv::mean() const
```

Return Value

Returns the mean vector.

```
SequenceContainerOrView<double> gaussian_mv::matrix() const
```

Return Value

Returns the variance-covariance matrix.

Parent topic: *Host Distributions*

uniform (discrete)

Class is used for generation of uniformly distributed integer types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers uniformly distributed over the interval $[a, b)$, where a, b are the left and right bounds of the interval, respectively, and $a, b \in R; a < b$.

The probability distribution is given by:

$$P(X = k) = \frac{1}{b-a}, k \in \{a, a+1, ..., b-1\}$$

The cumulative distribution function is as follows:

$$F_{a,b}(x) = \begin{cases} 0, x < a \\ \frac{\lfloor x-a+1 \rfloor}{b-a}, a \leq x < b, x \in R \\ 1, x \geq b \end{cases}$$

class uniform

Syntax

```
namespace oneapi::mkl::rng {
template<typename Method = uniform_method::by_default>
class uniform<std::int32_t, Method> {
public:
    using method_type = Method;
    using result_type = std::int32_t;
    uniform();
    explicit uniform(std::int32_t a, std::int32_t b);
    std::int32_t a() const;
    std::int32_t b() const;
};
}
```

Template parameters

typename Method = oneapi::mkl::rng::uniform_method::by_default
Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::uniform_method::by_default`
- `oneapi::mkl::rng::uniform_method::standard`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
uniform()	Default constructor
explicit uniform(std::int32_t a, std::int32_t b)	Constructor with parameters
std::int32_t a() const	Method to obtain left bound *a*
std::int32_t b() const	Method to obtain right bound *b*

Member types

```
method_type = Method
```

Description

The type which defines transformation method for generation.

```
result_type = std::int32_t
```

Description

The type which defines type of generated random numbers.

Constructors

```
uniform()
```

Description

Default constructor for distribution, parameters set as a = 0, b = std::numeric_limits<std::int32_t>::max().

```
uniform(std::int32_t a, std::int32_t b)
```

Description

Constructor with parameters. a is a left bound, b is a right bound, assume $a < b$.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $a \geq b$

Characteristics

```
a() const
```

Return Value

Returns the distribution parameter a - left bound.

```
b() const
```

Return Value

Returns the distribution parameter *b* - right bound.

Parent topic: *Host Distributions*

uniform_bits

Class is used for generation of uniformly distributed bits in 32/64-bit chunks.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide uniformly distributed bits in 32/64-bit chunks. It is designed to ensure each bit in the 32/64-bit chunk is uniformly distributed. Can be not supported by the specific engine.

class uniform_bits

Syntax

```
namespace oneapi::mkl::rng {
template<typename UIntType = std::uint32_t>
class uniform_bits {
public:
    using result_type = UIntType;
};
}
```

Template parameters

typename UIntType

Type of the produced values. Supported types:

- `std::uint32_t`
- `std::uint64_t`

Member types

```
uniform_bits::result_type = UIntType
```

Description

The type which defines type of generated random numbers.

Parent topic: *Host Distributions*

bits

Class is used for generation of underlying engine integer recurrence.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide integer random numbers. Each integer can be treated as a vector of several bits. In a truly random generator, these bits are random, while in pseudorandom generators this randomness can be violated.

class bits

Syntax

```
namespace oneapi::mkl::rng {
template<typename UIntType = std::uint32_t>
class bits {
public:
    using result_type = UIntType;
};
}
```

Template parameters

typename UIntType

Type of the produced values. Supported types:

- `std::uint32_t`

Member types

```
bits::result_type = UIntType
```

Description

The type which defines type of generated random numbers.

Parent topic: *Host Distributions*

bernoulli

Class is used for generation of Bernoulli distributed integer types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers Bernoulli distributed with probability p of a single trial success, where $p \in R; 0 \leq p; p \leq 1$.

The probability distribution is given by:

$$P(X = 1) = p$$

$$P(X = 0) = 1 - p$$

The cumulative distribution function is as follows:

$$F_p(x) = \begin{cases} 0, x < 0 \\ 1 - p, 0 \leq x < 1, x \in R \\ 1, x \geq 1 \end{cases}$$

class bernoulli

Syntax

```
namespace oneapi::mkl::rng {
template<typename IntType = std::int32_t, typename Method = bernoulli_method::by_default>
class bernoulli {
public:
    using method_type = Method;
    using result_type = IntType;
    bernoulli();
    explicit bernoulli(float p);
    float p() const;
};
}
```

Template parameters

typename IntType

Type of the produced values. Supported types:

- `std::int32_t`
- `std::uint32_t`

typename Method = oneapi::mkl::rng::bernoulli_method::by_default

Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::bernoulli_method::by_default`
- `oneapi::mkl::rng::bernoulli_method::icdf`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
bernoulli()	Default constructor
explicit bernoulli(float p)	Constructor with parameters
float p() const	Method to obtain probability p

Member types

```
bernoulli::method_type = Method
```

Description

The type which defines transformation method for generation.

```
bernoulli::result_type = IntType
```

Description

The type which defines type of generated random numbers.

Constructors

```
bernoulli::bernoulli()
```

Description

Default constructor for distribution, parameters set as $p = 0.5$f.

```
explicit bernoulli::bernoulli(float p)
```

Description

Constructor with parameters. p is a probability.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when *p > 1.0f*, or *p < 0.0f*

Characteristics

```
float p() const
```

Return Value

Returns the distribution parameter p - probability.

Parent topic: *Host Distributions*

geometric

Class is used for generation of geometrically distributed integer types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers geometrically distributed with probability p of a single success trial, where $p \in R; 0 < p < 1$.

The probability distribution is given by:

$$P(X = k) = p * (1 - p)^k, k = \{0, 1, 2, ...\}.$$

The cumulative distribution function is as follows:

$$F_p(x) = \begin{cases} 0, x < 0 \\ 1 - (1 - p)^{\lfloor x+1 \rfloor}, x \geq 0 \end{cases}$$

class geometric

Syntax

```
namespace oneapi::mkl::rng {
template<typename IntType = std::int32_t, typename Method = geometric_method::by_default>
class geometric {
public:
    using method_type = Method;
    using result_type = IntType;
    geometric();
    explicit geometric(float p);
```

(continues on next page)

(continued from previous page)

```
    float p() const;
};
}
```

Template parameters

typename IntType

Type of the produced values. Supported types:

- `std::int32_t`
- `std::uint32_t`

typename Method = oneapi::mkl::rng::geometric_method::by_default

Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::geometric_method::by_default`
- `oneapi::mkl::rng::geometric_method::icdf`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
geometric()	Default constructor
explicit geometric(float p)	Constructor with parameters
float p() const	Method to obtain probability value

Member types

```
geometric::method_type = Method
```

Description

The type which defines transformation method for generation.

```
geometric::result_type = IntType
```

Description

The type which defines type of generated random numbers.

Constructors

```
geometric::geometric()
```

Description

Default constructor for distribution, parameters set as p = 0.5.

```
explicit geometric::geometric(float p)
```

Description

Constructor with parameters. p is a probability value.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $p \geq 1.0f$, or $p \leq 0.0f$

Characteristics

```
float geometric::p() const
```

Return Value

Returns the distribution parameter p - probability value.

Parent topic: *Host Distributions*

binomial

Class is used for generation of binomially distributed integer types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers binomially distributed with a number of independent Bernoulli trials m, and with probability p of a single trial success, where $p \in R; 0 \leq p \leq 1, m \in N$.

A binomially distributed variate represents the number of successes in m independent Bernoulli trials with probability of a single trial success p.

The probability distribution is given by:

$$P(X = k) = C_m^k p^k (1-p)^{m-k}, k \in \{0, 1, ..., m\}$$

The cumulative distribution function is as follows:

$$F_{m,p}(x) = \begin{cases} 0, x < 0 \\ \sum_{k=0}^{\lfloor x \rfloor} C_m^k p^k (1-p)^{m-k}, 0 \le x < m, x \in R \\ 1, x \ge m \end{cases}$$

class binomial

Syntax

```
namespace oneapi::mkl::rng {
template<typename IntType = std::int32_t, typename Method = binomial_method::by_default>
class binomial {
public:
    using method_type = Method;
    using result_type = IntType;
    binomial();
    explicit binomial(std::int32_t ntrial, double p);
    std::int32_t ntrial() const;
    double p() const;
};
}
```

Template parameters

typename IntType

Type of the produced values. Supported types:

- `std::int32_t`

typename Method = oneapi::mkl::rng::binomial_method::by_default

Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::binomial_method::by_default`
- `oneapi::mkl::rng::binomial_method::btpe`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
binomial()	Default constructor
explicit binomial(std::int32_t ntrial, double p)	Constructor with parameters
std::int32_t ntrial() const	Method to obtain number of independent trials m
double p() const	Method to obtain success probability of a single trial p

Member types

```
binomial::method_type = Method
```

Description

The type which defines transformation method for generation.

```
binomial::result_type = IntType
```

Description

The type which defines type of generated random numbers.

Constructors

```
binomial::binomial()
```

Description

Default constructor for distribution, parameters set as $m = 5$, $p = 0.5$.

```
explicit binomial::binomial(std::int32_t ntrial, double p)
```

Description

Constructor with parameters. *ntrial* is the number of independent trials, p is the success probability of a single trial.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $p > 1.0$, or $p < 0.0$, or $ntrial < 1$

Characteristics

```
std::int32_t binomial::ntrial() const
```

Return Value

Returns the distribution parameter m - number of independent trials.

```
double binomial::p() const
```

Return Value

Returns the distribution parameter p - success probability of a single trial.

Parent topic: *Host Distributions*

hypergeometric

Class is used for generation of hypergeometrically distributed integer types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers hypergeometrically distributed with lot size l, size of sampling s, and number of marked elements in the lot m, where $l, m, s \in N \bigcup \{0\}; l \geq max(s, m)$.

Consider a lot of l elements comprising m marked and l - m unmarked elements. A trial sampling without replacement of exactly s elements from this lot helps to define the hypergeometric distribution, which is the probability that the group of s elements contains exactly k marked elements.

The probability distribution is given by:

$$P_(X = k) = \frac{C_m^k C_{l-m}^{s-k}}{C_l^s}, k \in \{max(0, s + m - l), ..., min(s, m)\}.$$

The cumulative distribution function is as follows:

$$F_{l,s,m}(x) = \begin{cases} 0, x < max(0, s + m - l) \\ \sum_{k=max(0,s+m-l)}^{\lfloor x \rfloor} \frac{C_m^k C_{l-m}^{s-k}}{C_l^s}, max(0, s + m - l) \leq x \leq min(s, m) \\ 1, x > min(s, m) \end{cases}$$

class hypergeometric

Syntax

```
namespace oneapi::mkl::rng {
template<typename IntType = std::int32_t, typename Method = hypergeometric_method::by_
↪default>
class hypergeometric {
public:
    using method_type = Method;
    using result_type = IntType;
    hypergeometric();
    explicit hypergeometric(std::int32_t l, std::int32_T s, std::int32_T m);
    std::int32_t s() const;
```

(continues on next page)

(continued from previous page)

```
    std::int32_t m() const;
    std::int32_t l() const;
};
}
```

Template parameters

typename IntType

Type of the produced values. Supported types:

- `std::int32_t`
- `std::uint32_t`

typename Method = oneapi::mkl::rng::hypergeometric_method::by_default

Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::hypergeometric_method::by_default`
- `oneapi::mkl::rng::hypergeometric_method::h2pe`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
hypergeometric()	Default constructor
explicit hypergeometric(std::int32_t l, std::int32_T s, std::int32_T m)	Constructor with parameters
std::int32_t s() const	Method to obtain lot size
std::int32_t m() const	Method to obtain size of sampling without replacement
std::int32_t l() const	Method to obtain number of marked elements

Member types

```
hypergeometric::method_type = Method
```

Description

The type which defines transformation method for generation.

```
hypergeometric::result_type = IntType
```

Description

The type which defines type of generated random numbers.

Constructors

```
hypergeometric::hypergeometric()
```

Description

Default constructor for distribution, parameters set as $l = 1$, $s = 1$, $m = 1$.

```
explicit hypergeometric::hypergeometric(std::int32_t l, std::int32_T s, std::int32_T m)
```

Description

Constructor with parameters. l is a lot size, s is a size of sampling without replacement, m is a number of marked elements.

Throws

oneapi::mkl::invalid_argument
: Exception is thrown when $s < 0$, or $m < 0$, or $l < (s > m ? s : m)$

Characteristics

```
std::int32_t hypergeometric::l() const
```

Return Value

Returns the distribution parameter l - lot size value.

```
std::int32_t hypergeometric::s() const
```

Return Value

Returns the distribution parameter s - size of sampling without replacement.

```
std::int32_t hypergeometric::m() const
```

Return Value

Returns the distribution parameter *m* - number of marked elements.

Parent topic: *Host Distributions*

poisson

Class is used for generation of Poisson distributed integer types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers Poisson distributed with distribution parameter λ, where $\lambda \in R; \lambda > 0;$.

The probability distribution is given by:

$$P(X = k) = \frac{\lambda^k e^{-\lambda}}{k!}.$$

The cumulative distribution function is as follows:

$$F_\lambda(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} \frac{\lambda^k e^{-\lambda}}{k!}, x \geq 0 \\ 0, x < 0 \end{cases}$$

class poisson

Syntax

```
namespace oneapi::mkl::rng {
template<typename IntType = std::int32_t, typename Method = poisson_method::by_default>
class poisson {
public:
    using method_type = Method;
    using result_type = IntType;
    poisson();
    explicit poisson(double lambda);
    double lambda() const;
};
}
```

Template parameters

typename IntType

Type of the produced values. Supported types:

- `std::int32_t`

typename Method = oneapi::mkl::rng::poisson_method::by_default

Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::poisson_method::by_default`

- `oneapi::mkl::rng::poisson_method::ptpe`
- `oneapi::mkl::rng::poisson_method::gaussian_icdf_based`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
poisson()	Default constructor
explicit poisson(double lambda)	Constructor with parameters
double lambda() const	Method to obtain distribution parameter

Member types

```
poisson::method_type = Method
```

Description

The type which defines transformation method for generation.

```
poisson::result_type = IntType
```

Description

The type which defines type of generated random numbers.

Constructors

```
poisson::poisson()
```

Description

Default constructor for distribution, parameters set as *lambda* = 0.5.

```
explicit poisson::poisson(double lambda)
```

Description

Constructor with parameters. *lambda* is a distribution parameter.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $lambda \leq 0.0$

Characteristics

```
double poisson::lambda() const
```

Return Value

Returns the distribution parameter *lambda*.

Parent topic: *Host Distributions*

poisson_v

Class is used for generation of Poisson distributed integer types random numbers with varying mean.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide n random numbers Poisson distributed, with distribution parameter λ_i, where $\lambda_i \in R; \lambda_i > 0; i = 1, ..., n$.

The probability distribution is given by:

$$P(X_i = k) = \frac{\lambda_i^k e^{-\lambda_i}}{k!}, k \in \{0, 1, 2, ...\}.$$

The cumulative distribution function is as follows:

$$F_{\lambda_i}(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} \frac{\lambda_i^k e^{-\lambda_i}}{k!}, x \geq 0 \\ 0, x < 0 \end{cases}$$

class poisson_v

Let `SequenceContainerOrView` be a type that can be one of C++ Sequence containers or C++ Views (`span`, `mdspan`). It's implementation defined which type `SequenceContainerOrView` represents.

Syntax

```
namespace oneapi::mkl::rng {
template<typename IntType = std::int32_t, typename Method = poisson_v_method::by_default>
class poisson_v {
public:
    using method_type = Method;
    using result_type = IntType;
    explicit poisson_v(SequenceContainerOrView<double> lambda);
    SequenceContainerOrView<double> lambda() const;
};
}
```

Template parameters

typename IntType

Type of the produced values. Supported types:

- `std::int32_t`

typename Method = oneapi::mkl::rng::poisson_v_method::by_default

Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::poisson_v_method::by_default`
- `oneapi::mkl::rng::poisson_v_method::gaussian_icdf_based`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
explicit poisson_v(SequenceContainerOrView<double> lambda)	Constructor with parameters
SequenceContainerOrView<double> lambda() const	Method to obtain distribution parameter

Member types

```
poisson_v::method_type = Method
```

Description

The type which defines transformation method for generation.

```
poisson_v::result_type = IntType
```

Description

The type which defines type of generated random numbers.

Constructors

```
explicit poisson_v::poisson_v(SequenceContainerOrView<double> lambda)
```

Description

Constructor with parameters. `lambda` is a distribution parameter.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $lambda.size() \leq 1$

Characteristics

```
SequenceContainerOrView<double> poisson_v::lambda() const
```

Return Value

Returns the distribution parameter `lambda`.

Parent topic: *Host Distributions*

negative_binomial

Class is used for generation of negative binomially distributed integer types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers negative binomially distributed with distribution parameters a and p, where $p, a \in R; 0 \leq p \leq 1, a > 0$.

The probability distribution is given by:

$$P(X = k) = C^{k}_{a+k-1} p^{a} (1-p)^{k}, k \in \{0, 1, 2, ...\}$$

The cumulative distribution function is as follows:

$$F_{a,p}(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} C_{a+k-1}^{k} p^{a}(1-p)^{k}, x \geq 0, x \in R \\ 0, x < 0 \end{cases}$$

class negative_binomial

Syntax

```
namespace oneapi::mkl::rng {
template<typename IntType = std::int32_t, typename Method = negative_binomial_method::by_
↪default>
class negative_binomial {
public:
    using method_type = Method;
    using result_type = IntType;
    negative_binomial();
    explicit negative_binomial(double a, double p);
    double a() const;
    double p() const;
};
}
```

Template parameters

typename IntType

Type of the produced values. Supported types:

- `std::int32_t`
- `std::uint32_t`

typename Method = oneapi::mkl::rng::negative_binomial_method::by_default

Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::negative_binomial_method::by_default`
- `oneapi::mkl::rng::negative_binomial_method::nbar`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
negative_binomial()	Default constructor
explicit negative_binomial(double a, double p)	Constructor with parameters
double a() const	Method to obtain the first distribution parameter *a*
double p() const	Method to obtain the second distribution parameter *p*

Member types

```
negative_binomial::method_type = Method
```

Description

The type which defines transformation method for generation.

```
negative_binomial::result_type = IntType
```

Description

The type which defines type of generated random numbers.

Constructors

```
negative_binomial::negative_binomial()
```

Description

Default constructor for distribution, parameters set as $a = 0.1$, $p = 0.5$.

```
explicit negative_binomial::negative_binomial(double a, double p)
```

Description

Constructor with parameters. a is the first distribution parameter, p is the second distribution parameter.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $p \geq 1.0$, or $p \leq 0.0$, or $a \leq 0.0$

Characteristics

```
double negative_binomial::a() const
```

Return Value

Returns the distribution parameter a - the first distribution parameter.

```
double negative_binomial::p() const
```

Return Value

Returns the distribution parameter p - the second distribution parameter.

Parent topic: *Host Distributions*

multinomial

Class is used for generation of multinomially distributed integer types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide n random numbers multinomially distributed, with independent trials $(ntrial, m)$ and possible mutually exclusive outcomes k, with corresponding probabilities p_i, where $p_i \in R; 0 \leq p_i \leq 1; m, k \in N$.

The probability distribution is given by:

$$P(X_1 = x_1, ..., X_k = x_k) = \frac{m!}{\Pi_{i=1}^{k} x_i!} \Pi_{i=1}^{k} p_i^{x_i}, 0 \leq x_i \leq m, \sum_{i=1}^{k} x_i = m$$

class multinomial

Let `SequenceContainerOrView` be a type that can be one of C++ Sequence containers or C++ Views (`span`, `mdspan`). It's implementation defined which type `SequenceContainerOrView` represents.

Syntax

```
namespace oneapi::mkl::rng {
template<typename IntType = std::int32_t, typename Method = multinomial_method::by_
↪default>
class multinomial {
public:
    using method_type = Method;
    using result_type = IntType;
    explicit multinomial(double ntrial, SequenceContainerOrView<double> p);
    std::int32_t ntrial() const;
    SequenceContainerOrView<double> p() const;
};
}
```

Template parameters

typename IntType

Type of the produced values. Supported types:

- `std::int32_t`
- `std::uint32_t`

typename Method = oneapi::mkl::rng::multinomial_method::by_default

Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::multinomial_method::by_default`
- `oneapi::mkl::rng::multinomial_method::poisson_icdf_based`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
explicit multinomial(double ntrial, SequenceContainerOrView<double> p)	Constructor with parameters
std::int32_t ntrial() const	Method to obtain number of independent trials
SequenceContainerOrView<double> p() const	Method to obtain a probability parameter of possible outcomes

Member types

```
multinomial::method_type = Method
```

Description

The type which defines the transformation method for generation.

```
multinomial::result_type = IntType
```

Description

The type which defines the type of generated random numbers.

Constructors

```
explicit multinomial::multinomial(double ntrial, SequenceContainerOrView<double> p)
```

Description

Constructor with parameters. `ntrial` is a number of independent trials, `p` is a probability parameter.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $ntrial < 0$, or $p.size() < 1$

Characteristics

```
std::int32_t multinomial::ntrial() const
```

Return Value

Returns the distribution parameter *ntrial*.

```
SequenceContainerOrView<double> multinomial::p() const
```

Return Value

Returns the distribution parameter *p*.

Parent topic: *Host Distributions*

Bibliography

For more information about the RNG functionality, refer to the following publications:

- **RNG**

[Bratley88]
Bratley P. and Fox B.L. *Implementing Sobol's Quasirandom Sequence Generator*, ACM Transactions on Mathematical Software, Vol. 14, No. 1, Pages 88-100, March 1988.

[Bratley92]
Bratley P., Fox B.L., and Niederreiter H. *Implementation and Tests of Low-Discrepancy Sequences*, ACM Transactions on Modeling and Computer Simulation, Vol. 2, No. 3, Pages 195-213, July 1992.

[Coddington94]
Coddington, P. D. *Analysis of Random Number Generators Using Monte Carlo Simulation*. Int. J. Mod. Phys. C-5, 547, 1994.

[L'Ecuyer99]
L'Ecuyer, Pierre. *Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure*. Mathematics of Computation, 68, 225, 249-260, 1999.

[L'Ecuyer99a]
L'Ecuyer, Pierre. *Good Parameter Sets for Combined Multiple Recursive Random Number Generators*. Operations Research, 47, 1, 159-164, 1999.

[Kirkpatrick81]
Kirkpatrick, S., and Stoll, E. *A Very Fast Shift-Register Sequence Random Number Generator*. Journal of Computational Physics, V. 40. 517-526, 1981.

[Matsumoto98]
Matsumoto, M., and Nishimura, T. *Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator*, ACM Transactions on Modeling and Computer Simulation, Vol. 8, No. 1, Pages 3-30, January 1998.

[Matsumoto00]
Matsumoto, M., and Nishimura, T. *Dynamic Creation of Pseudorandom Number Generators*, 56-69, in: Monte Carlo and Quasi-Monte Carlo Methods 1998, Ed. Niederreiter, H. and Spanier, J., Springer 2000, http://www.math.sci.hiroshima-u.ac.jp/%7Em-mat/MT/DC/dc.html.

[NAG]
NAG Numerical Libraries. http://www.nag.co.uk/numeric/numerical_libraries.asp

[Saito08]
Saito, M., and Matsumoto, M. *SIMD-oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator*. Monte Carlo and Quasi-Monte Carlo Methods 2006, Springer, Pages 607 – 622, 2008.

http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/earticles.html

[Salmon11]
Salmon, John K., Morales, Mark A., Dror, Ron O., and Shaw, David E., *Parallel Random Numbers: As Easy as 1, 2, 3*. SC '11 Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 2011.

[Sobol76]
Sobol, I.M., and Levitan, Yu.L. *The production of points uniformly distributed in a multidimensional cube*. Preprint 40, Institute of Applied Mathematics, USSR Academy of Sciences, 1976 (In Russian).

[MT2203]
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html.

[FIPS-197]
Federal Information Processing Standards Publication 197, ADVANCED ENCRYPTION STANDARD (AES)

Parent topic: *Random Number Generators*

Random Number Generators Device Routines

The main purpose of Device routines is to make them callable from your SYCL kernels; however, there are no limitations to be called from the Host. For example:

```
sycl::queue queue;

queue.submit([&](sycl::handler& cgh) {
  cgh.parallel_for(range,[=](...) {
     oneapi::mkl::rng::device::routine(...); // calling routine from user's kernel code
  });
});
```

(continues on next page)

(continued from previous page)

```
oneapi::mkl::rng::device::routine(...); // calling routine from host
```

Structure

RNG domain contains two classes types:

- Engines (basic random number generators) classes, which holds the state of generator and is a source of independent and identically distributed random variables. Refer to *Host Engines (Basic Random Number Generators)* for a detailed description.
- Distribution classes templates (transformation classes) for different types of statistical distributions, for example, uniform, normal (Gaussian), binomial, etc. These classes contain all of the distribution's parameters (including generation method). Refer to *Device Distributions* for a detailed description of the distributions.

The RNG domain also contains two types of free functions:

- Generation routines. The current routines are used to obtain random numbers from a given engine with proper statistics defined by a given distribution. Refer to the *Device Generate Routines* section for a detailed description.
- Service routines. The routines are used to modify the engine state. Refer to *Device Service Routines* for a description of these routines.

Engine classes work with both generation and service routines. Distribution classes are used in generation routines only. Refer to the *oneMKL RNG Device Usage Model* section for the description of typical RNG scenario.

oneMKL RNG Device Usage Model

- *Example of Scalar Random Numbers Generation*
- *Example of Vector Random Numbers Generation*

A typical usage model for device routines is the same as described in *oneMKL RNG Host Usage Model*:

1. Create and initialize the object for basic random number generator.
2. Create and initialize the object for distribution generator.
3. Call the generate routine to get random numbers with appropriate statistical distribution.

Example of Scalar Random Numbers Generation

```
#include "oneapi/mkl/rng/device.hpp"

int main() {
    sycl::queue q;
    // Prepare a memory for random numbers
    // Submit a kernel to generate on device
    q.submit([&](sycl::handler& cgh) {
        // ...
        cgh.parallel_for(n, [=](size_t idx) {
            // Create an engine object
```

(continues on next page)

(continued from previous page)

```
            oneapi::mkl::rng::device::philox4x32x10<> engine(seed, idx);
            // Create a distribution object
            oneapi::mkl::rng::device::uniform<float> distr;
            // Call generate function to obtain scalar random number
            float res = oneapi::mkl::rng::device::generate(distr, engine);
            // ...
        });
    });
    // ...
}
```

Example of Vector Random Numbers Generation

```
#include "oneapi/mkl/rng/device.hpp"

int main() {
    sycl::queue q;
    // Prepare an array for random numbers
    // Submit a kernel to generate on device
    q.submit([&](sycl::handler& cgh) {
        // ...
        cgh.parallel_for((n / vec_size), [=](size_t idx) {
            // Create an engine object
            oneapi::mkl::rng::device::philox4x32x10<vec_size> engine(seed, idx * vec_
↪size);
            // Create a distribution object
            oneapi::mkl::rng::device::uniform<float> distr;
            // Call generate function to obtain random numbers
            sycl::vec<float, vec_size> res = oneapi::mkl::rng::device::generate(distr,␣
↪engine);
            // ...
        });
    });
    // ...
}
```

Parent topic: *Random Number Generators Device Routines*

Device Generate Routines

Use the *generate* routine to obtain random numbers from a given engine with proper statistics of a given distribution.

generate

Description

Entry point to obtain random numbers from a given engine with proper statistics of a given distribution.

Syntax

```
namespace oneapi::mkl::rng::device {
  template<typename Distr, typename Engine>
  auto generate(Distr& distr, Engine& engine) ->
    typename std::conditional<Engine::vec_size == 1, typename Distr::result_type,
                              sycl::vec<typename Distr::result_type, Engine::vec_size>>
↪::type
}
```

Template Parameters

Distr
: Type of distribution which is used for random number generation.

Engine
: Type of engine which is used for random number generation.

Input Parameters

distr
: Distribution object. See *Device Distributions* for details.

engine
: Engine object. See *Device Engines (Basic Random Number Generators)* for details.

Return Value

Returns *Distr::result_type* if *Engine::vec_size == 1* or *sycl::vec<typename Distr::result_type, Engine::vec_size>* with generated random numbers.

Parent topic: *Device Generate Routines*

Device Engines (Basic Random Number Generators)

oneMKL RNG provides following device pseudorandom number generators:

Routine	Description
mrg32k3a	The combined multiple recursive pseudorandom number generator `MRG32k3a` [*L'Ecuyer99*]
philox4x32x10	Philox4x32-10 counter-based pseudorandom number generator with a period of 2^{128} `PHILOX4X32X10` [*Salmon11*]
mcg31m1	The 31-bit multiplicative congruential pseudorandom number generator MCG($1132489760, 2^{32}-1$) *[L'Ecuyer99a]*.
mcg59	The 59-bit multiplicative congruential pseudorandom number generator MCG($13^{13}, 2^{59}$) from NAG Numerical Libraries *[NAG]*.

Parent topic: *Random Number Generators Device Routines*

mrg32k3a

The combined multiple recursive pseudorandom number generator MRG32k3a.

Description

MRG32k3a engine is a 32-bit combined multiple recursive generator with two components of order 3 [*L'Ecuyer99a*]. MRG32k3a combined generator meets the requirements for modern RNGs, such as good multidimensional uniformity, or a long period ($p \approx 2^{191}$).

Generation algorithm

$x_n = a_{11}x_{n-1} + a_{12}x_{n-2} + a_{13}x_{n-3} (mod\ m_1)$

$y_n = a_{21}y_{n-1} + a_{22}y_{n-2} + a_{23} (mod\ m_2)$

$z_n = x_n - y_n (mod\ m_1)$

$u_n = z_n / m_1$

$a_{11} = 0, a_{12} = 1403580, a_{13} = -810728, m_1 = 2^{32} - 209$

$a_{21} = 527612, a_{22} = 0, a_{23} = -1370589, m_2 = 2^{32} - 22853$

class mrg32k3a

Syntax

```
namespace oneapi::mkl::rng::device {
  template<std::int32_t VecSize = 1>
  class mrg32k3a {
  public:
    static constexpr std::uint32_t default_seed = 1;
    static constexpr std::int32_t vec_size = VecSize;
```

(continues on next page)

(continued from previous page)

```
    mrg32k3a();
    mrg32k3a(std::uint32_t seed, std::uint64_t offset = 0);
    mrg32k3a(std::initializer_list<std::uint32_t> seed, std::uint64_t offset = 0);
    mrg32k3a(std::uint32_t seed, std::initializer_list<std::uint64_t> offset);
    mrg32k3a(std::initializer_list<std::uint32_t> seed, std::initializer_list
↪<std::uint64_t> offset);
  };
}
```

Class Template Parameters

VecSize

Describes the size of vector which will be produced by generate function by this engine. VecSize values may be 1, 2, 3, 4, 8, 16 as `sycl::vec` class size. By default VecSize = 1, for this case, a single random number is returned by the `generate` function.

Class Members

Routine	Description
mrg32k3a()	Default constructor
mrg32k3a(std::uint32_t seed, std::uint64_t offset = 0)	Constructor for common seed initialization of the engine and common number of skipped elements
mrg32k3a(std::initializer_list<std::uint32_t> seed, std::uint64_t offset = 0)	Constructor for extended seed initialization of the engine and common number of skipped elements
mrg32k3a(std::uint32_t seed, std::initializer_list<std::uint64_t> offset)	Constructor for common seed initialization of the engine and extended number of skipped elements
mrg32k3a(std::initializer_list<std::uint32_t> seed, std::initializer_list<std::uint64_t> offset)	Constructor for extended seed initialization of the engine and extended number of skipped elements

Constructors

```
mrg32k3a::mrg32k3a()
```

```
mrg32k3a::mrg32k3a(std::uint32_t seed, std::uint64_t offset = 0)
```

Input Parameters

seed

The initial conditions of the generator state, assume if $n = 0 : x_{-3} = x_{-2} = x_{-1} = y_{-3} = y_{-2} = y_{-1} = 1$

if $n = 1 : x_{-3} = seed[0] \ mod \ m_1, x_{-2} = x_{-1} = y_{-3} = y_{-2} = y_{-1} = 1$

if $n = 2 : x_{-3} = seed[0] \ mod \ m_1, x_{-2} = seed[1] \ mod \ m_1, x_{-1} = y_{-3} = y_{-2} = y_{-1} = 1$

if $n = 3 : x_{-3} = seed[0] \ mod \ m_1, x_{-2} = seed[1] \ mod \ m_1, x_{-1} = seed[2] \ mod \ m_1$

$y_{-3} = y_{-2} = y_{-1} = 1$

if $n = 4 : x_{-3} = seed[0] \ mod \ m_1, x_{-2} = seed[1] \ mod \ m_1, x_{-1} = seed[2] \ mod \ m_1$

$y_{-3} = seed[3] \ mod \ m_2, y_{-2} = y_{-1} = 1$

if $n = 5 : x_{-3} = seed[0] \ mod \ m_1, x_{-2} = seed[1] \ mod \ m_1, x_{-1} = seed[2] \ mod \ m_1$

$y_{-3} = seed[3] \ mod \ m_2, y_{-2} = seed[4] \ mod \ m_2, y_{-1} = 1$

if $n \geqslant 6 : x_{-3} = seed[0] \ mod \ m_1, x_{-2} = seed[1] \ mod \ m_1, x_{-1} = seed[2] \ mod \ m_1$

$y_{-3} = seed[3] \ mod \ m_2, y_{-2} = seed[4] \ mod \ m_2, y_{-1} = seed[5] \ mod \ m_2$

if the values prove to be $x_{-3} = x_{-2} = x_{-1} = 0$, assume $x_{-3} = 1$

if the values prove to be $y_{-3} = y_{-2} = y_{-1} = 0$, assume $y_{-3} = 1$.

offset
Number of skipped elements.

```
mrg32k3a::mrg32k3a(std::initializer_list<std::uint32_t> seed, std::uint64_t offset = 0)
```

Input Parameters

seed
Initial conditions of the engine state.

offset
Number of skipped elements.

```
mrg32k3a::mrg32k3a(std::uint32_t seed, std::initializer_list<std::uint64_t> offset)
```

Input Parameters

seed
The initial conditions of the generator state, assume if $n = 0 : x_{-3} = x_{-2} = x_{-1} = y_{-3} = y_{-2} = y_{-1} = 1$

if $n = 1 : x_{-3} = seed[0] \ mod \ m_1, x_{-2} = x_{-1} = y_{-3} = y_{-2} = y_{-1} = 1$

if $n = 2 : x_{-3} = seed[0] \ mod \ m_1, x_{-2} = seed[1] \ mod \ m_1, x_{-1} = y_{-3} = y_{-2} = y_{-1} = 1$

if $n = 3 : x_{-3} = seed[0] \ mod \ m_1, x_{-2} = seed[1] \ mod \ m_1, x_{-1} = seed[2] \ mod \ m_1$

$y_{-3} = y_{-2} = y_{-1} = 1$

if $n = 4 : x_{-3} = seed[0] \ mod \ m_1, x_{-2} = seed[1] \ mod \ m_1, x_{-1} = seed[2] \ mod \ m_1$

$y_{-3} = seed[3] \ mod \ m_2, y_{-2} = y_{-1} = 1$

if $n = 5 : x_{-3} = seed[0] \ mod \ m_1, x_{-2} = seed[1] \ mod \ m_1, x_{-1} = seed[2] \ mod \ m_1$

$y_{-3} = seed[3] \ mod \ m_2, y_{-2} = seed[4] \ mod \ m_2, y_{-1} = 1$

if $n \geqslant 6 : x_{-3} = seed[0] \ mod \ m_1, x_{-2} = seed[1] \ mod \ m_1, x_{-1} = seed[2] \ mod \ m_1$

$y_{-3} = seed[3] \ mod \ m_2, y_{-2} = seed[4] \ mod \ m_2, y_{-1} = seed[5] \ mod \ m_2$

if the values prove to be $x_{-3} = x_{-2} = x_{-1} = 0$, assume $x_{-3} = 1$

if the values prove to be $y_{-3} = y_{-2} = y_{-1} = 0$, assume $y_{-3} = 1$.

offset
Number of skipped elements. Offset is calculated as: `num_to_skip` [0]+ `num_to_skip` [1]*2^{64} + `num_to_skip` [2]* 2^{128} + ... + `num_to_skip` [n-1]*2^{64} *(n-1).

```
mrg32k3a::mrg32k3a(std::initializer_list<std::uint32_t> seed, std::initializer_list
↪<std::uint64_t> offset)
```

Input Parameters

seed
Initial conditions of the engine state.

offset
Number of skipped elements. Offset is calculated as: `num_to_skip` [0]+ `num_to_skip` [1]*2^{64} + `num_to_skip` [2]* 2^{128} + … + `num_to_skip` [n-1]*2^{64} *(n-1).

Parent topic: *Device Engines (Basic Random Number Generators)*

philox4x32x10

A Philox4x32-10 counter-based pseudorandom number generator [*Salmon11*].

Description

The Philox4x32x10 engine is a keyed family of generator of counter-based BRNG. The state consists of 128-bit integer counter c and two 32-bits keys k_0 and k_1.

Generation algorithm

The generator has 32-bit integer output obtained in the following way [*Salmon11*]:

1. $c_n = c_{n-1} + 1$
2. $\omega_n = f(c_n)$**, where f is a function that takes 128-bit argument and returns a 128-bit number. The returned number is obtained as follows:**
   2.1. The argument c is interpreted as four 32-bit numbers $c = \overline{L_1 R_1 L_0 R_0}$, where $\overline{ABCD} = A \cdot 2^{96} + B \cdot 2^{64} + C \cdot 2^{32} + D$, put $k_0^0 = k_0, k_1^0 = k_1$.

   2.2. The following recurrence is calculated:

   $L_1^{i+1} = mullo(R_1^i, 0xD2511F53)$

   $R_1^{i+1} = mulhi(R_0^i, 0xCD9E8D57) \oplus k_0^i \oplus L_0^i$

   $L_0^{i+1} = mullo(R_0^i, 0xCD9E8D57)$

   $R_0^{i+1} = mulhi(R_1^i, 0xD2511F53) \oplus k_1^i \oplus L_1^i$

   $k_0^{i+1} = k_0^i + 0xBB67AE85$

   $k_1^{i+1} = k_1^i + 0x9E3779B9$, where $mulhi(a, b)$ and $mullo(a, b)$ are high and low parts of the $a \cdot b$ product respectively.

   2.3. Put $f(c) = \overline{L_1^N R_1^N L_0^N R_0^N}$, where $N = 10$
3. Integer output: $r_{4n+k} = \omega_n(k)$, where $\omega_n(k)$ is the k-th 32-bit integer in quadruple $\omega_n, k = 0, 1, 2, 3$
4. Real output: $u_n = (int) r_n / 2^{32} + 1/2$

class philox4x32x10

Syntax

```
namespace oneapi::mkl::rng::device {
  template<std::int32_t VecSize = 1>
  class philox4x32x10 {
  public:
    static constexpr std::uint64_t default_seed = 1;
    static constexpr std::int32_t vec_size = VecSize;

    philox4x32x10();
    philox4x32x10(std::uint64_t seed, std::uint64_t offset = 0);
    philox4x32x10(std::initializer_list<std::uint64_t> seed, std::uint64_t offset = 0);
    philox4x32x10(std::uint64_t seed, std::initializer_list<std::uint64_t> offset);
    philox4x32x10(std::initializer_list<std::uint64_t> seed, std::initializer_list
↪<std::uint64_t> offset);
  };
}
```

Class Template Parameters

VecSize

Describes the size of vector which will be produced by generate function by this engine. VecSize values may be 1, 2, 3, 4, 8, 16 as `sycl::vec` class size. By default VecSize = 1, for this case, a single random number is returned by the `generate` function.

Class Members

Routine	Description
philox4x32x10()	Default constructor
philox4x32x10(std::uint32_t seed, std::uint64_t offset = 0)	Constructor for common seed initialization of the engine and common number of skipped elements
philox4x32x10(std::initializer_list<std::uint32_t> seed, std::uint64_t offset = 0)	Constructor for extended seed initialization of the engine and common number of skipped elements
philox4x32x10(std::uint32_t seed, std::initializer_list<std::uint64_t> offset)	Constructor for common seed initialization of the engine and extended number of skipped elements
philox4x32x10(std::initializer_list<std::uint32_t> seed, std::initializer_list<std::uint64_t> offset)	Constructor for extended seed initialization of the engine and extended number of skipped elements

Constructors

```
philox4x32x10::philox4x32x10()
```

```
philox4x32x10::philox4x32x10(std::uint32_t seed, std::uint64_t offset = 0)
```

Input Parameters

seed
The initial conditions of the generator state, assume $k = seed, c = 0$, where k is a 64-bit key, c is a 128-bit counter.

offset
Number of skipped elements.

```
philox4x32x10::philox4x32x10(std::initializer_list<std::uint32_t> seed, std::uint64_t␣
↪offset = 0)
```

Input Parameters

seed
The initial conditions of the generator state, assume if $n = 0 : k = 0, c = 0$

if $n = 1 : k = seed[0], c = 0$

if $n = 2 : k = seed[0], c = seed[1]$

if $n = 3 : k = seed[0], c = seed[1] + seed[2] \cdot 2^{64}$

for $n > 3$ following arguments are ignored.

offset
Number of skipped elements.

```
philox4x32x10::philox4x32x10(std::uint32_t seed, std::initializer_list<std::uint64_t>␣
↪offset)
```

Input Parameters

seed
The initial conditions of the generator state, assume $k = seed, c = 0$, where k is a 64-bit key, c is a 128-bit counter.

offset
Number of skipped elements. Offset is calculated as: `num_to_skip` [0]+ `num_to_skip` [1]*2^{64} + `num_to_skip` [2]* 2^{128} + ... + `num_to_skip` [n-1]*2^{64} *(n-1).

```
philox4x32x10::philox4x32x10(std::initializer_list<std::uint32_t> seed, std::initializer_
↪list<std::uint64_t> offset)
```

Input Parameters

seed

The initial conditions of the generator state, assume if $n = 0 : k = 0, c = 0$

if $n = 1 : k = seed[0], c = 0$

if $n = 2 : k = seed[0], c = seed[1]$

if $n = 3 : k = seed[0], c = seed[1] + seed[2] \cdot 2^{64}$

for $n > 3$ following arguments are ignored.

offset

Number of skipped elements. Offset is calculated as: `num_to_skip` [0]+ `num_to_skip` [1]*2^{64} + `num_to_skip` [2]* 2^{128} + … + `num_to_skip` [n-1]*2^{64} *(n-1).

Parent topic: *Device Engines (Basic Random Number Generators)*

mcg31m1

The 31-bit multiplicative congruential pseudorandom number generator MCG($1132489760, 2^{32} - 1$) *[L'Ecuyer99a]*.

Description

The mcg31m1 engine is a 31-bit multiplicative congruential generator *[L'Ecuyer99]*. The mcg31m1 generator belongs to linear congruential generators with the period length of approximately 2^{31}. Such generators are still used as default random number generators in various software systems, mainly due to the simplicity of the portable versions implementation, speed, and compatibility with the earlier systems versions. However, their period length does not meet the requirements for modern basic generators. Still, the mcg31m1 generator possesses good statistic properties and you may successfully use it to generate random numbers of different distributions for small samplings.

Generation algorithm

$x_n = ax_{n-1}(mod\ m)$

$u_n = x_n/m$

$a = 1132489760, m = 2^{31} - 1$

class mcg31m1

Syntax

```
namespace oneapi::mkl::rng::device {
  template<std::int32_t VecSize = 1>
  class mcg31m1 {
  public:
    static constexpr std::uint32_t default_seed = 1;
    static constexpr std::int32_t vec_size = VecSize;

    mcg31m1();
    mcg31m1(std::uint32_t seed, std::uint64_t offset = 0);
```

(continues on next page)

(continued from previous page)

```
  };
}
```

Class Template Parameters

VecSize

Describes the size of vector which will be produced by generate function by this engine. VecSize values may be 1, 2, 3, 4, 8, 16 as `sycl::vec` class size. By default VecSize = 1, for this case, a single random number is returned by the `generate` function.

Class Members

Routine	Description
mcg31m1()	Default constructor
mcg31m1(std::uint32_t seed, std::uint64_t offset = 0)	Constructor for common seed initialization of the engine and common number of skipped elements

Constructors

```
mcg31m1::mcg31m1()
```

```
mcg31m1::mcg31m1(std::uint32_t seed, std::uint64_t offset = 0)
```

Input Parameters

seed

The initial conditions of the generator state, assume $x_0 = seed \ mod \ 0x7FFFFFFF$, if $x_0 = 0$, assume $x_0 = 1$.

offset

Number of skipped elements.

Parent topic: *Device Engines (Basic Random Number Generators)*

mcg59

The 59-bit multiplicative congruential pseudorandom number generator MCG($13^{13}, 2^{59}$) from NAG Numerical Libraries.

Description

The mcg59 engine is a 59-bit multiplicative congruential generator from NAG Numerical Libraries *NAG*. The mcg59 generator belongs to linear congruential generators with the period length of approximately 2^{57}.

Generation algorithm

$x_n = ax_{n-1}(mod\ m)$

$u_n = x_n/m$

$a = 13^{13}, m = 2^{59}$

class mcg59

Syntax

```
namespace oneapi::mkl::rng::device {
  template<std::int32_t VecSize = 1>
  class mcg59 {
  public:
    static constexpr std::uint32_t default_seed = 1;
    static constexpr std::int32_t vec_size = VecSize;

    mcg59();
    mcg59(std::uint64_t seed, std::uint64_t offset = 0);
  };
}
```

Class Template Parameters

VecSize

Describes the size of vector which will be produced by generate function by this engine. VecSize values may be 1, 2, 3, 4, 8, 16 as `sycl::vec` class size. By default VecSize = 1, for this case, a single random number is returned by the `generate` function.

Class Members

Routine	Description
mcg59()	Default constructor
mcg59(std::uint64_t seed, std::uint64_t offset = 0)	Constructor for common seed initialization of the engine and common number of skipped elements

Constructors

```
mcg59::mcg59()
```

```
mcg59::mcg59(std::uint64_t seed, std::uint64_t offset = 0)
```

Input Parameters

seed
The initial conditions of the generator state, assume $x_0 = seed \ mod \ 2^{59}$, if $x_0 = 0$, assume $x_0 = 1$.

offset
Number of skipped elements.

Parent topic: *Device Engines (Basic Random Number Generators)*

Device Distributions

oneMKL RNG routines are used to generate random numbers with different types of distributions. Each function group is introduced below by the type of underlying distribution and contains a short description of its functionality, as well as specifications of the call sequence and the explanation of input and output parameters. The Device Continuous Distribution Generators table and Device Discrete Distribution Generators table mention random number generator routines with data types and output distributions, and sets correspondence between data types of the generator routines and the basic random number generators.

Device Continuous Distribution Generators

Type of Distribution	Data Types	BRNG Data Type	Description
uniform (Continuous)	float, double	float, double	Uniform continuous distribution on the interval [a,b)
gaussian	float, double	float, double	Normal (Gaussian) distribution
exponential	float, double	float, double	Exponential distribution
lognormal	float, double	float, double	Lognormal distribution
beta	float, double	float, double	Beta distribution
gamma	float, double	float, double	Gamma distribution

Device Discrete Distribution Generators

Type of Distribution	Data Types	BRNG Data Type	Description
uniform (Discrete)	integer	float	Uniform discrete distribution on the interval [a,b)
bits	integer	integer	Bits of underlying BRNG integer sequence
uniform_bits	integer	integer	Uniformly distributed bits in 32/64-bit chunks
poisson	integer	integer	Poisson distribution
bernoulli	integer	integer	Bernoulli distribution

NOTE: In case of `integer` check desired distribution for supported data types.

Parent topic: *Random Number Generators Device Routines*

Distributions Template Parameter Method

Method Type	Distributions	Math Description
`uniform_method::standard` `uniform_method::accurate`	`uniform`	Standard method. `uniform_method::accurate` checks for additional `float` and `double` data types. For `integer` data types, it uses `double` as a `BRNG` data type (`float BRNG` data type is used in `uniform_method::standard` method on GPU).
`gaussian_method::box_muller2`	`gaussian`	Generates normally distributed random numbers *x1* and *x2* through the pair of uniformly distributed numbers *u1* and *u2* according to the formulas: $x_1 = \sqrt{-2 \ln u_1} \sin 2\pi u_2 x_2 = \sqrt{-2 \ln u_1} \cos 2\pi u_2$
`exponential_method::icdf` `exponential_method::icdf_accurate`	`exponential`	Inverse cumulative distribution function (ICDF) method.
`lognormal_method::box_muller2`	`lognormal`	Normally distributed random numbers *x1* and *x2* are produced through the pair of uniformly distributed numbers *u1* and *u2* according to the formulas: $x_1 = -2 \ln u_1 \sin 2\pi u_2$ $x_2 = -2 \ln u_1 \cos 2\pi u_2$ Then *x1* and *x2* are converted to lognormal distribution.
`bernoulli_method::icdf`	`bernoulli`	Inverse cumulative distribution function (ICDF) method.
`poisson_method::devroye`	`poisson`	Acceptance/rejection method for $\lambda \geq 27$ with decomposition into four regions: • Two parallelograms • Triangle • Left exponential tail • Right exponential tail

NOTE: Methods provided for exposition purposes.

uniform (Continuous)

Generates random numbers with uniform distribution.

Description

The class object is used in `generate` function to provide random numbers uniformly distributed over the interval [`a`, `b`), where `a`, `b` are the left and right bounds of the interval, respectively, and $a, b \in R; a < b$ `a`, `b`$\in$`R` ; `a` < `b`.

The probability density function is given by:

$$f_{a,b}(x) = \begin{cases} \frac{1}{b-a}, & x \in [a,b) \\ 1, & x \notin [a,b) \end{cases}, -\infty < x < +\infty$$

The cumulative distribution function is as follows:

$$F_{a,b}(x) = \begin{cases} 0, & x < a \\ \frac{x-a}{b-a}, & a \leq x < b \\ 1, & x \geq b \end{cases}, -\infty < x < +\infty$$

class uniform

Syntax

```
namespace oneapi::mkl::rng::device {
  template<typename Type = float, typename Method = uniform_method::by_default>
  class uniform {
  public:
    using method_type = Method;
    using result_type = Type;

    uniform();
    explicit uniform(Type a, Type b);

    Type a() const;
    Type b() const;
  };
}
```

Template parameters

typename RealType
: Type of the produced values. Supported types:

  - `float`
  - `double`

typename Method
: Generation method. The specific values are as follows:

  - `oneapi::mkl::rng::device::uniform_method::by_default`
  - `oneapi::mkl::rng::device::uniform_method::standard`
  - `oneapi::mkl::rng::device::uniform_method::accurate`

  See description of the methods in *Distributions methods template parameter*

Class Members

Routine	Description
uniform()	Default constructor
explicit uniform(RealType a, RealType b)	Constructor with parameters
RealType a() const	Method to obtain left bound a
RealType b() const	Method to obtain right bound b

Member types

```
uniform::method_type = Method
```

Description

The type which defines transformation method for generation.

```
uniform::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
uniform::uniform()
```

Description

Default constructor for distribution, parameters set as a = 0.0, b = 1.0.

```
explicit uniform::uniform(RealType a, RealType b)
```

Description

Constructor with parameters. a is a left bound, b is a right bound, assume $a < b$.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $a \geq b$

Characteristics

```
RealType uniform::a() const
```

Return Value

Returns the distribution parameter a - left bound.

```
RealType uniform::b() const
```

Return Value

Returns the distribution parameter b - right bound.

Parent topic: *Device Distributions*

gaussian

Generates normally distributed random numbers.

Description

The `gaussian` class object is used in the `generate` and function to provide random numbers with normal (Gaussian) distribution with mean (`a`) and standard deviation (`stddev`, σ), where $a, \sigma \in \mathbb{R}; \sigma > 0$

The probability density function is given by:

$$f_{a,\sigma}(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y-a)^2}{2\sigma^2}\right) dy, -\infty < x < +\infty$$

The cumulative distribution function is as follows:

$$F_{a,\sigma}(x) = \int_{-\infty}^{x} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y-a)^2}{2\sigma^2}\right) dy, -\infty < x < +\infty$$

The cumulative distribution function $F_{a,\sigma}(x)$ can be expressed in terms of standard normal distribution $\phi(x)$ as

$$F_{a,\sigma}(x) = \phi((x-a)/\sigma)$$

class gaussian

Syntax

```
namespace oneapi::mkl::rng::device {
  template<typename RealType, typename Method>
  class gaussian {
  public:
     using method_type = Method;
     using result_type = RealType;

     gaussian();
     explicit gaussian(RealType mean, RealType stddev);

     RealType mean() const;
     RealType stddev() const;
  };
}
```

Template parameters

typename RealType
Type of the produced values. Supported types:

- `float`
- `double`

typename Method
Generation method. The specific values are as follows:

- `oneapi::mkl::rng::device::gaussian_method::by_default`
- `oneapi::mkl::rng::device::gaussian_method::box_muller2`

See description of the methods in *Distributions methods template parameter*

Class Members

Routine	Description
gaussian()	Default constructor
explicit gaussian(RealType mean, RealType stddev)	Constructor with parameters
RealType mean() const	Method to obtain left bound *a*
RealType stddev() const	Method to obtain right bound *b*

Member types

```
gaussian::method_type = Method
```

Description

The type which defines transformation method for generation.

```
gaussian::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
gaussian::gaussian()
```

Description

Default constructor for distribution, parameters set as *mean* = 0.0, *stddev* = 1.0.

```
explicit gaussian::gaussian(RealType a, RealType b)
```

Description

Constructor with parameters. *mean* is a mean value, *stddev* is a standard deviation value.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $stddev \leq 0$

Characteristics

```
RealType gaussian::mean() const
```

Return Value

Returns the distribution parameter *mean* - mean value.

```
RealType gaussian::stddev() const
```

Return Value

Returns the distribution parameter *stddev* - standard deviation value.

Parent topic: *Device Distributions*

lognormal

Generates lognormally distributed random numbers.

Description

The `lognormal` class object is used in the `generate` and function to provide random numbers with average of distribution (`m`, `a`) and standard deviation (`s`, σ) of subject normal distribution, displacement (`displ`, `b`), and scalefactor (`scale`, β), where $a, \sigma, b, \beta \in \mathbb{R}; \sigma > 0, \beta > 0$.

The probability density function is given by:

$$f_{a,\sigma,b,\beta}(x) = \begin{cases} \frac{1}{\sigma(x-b)\sqrt{2\pi}} \exp\left(-\frac{\ln(\frac{x-b}{\beta})-a)^2}{2\sigma^2}\right), & x > b \\ 0, & x \leq b \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,\sigma,b,\beta}(x) = \begin{cases} \Phi\left(\frac{\ln(\frac{x-b}{\beta})-a}{\sigma}\right), & x > b \\ 0, & x \leq b \end{cases}$$

class lognormal

Syntax

```
namespace oneapi::mkl::rng::device {
  template<typename RealType, typename Method>
  class lognormal {
  public:
     using method_type = Method;
     using result_type = RealType;

     lognormal();
     explicit lognormal(RealType m, RealType s, RealType displ = (RealType)0.0, RealType␣
↪scale = (RealType)1.0);

     RealType m() const;
     RealType s() const;
```

(continues on next page)

(continued from previous page)

```
    RealType displ() const;
    RealType scale() const;
  };
}
```

Template parameters

typename RealType
Type of the produced values. Supported types:

- `float`
- `double`

typename Method = oneapi::mkl::rng::lognormal_method::by_default
Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::device::lognormal_method::by_default`
- `oneapi::mkl::rng::device::lognormal_method::box_muller2`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
lognormal()	Default constructor
explicit lognormal(RealType m, RealType s, RealType displ = (RealType)0.0, RealType scale = (RealType)1.0)	Constructor with parameters
RealType m() const	Method to obtain mean value
RealType s() const	Method to obtain standard deviation value
RealType displ() const	Method to obtain displacement value
RealType scale() const	Method to obtain scalefactor value

Member types

```
lognormal::method_type = Method
```

Description

The type which defines transformation method for generation.

```
lognormal::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
lognormal::lognormal()
```

Description

Default constructor for distribution, parameters set as m = 0.0, s = 1.0, *displ* = 0.0, *scale* = 1.0.

```
explicit lognormal::lognormal(RealType m, RealType s, RealType displ = (RealType)0.0,␣
↪RealType scale = (RealType)1.0)
```

Description

Constructor with parameters. m is a mean value, s is a standard deviation value, *displ* is a displacement value, *scale* is a scalefactor value.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $s \leq 0$, or $scale \leq 0$

Characteristics

```
RealType lognormal::m() const
```

Return Value

Returns the distribution parameter m - mean value.

```
RealType lognormal::s() const
```

Return Value

Returns the distribution parameter *s* - standard deviation value.

```
RealType lognormal::displ() const
```

Return Value

Returns the distribution parameter *displ* - displacement value.

```
RealType lognormal::scale() const
```

Return Value

Returns the distribution parameter *scale* - scalefactor value.

Parent topic: *Device Distributions*

exponential

Generates exponentially distributed random numbers.

Description

The `exponential` class object is used in the `generate` function to provide random numbers with exponential distribution that has displacement a and scalefactor β, where $a, \beta \in R; \beta > 0$.

The probability density function is given by:

$$f_{a,\beta}(x) = \begin{cases} \frac{1}{\beta}\exp(-\frac{(x-a)}{\beta}), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \begin{cases} 1 - \exp(-\frac{(x-a)}{\beta}), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty$$

class exponential

Syntax

```
namespace oneapi::mkl::rng::device {
  template<typename RealType, typename Method>
  class exponential {
  public:
    using method_type = Method;
    using result_type = RealType;

    exponential();
```

(continues on next page)

(continued from previous page)

```
    explicit exponential(RealType a, RealType beta);

    RealType a() const;
    RealType beta() const;
  };
}
```

Template parameters

typename RealType
Type of the produced values. Supported types:

- `float`
- `double`

typename Method = oneapi::mkl::rng::exponential_method::by_default
Generation method. The specific values are as follows:

- `oneapi::mkl::rng::device::exponential_method::by_default`
- `oneapi::mkl::rng::device::exponential_method::icdf`
- `oneapi::mkl::rng::device::exponential_method::icdf_accurate`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
exponential()	Default constructor
explicit exponential(RealType a, RealType beta)	Constructor with parameters
RealType a() const	Method to obtain displacement value
RealType beta() const	Method to obtain scalefactor

Member types

```
exponential::method_type = Method
```

Description

The type which defines transformation method for generation.

```
exponential::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
exponential::exponential()
```

Description

Default constructor for distribution, parameters set as *a* = 0.0, *beta* = 1.0.

```
explicit exponential::exponential(RealType a, RealType beta)
```

Description

Constructor with parameters. *a* is a displacement, *beta* is a scalefactor.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $beta \leq 0$

Characteristics

```
RealType exponential::a() const
```

Return Value

Returns the distribution parameter *a* - displacement.

```
RealType exponential::beta() const
```

Return Value

Returns the distribution parameter *beta* - scalefactor value.

Parent topic: *Device Distributions*

uniform (Discrete)

Generates random numbers uniformly distributed over the interval `[a, b)`.

Description

The `uniform` class object is used in `generate` and function to provide random numbers uniformly distributed over the interval `[a, b)`, where `a, b` are the left and right bounds of the interval respectively, and $a, b \in Z; a < b$.

The probability distribution is given by:

$$P(X = k) = \frac{1}{b-a}, k \in \{a, a+1, \ldots, b-1\}$$

The cumulative distribution function is as follows:

$$F_{a,b}(x) = \begin{cases} 0, & x < a \\ \frac{x-a+1}{b-a}, & a \le x < b \\ 1, & x \ge b \end{cases}, x \in R$$

class uniform

Syntax

```
namespace oneapi::mkl::rng::device {
  template<typename Type, typename Method>
  class uniform<Type, Method> {
  public:
    using method_type = Method;
    using result_type = Type;

    uniform();
    explicit uniform(Type a, Type b);

    Type a() const;
    Type b() const;
  };
}
```

Template parameters

typename Type
: Type of the produced values. Supported types:

  - `std::int32_t`
  - `std::uint32_t`

typename Method = oneapi::mkl::rng::uniform_method::by_default
: Transformation method, which will be used for generation. Supported types:

  - `oneapi::mkl::rng::device::uniform_method::by_default`
  - `oneapi::mkl::rng::device::uniform_method::standard`

- `oneapi::mkl::rng::device::uniform_method::accurate`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
uniform()	Default constructor
explicit uniform(Type a, Type b)	Constructor with parameters
Type a() const	Method to obtain left bound a
Type b() const	Method to obtain right bound b

Constructors

```
uniform::uniform()
```

Description

Default constructor for distribution, parameters set as $a = 0$, $b = (1 << 23)$ with `uniform_method::standard` or std::numeric_limits<Type>::max() with `uniform_method::accurate`.

```
explicit uniform::uniform(Type a, Type b)
```

Description

Constructor with parameters. a is a left bound, b is a right bound, assume $a < b$.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $a \geq b$

Characteristics

```
uniform::a() const
```

Return Value

Returns the distribution parameter a - left bound.

```
uniform::b() const
```

Return Value

Returns the distribution parameter *b* - right bound.

Parent topic: *Device Distributions*

bits

Generates bits of underlying engine (BRNG) integer sequence.

Description

The `bits` class object is used in `generate` and function to provide integer random values. Each integer can be treated as a vector of several bits. In pseudorandom generators this randomness can be violated. See *VS Notes* for details.

class bits

Syntax

```
namespace oneapi::mkl::rng::device {
  template<typename UIntType = std::uint32_t>
  class bits {
    using result_type = UIntType;
  };
}
```

Template parameters

typename UIntType

Type of the produced values. Supported types:

- `std::uint32_t` for philox4x32x10, mrg32k3a and mcg31m1 engines.
- `std::uint64_t` for mcg59.

Member types

```
bits::result_type = UIntType
```

Description

The type which defines type of generated random numbers.

Parent topic: *Device Distributions*

uniform_bits

Generates uniformly distributed bits in 32/64-bit chunks.

Description

The `uniform_bits` class object is used in `generate` and function to generate uniformly distributed bits in 32/64-bit chunks. It is designed to ensure each bit in the 32/64-bit chunk is uniformly distributed. This distribution is supported for philox4x32x10 and mcg59 engines. When generating 64-bit chunks, twice as much engine offset needs to be provided.

`UIntType` denotes the chunk size and can be `std::uint32_t`, `std::uint64_t`. See *VS Notes* for details.

class uniform_bits

Syntax

```
namespace oneapi::mkl::rng::device {
  template<typename UIntType = std::uint32_t>
  class uniform_bits {
    using result_type = UIntType;
  };
}
```

Template parameters

typename UIntType

Type of the produced values. Supported types:

- `std::uint32_t`
- `std::uint64_t`

Member types

```
uniform_bits::result_type = UIntType
```

Description

The type which defines type of generated random numbers.

Parent topic: *Device Distributions*

poisson

Generates Poisson distributed random values.

Description

The `poisson` class object is used in the `generate` and function to provide Poisson distributed random numbers with distribution parameter λ, where $\lambda \in R; \lambda > 0$.

The probability distribution is given by:

$$P(X = k) = \frac{\lambda^k e^{-\lambda}}{k!}$$

$k \in \{0, 1, 2, \ldots\}$.

The cumulative distribution function is as follows:

$$F_\lambda(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} \frac{\lambda^k e^{-\lambda}}{k!}, & x \geq 0 \\ 0, & x < 0 \end{cases}, x \in R$$

class poisson

Syntax

```
namespace oneapi::mkl::rng::device {
  template<typename IntType, typename Method>
  class poisson {
  public:
    using method_type = Method;
    using result_type = IntType;

    poisson();
    explicit poisson(double lambda);

    double lambda() const;
  };
}
```

Template parameters

typename IntType

Type of the produced values. Supported types:

- `std::int32_t`
- `std::uint32_t`

typename Method = oneapi::mkl::rng::poisson_method::by_default

Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::device::poisson_method::by_default`
- `oneapi::mkl::rng::device::poisson_method::devroye`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
poisson()	Default constructor
explicit poisson(double lambda)	Constructor with parameters
double lambda() const	Method to obtain distribution parameter

Member types

```
poisson::method_type = Method
```

Description

The type which defines transformation method for generation.

```
poisson::result_type = IntType
```

Description

The type which defines type of generated random numbers.

Constructors

```
poisson::poisson()
```

Description

Default constructor for distribution, parameters set as *lambda* = 0.5.

```
explicit poisson::poisson(double lambda)
```

Description

Constructor with parameters. *lambda* is a distribution parameter.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $lambda \leq 0$

Characteristics

```
double poisson::lambda() const
```

Return Value

Returns the distribution parameter *lambda*.

Parent topic: *Device Distributions*

bernoulli

Generates Bernoulli distributed random values.

Description

The `bernoulli` class object is used in the `generate` and function to provide Bernoulli distributed random numbers with probability `p` of a single trial success, where $p \in R; 0 \leq p \leq 1$.

The probability distribution is given by:

$$P(X = 1) = p$$

$$P(X = 0) = 1 - p$$

The cumulative distribution function is as follows:

$$F_p(x) = \begin{cases} 0, & x < 0 \\ 1 - p, & 0 \leq x < 1, x \in R \\ 1, & x \geq 1 \end{cases}$$

class bernoulli

Syntax

```
namespace oneapi::mkl::rng::device {
  template<typename IntType, typename Method>
  class bernoulli {
  public:
    using method_type = Method;
    using result_type = IntType;

    bernoulli();
    explicit bernoulli(float p);
```

(continues on next page)

(continued from previous page)

```
    float p() const;
  };
}
```

Template parameters

typename IntType

Type of the produced values. Supported types:

- `std::int32_t`
- `std::uint32_t`

typename Method = oneapi::mkl::rng::bernoulli_method::by_default

Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::bernoulli_method::by_default`
- `oneapi::mkl::rng::bernoulli_method::icdf`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
bernoulli()	Default constructor
explicit bernoulli(float p)	Constructor with parameters
float p() const	Method to obtain probability *p*

Member types

```
bernoulli::method_type = Method
```

Description

The type which defines transformation method for generation.

```
bernoulli::result_type = IntType
```

Description

The type which defines type of generated random numbers.

Constructors

```
bernoulli::bernoulli()
```

Description

Default constructor for distribution, parameters set as p = 0.5f.

```
explicit bernoulli::bernoulli(float p)
```

Description

Constructor with parameters. p is a probability.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $p > 1$, or $p < 0$

Characteristics

```
float bernoulli::p() const
```

Return Value

Returns the distribution parameter p - probability.

Parent topic: *Device Distributions*

beta

Generates beta distributed random numbers.

Description

The `beta` class object is used in the `generate` function to provide random numbers with beta distribution that has shape parameters p and q, displacement α and scale parameter (b, β), where p, q. $\alpha, \beta \in R; p > 0; q > 0; \beta > 0$.

The probability distribution is given by:

$$f_{p,q,\alpha,\beta}(x) = \begin{cases} \frac{1}{B(p,q)*\beta^{p+q-1}}(x-a)^{p-1} * (\beta+\alpha-x)^{q-1}, \alpha \le x < \alpha+\beta \\ 0, x < \alpha, x \ge \alpha+\beta \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,b}(x) = \begin{cases} 0, x < \alpha \\ \int_{\alpha}^{x} \frac{1}{B(p,q)*\beta^{p+q-1}}(y-\alpha)^{p-1} * (\beta+\alpha-y)^{q-1}dy, \alpha \le x < \alpha+\beta, x \in R \\ 1, x \ge \alpha+\beta \end{cases}$$

Where $B(p,1)$ is the complete beta function.

class beta

Syntax

```
namespace oneapi::mkl::rng::device {
  template<typename RealType, typename Method>
  class beta {
  public:
    using method_type = Method;
    using result_type = RealType;

    beta();
    explicit beta(RealType p, RealType q, RealType a, RealType b);

    RealType p() const;
    RealType q() const;
    RealType a() const;
    RealType b() const;
    std::size_t count_rejected_numbers() const;
  };
}
```

Template parameters

typename RealType
: Type of the produced values. Supported types:

  - `float`
  - `double`

typename Method
: Generation method. The type is unspecified.

Class Members

Routine	Description
beta()	Default constructor
explicit beta(RealType p, RealType q, RealType a, RealType b)	Constructor with parameters
RealType p() const	Method to obtain shape `p`
RealType q() const	Method to obtain shape `q`
RealType a() const	Method to obtain displacement α
RealType b() const	Method to obtain scale parameter β
size_t count_rejected_numbers() const	Method to obtain amount of random numbers that were rejected during the last `generate` function call. If no `generate` calls, `0` is returned.

Member types

```
beta::method_type = Method
```

Description

The type which defines transformation method for generation.

```
beta::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
beta::beta()
```

Description

Default constructor for distribution, parameters set as `p` = 1.0, `q` = 1.0, α = 0.0, β = 1.0.

```
explicit beta::beta(RealType p, RealType q, RealType a, RealType b)
```

Description

Constructor with parameters. `p` and `q` are shapes, α is a displacement, β is a scale parameter.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $p \leq 0$, or $q \leq 0$, or $\beta \leq 0$

Characteristics

```
RealType beta::p() const
```

Return Value

Returns the distribution parameter `p` - shape.

```
RealType beta::q() const
```

Return Value

Returns the distribution parameter `q` - shape.

```
RealType beta::a() const
```

Return Value

Returns the distribution parameter α - displacement.

```
RealType beta::b() const
```

Return Value

Returns the distribution parameter β - scale parameter value.

```
std::size_t beta::count_rejected_numbers() const
```

Return Value

Returns the amount of random numbers that were rejected during the last `generate` function call. If no `generate` calls, `0` is returned.

Parent topic: *Device Distributions*

gamma

Generates gamma distributed random numbers.

Description

The `gamma` class object is used in the `generate` function to provide random numbers with gamma distribution that has shape α, displacement a, and scale parameter β, where $a, \alpha, \beta \in R; \alpha > 0; \beta > 0$.

The probability distribution is given by:

$$f_{a,\alpha,\beta}(x) = \begin{cases} \frac{1}{\Gamma(\alpha)\beta^{\alpha}}(x-a)^{\alpha-1}e^{-(x-a)/\beta}, x \geq a \\ 0, x < a \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,\alpha,\beta}(x) = \begin{cases} \int_a^x \frac{1}{\Gamma(\alpha)\beta^{\alpha}}(y-a)^{\alpha-1}e^{-(y-a)/\beta}dy, x \geq a \\ 0, x < a \end{cases}$$

class gamma

Syntax

```
namespace oneapi::mkl::rng::device {
  template<typename RealType, typename Method>
  class gamma {
  public:
    using method_type = Method;
    using result_type = RealType;

    gamma();
    explicit gamma(RealType alpha, RealType a, RealType beta);

    RealType alpha() const;
    RealType a() const;
    RealType beta() const;
    std::size_t count_rejected_numbers() const;
  };
}
```

Template parameters

typename RealType
Type of the produced values. Supported types:

- `float`
- `double`

typename Method
Generation method. The type is unspecified.

Class Members

Routine	Description
gamma()	Default constructor
explicit gamma(RealType alpha, RealType a, RealType beta)	Constructor with parameters
RealType alpha() const	Method to obtain shape value
RealType a() const	Method to obtain displacement value
RealType beta() const	Method to obtain scale parameter
size_t count_rejected_numbers() const	Method to obtain amount of random numbers that were rejected during the last `generate` function call. If no `generate` calls, `0` is returned.

Member types

```
gamma::method_type = Method
```

Description

The type which defines transformation method for generation.

```
gamma::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
gamma::gamma()
```

Description

Default constructor for distribution, parameters set as `alpha` = 1.0, `a` = 0.0, `beta` = 1.0.

```
explicit gamma::gamma(RealType alpha, RealType a, RealType beta)
```

Description

Constructor with parameters. `alpha` is a shape, `a` is a displacement, `beta` is a scale parameter.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $alpha \leq 0$ or $beta \leq 0$

Characteristics

```
RealType gamma::alpha() const
```

Return Value

Returns the distribution parameter `alpha` - shape.

```
RealType gamma::a() const
```

Return Value

Returns the distribution parameter `a` - displacement.

```
RealType gamma::beta() const
```

Return Value

Returns the distribution parameter `beta` - scale parameter value.

```
std::size_t gamma::count_rejected_numbers() const
```

Return Value

Returns the amount of random numbers that were rejected during the last `generate` function call. If no `generate` calls, `0` is returned.

Parent topic: *Device Distributions*

Device Service Routines

Routine	Description
skip_ahead	Proceed state of engine by the skip-ahead method to skip a given number of elements from the original sequence.

skip_ahead

Description

Proceed state of engine by the skip-ahead method.

The `skip_ahead` function supports the following interfaces to apply the skip-ahead method:

- Common interface
- Interface with a partitioned number of skipped elements

skip_ahead

Common Interface

```
namespace oneapi::mkl::rng::device {
  template<typename Engine>
  void skip_ahead (Engine& engine, std::uint64_t num_to_skip)
}
```

Template Parameters

Engine
: Object of engine class, which supports the block-splitting method.

Input Parameters

engine
: Engine which state would be skipped.

num_to_skip
: Number of skipped elements.

Interface with a partitioned number of skipped elements

```
namespace oneapi::mkl::rng::device {
  template<typename Engine>
  void skip_ahead (Engine& engine, std::initializer_list<std::uint64_t> num_to_skip)
}
```

Template Parameters

Engine
Object of engine class, which supports the block-splitting method.

Input Parameters

engine
Engine which state would be skipped.

num_to_skip
Partitioned number of skipped elements. The total number of skipped elements would be: $num\_to\_skip[0] + num\_to\_skip[1] \cdot 2^{64} + ... + num\_to\_skip[1] \cdot 2^{64(n-1)}$, where *n* is a number of elements in *num_to_skip* list.

Parent topic: *Device Service Routines*

Parent topic: *Random Number Generators Device Routines*

Parent topic: *Random Number Generators*

Parent topic: *Random Number Generators*

9.2.5 Summary Statistics

The oneMKL provides a set of *Summary Statistics routines* that compute basic statistical estimates for single and double precision multi-dimensional datasets.

Summary Statistics

Definitions

The oneMKL Summary Statistics domains consists of:

- Dataset structure. The structure consolidates the information of a multi-dimensional dataset (see detailed description in *dataset*).
- Computation routines. The routines are represented as free functions (see detailed description for each routine in *Summary Statistics Routines*):
  - Raw and central sums / moments up to the fourth order
  - Variation coefficient
  - Skewness and excess kurtosis (further referred as kurtosis)
  - Minimum and maximum

Refer to *oneMKL Summary Statistics Usage Model*.

oneMKL Summary Statistics Usage Model

Description

A typical algorithm for summary statistics is as follows:

1. Create and initialize an object for dataset.
2. Call the summary statistics routine to calculate the appropriate estimate.

The following example demonstrates how to calculate mean values for a 3-dimensional dataset filled with random numbers. For dataset creation, the *make_dataset* helper function is used.

USM-based example

```
#include "oneapi/mkl/stats.hpp"

int main() {
    sycl::queue queue;

    constexpr std::size_t n_observations = 1000;
    constexpr std::size_t n_dims = 3;

    // allocate Unified Shared Memory for the dataset of the size n_observations * n_
↪dims and fill it with any data
    // allocate Unified Shared Memory for the mean output of the size n_dims

    // create oneapi::mkl::stats::dataset
    auto dataset = oneapi::mkl::stats::make_dataset<oneapi::mkl::stats::layout::row_
↪major>(n_dims, n_observations, dataset_ptr);

    // call statistics computation routine
    auto event = oneapi::mkl::stats::mean(queue, dataset, mean_ptr);

    // wait until computations are completed
    event.wait();

    // ...
}
```

Parent topic: *Summary Statistics*

dataset

The structure consolidates the information of a multi-dimensional dataset.

Description

The `dataset` struct object is used in *Summary Statistics Routines* as a multi-dimensional data storage. `dataset` struct contains information about observations matrix and its size (dimensions x observations), observations weights and indices for dimensions (defines dimensions to be processed).

structure dataset (Buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<layout ObservationsLayout, typename Type>
    struct dataset<ObservationsLayout, sycl::buffer<Type, 1>> {

    explicit dataset(std::int64_t n_dims_, std::int64_t n_observations_,
                sycl::buffer<Type, 1> observations_, sycl::buffer<Type, 1> weights_ = {0}
↪,
                sycl::buffer<std::int64_t, 1> indices_ = {0});

    std::int64_t n_dims;
    std::int64_t n_observations;
    sycl::buffer<Type, 1> observations;
    sycl::buffer<Type, 1> weights = {0};
    sycl::buffer<std::int64_t, 1> indices = {0};
    static constexpr layout layout = ObservationsLayout;
    };
}
```

Template parameters

typename Type
Type of the multi-dimensional data. Supported types:

- `float`
- `double`

oneapi::mkl::stats::layout ObservationsLayout
Type of the multi-dimensional data layout. Supported types:

- `oneapi::mkl::stats::layout::row_major`
- `oneapi::mkl::stats::layout::col_major`

Struct Members

Routine	Description
explicit dataset(std::int64_t n_dims_, std::int64_t n_observations_, sycl::buffer<Type, 1> observations_, sycl::buffer<Type, 1> weights_ = {0}, sycl::buffer<std::int64_t, 1> indices_ = {0})	Constructor

Constructors

```
explicit dataset::dataset(std::int64_t n_dims_, std::int64_t n_observations_,
    sycl::buffer<Type, 1> observations_,
    sycl::buffer<Type, 1> weights_ = {0},
    sycl::buffer<std::int64_t, 1> indices_ = {0});
```

Description

Constructor with parameters.

- *n_dims_* is the number of dimensions
- *n_observations_* is the number of observations
- *observations_* is the matrix of observations
- *weights_* is an optional parameter, represents an array of weights for observations (of size *n_observations*). If the parameter is not specified, each observation is assigned a weight equal 1.
- *indices_* is an optional parameter, represents an array of dimensions that are processed (of size *n_dims*). If the parameter is not specified, all dimensions are processed.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $n\_dims\_ \leq 0$, or $n\_observations\_ \leq 0$, or *observations_.get_count()* == *0*

structure dataset (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<layout ObservationsLayout, typename Type>
    struct dataset<Type*, ObservationsLayout> {
    explicit dataset(std::int64_t n_dims_, std::int64_t n_observations_, Type*␣
↪observations_,
                Type* weights_ = nullptr, std::int64_t* indices_ = nullptr);

    std::int64_t n_dims;
```

(continues on next page)

(continued from previous page)

```
    std::int64_t n_observations;
    Type* observations;
    Type* weights = nullptr;
    std::int64_t* indices = nullptr;
    static constexpr layout layout = ObservationsLayout;
    };
}
```

Template parameters

typename Type
Type of the multi-dimensional data. Supported types:

- `float`
- `double`

oneapi::mkl::stats::layout ObservationsLayout
Type of the multi-dimensional data layout. Supported types:

- `oneapi::mkl::stats::layout::row_major`
- `oneapi::mkl::stats::layout::col_major`

Struct Members

Routine	Description
explicit dataset(std::int64_t n_dims_, std::int64_t n_observations_, Type observations_, Type* weights_ = nullptr, std::int64_t* indices_ = nullptr)*	Constructor

Constructors

```
explicit dataset::dataset(std::int64_t n_dims_, std::int64_t n_observations_,
    Type* observations_,
    Type* weights_ = nullptr,
    std::int64_t* indices_ = nullptr);
```

Description

Constructor with parameters.

- *n_dims_* is the number of dimensions
- *n_observations_* is the number of observations
- *observations_* is the matrix of observations
- *weights_* is an optional parameter, represents an array of weights for observations (of size *n_observations*). If the parameter is not specified, each observation is assigned a weight equal 1.

- *indices_* is an optional parameter, represents an array of dimensions that are processed (of size *n_dims*). If the parameter is not specified, all dimensions are processed.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when *n_dims_* ≤ 0, or *n_observations_* ≤ 0, or *observations_* == *nullptr*

Parent topic: *Summary Statistics*

Summary Statistics Routines

The oneMKL Summary Statistics routines calculate next estimates:

Routine	Description
raw_sum	Raw sums up to the fourth order
central_sum	Central sums up to the fourth order
central_sum with provided mean	Central sums up to the fourth order with provided mean
mean	Mean value
raw_moment	Raw moments up to the fourth order
central_moment	Central moments up to the fourth order
central_moment with provided mean	Central moments up to the fourth order with provided mean
variation	Variation coefficient
variation with provided mean	Variation coefficient with provided mean
skewness	Skewness value
skewness with provided mean	Skewness value with provided mean
kurtosis	Kurtosis value
kurtosis with provided mean	Kurtosis value with provided mean
min	Min value
max	Max value
min_max	Min and max values

Parent topic: *Summary Statistics*

raw_sum

Entry point to compute raw sums up to the 4th order.

Description and Assumptions

The oneapi::mkl::stats::raw_sum function is used to compute an array of raw sums up to the 4th order (raw sums for each dataset's dimension).

raw_sum supports the following precisions for data:

T
`float`
`double`

raw_sum (Buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    void raw_sum(sycl::queue& queue,
    const dataset<ObservationsLayout, sycl::buffer<Type, 1>>& data,
    sycl::buffer<Type, 1> sum,
    sycl::buffer<Type, 1> raw_sum_2 = {0},
    sycl::buffer<Type, 1> raw_sum_3 = {0},
    sycl::buffer<Type, 1> raw_sum_4 = {0});
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

data
Dataset which is used for computation.

Output Parameters

sum
sycl::buffer array of sum values.

raw_sum_2
Optional parameter. sycl::buffer array of 2nd order raw sum values.

raw_sum_3
Optional parameter. sycl::buffer array of 3rd order raw sum values.

raw_sum_4
Optional parameter. sycl::buffer array of 4th order raw sum values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when sum.get_count() == 0 & raw_sum_2.get_count() == 0 & raw_sum_3.get_count() == 0 & raw_sum_4.get_count() == 0, or dataset object is invalid

raw_sum (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    sycl::event raw_sum(sycl::queue& queue,
    const dataset<ObservationsLayout, Type*>& data,
    Type* sum,
    Type* raw_sum_2 = nullptr,
    Type* raw_sum_3 = nullptr,
    Type* raw_sum_4 = nullptr,
    const std::vector<sycl::event> &dependencies = {});
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

data
Dataset which is used for computation.

dependencies
Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

sum
Pointer to the array of sum values.

raw_sum_2
Optional parameter. Pointer to the array of the 2nd order raw sum values.

raw_sum_3
Optional parameter. Pointer to the array of the 3rd order raw sum values.

raw_sum_4
Optional parameter. Pointer to the array of the 2nd order raw sum values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when sum == nullptr & raw_sum_2 == nullptr & raw_sum_3 == nullptr & raw_sum_4 == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

central_sum

Entry point to compute central sums up to the 4th order.

Description and Assumptions

The oneapi::mkl::stats::central_sum function is used to compute an array of central sums up to the 4th order (central sums for each dataset's dimension).

central_sum supports the following precisions for data:

T
float
double

central_sum (Buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    void central_sum(sycl::queue& queue,
    const dataset<ObservationsLayout, sycl::buffer<Type, 1>>& data,
    sycl::buffer<Type, 1> central_sum_2,
```

(continues on next page)

(continued from previous page)

```
    sycl::buffer<Type, 1> central_sum_3 = {0},
    sycl::buffer<Type, 1> central_sum_4 = {0});
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

data
Dataset which is used for computation.

Output Parameters

central_sum_2
sycl::buffer array of 2nd order central sum values.

central_sum_3
Optional parameter. sycl::buffer array of 3rd order central sum values.

central_sum_4
Optional parameter. sycl::buffer array of 4th order central sum values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when central_sum_2.get_count() == 0 & central_sum_3.get_count() == 0 & central_sum_4.get_count() == 0, or dataset object is invalid

central_sum (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    sycl::event central_sum(
    sycl::queue& queue,
    const dataset<ObservationsLayout, Type*>& data,
    Type* central_sum_2,
    Type* central_sum_3 = nullptr,
    Type* central_sum_4 = nullptr,
    const std::vector<sycl::event> &dependencies = {});
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

data
Dataset which is used for computation.

dependencies
Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

central_sum_2
Pointer to the array of the 2nd order central sum values.

central_sum_3
Optional parameter. Pointer to the array of the 3rd order central sum values.

central_sum_4
Optional parameter. Pointer to the array of the 2nd order central sum values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when central_sum_2 == nullptr & central_sum_3 == nullptr & central_sum_4 == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

central_sum with provided mean

Entry point to compute central sums up to the 4th order with the provided mean values.

Description and Assumptions

The oneapi::mkl::stats::central_sum function is used to compute an array of central sums up to the 4th order (central sums for each dataset's dimension) with the provided mean values.

central_sum with provided mean supports the following precisions for data:

T
float
double

central_sum with provided mean (buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    void central_sum(sycl::queue& queue,
    sycl::buffer<Type, 1> mean,
    const dataset<ObservationsLayout, sycl::buffer<Type, 1>>& data,
    sycl::buffer<Type, 1> central_sum_2,
    sycl::buffer<Type, 1> central_sum_3 = {0},
    sycl::buffer<Type, 1> central_sum_4 = {0});
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

mean
sycl::buffer to the array of provided mean values.

data
Dataset which is used for computation.

Output Parameters

central_sum_2
sycl::buffer array of 2nd order central sum values.

central_sum_3
Optional parameter. sycl::buffer array of 3rd order central sum values.

central_sum_4
Optional parameter. sycl::buffer array of 4th order central sum values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when central_sum_2.get_count() == 0 & central_sum_3.get_count() == 0 & central_sum_4.get_count() == 0, or mean.get_count() == 0, or dataset object is invalid

central_sum with provided mean (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    sycl::event central_sum(sycl::queue& queue,
    Type* mean,
    const dataset<ObservationsLayout, Type*>& data,
    Type* central_sum_2,
    Type* central_sum_3 = nullptr,
    Type* central_sum_4 = nullptr,
```

(continues on next page)

(continued from previous page)

```
    const std::vector<sycl::event> &dependencies = {});
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

mean
Pointer to the array of provided mean values.

data
Dataset which is used for computation.

dependencies
Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

central_sum_2
Pointer to the array of the 2nd order central sum values.

central_sum_3
Optional parameter. Pointer to the array of the 3rd order central sum values.

central_sum_4
Optional parameter. Pointer to the array of the 2nd order central sum values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when central_sum_2 == nullptr & central_sum_3 == nullptr & central_sum_4 == nullptr, or mean == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

mean

Entry point to compute mean values.

Description and Assumptions

The oneapi::mkl::stats::mean function is used to compute a mean array (mean value for each dataset's dimension).

mean supports the following precisions for data:

T
float
double

mean (buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    void mean(sycl::queue& queue,
    const dataset<ObservationsLayout, sycl::buffer<Type, 1>>& data,
    sycl::buffer<Type, 1> mean);
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

data
Dataset which is used for computation.

Output Parameters

mean
sycl::buffer array of mean values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when mean.get_count() == 0, or dataset object is invalid

mean (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    sycl::event mean(sycl::queue& queue,
    const dataset<ObservationsLayout, Type*>& data,
    Type* mean,
    const std::vector<sycl::event> &dependencies = {});
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

data
Dataset which is used for computation.

dependencies
Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

mean
Pointer to the array of mean values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when mean == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

raw_moment

Entry point to compute raw moments up to the 4th order.

Description and Assumptions

The oneapi::mkl::stats::raw_moment function is used to compute array of raw moments up to the 4th order (raw moments for each dataset's dimension).

raw_moment supports the following precisions for data:

T
float
double

oneapi::mkl::stats::raw_moment (buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = fast, typename Type, layout ObservationsLayout>
    void raw_moment(sycl::queue& queue,
    const dataset<ObservationsLayout, sycl::buffer<Type, 1>>& data,
    sycl::buffer<Type, 1> mean,
    sycl::buffer<Type, 1> raw_moment_2 = {0},
    sycl::buffer<Type, 1> raw_moment_3 = {0},
    sycl::buffer<Type, 1> raw_moment_4 = {0});
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

data
Dataset which is used for computation.

Output Parameters

mean
sycl::buffer array of mean values.

raw_moment_2
Optional parameter. sycl::buffer array of 2nd order raw moment values.

raw_moment_3
Optional parameter. sycl::buffer array of 3rd order raw moment values.

raw_moment_4
Optional parameter. sycl::buffer array of 4th order raw moment values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when mean.get_count() == 0 & raw_moment_2.get_count() == 0 & raw_moment_3.get_count() == 0 & raw_moment_4.get_count() == 0, or dataset object is invalid

raw_moment (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    sycl::event raw_moment(
    sycl::queue& queue,
    const dataset<ObservationsLayout, Type*>& data,
    Type* mean,
    Type* raw_moment_2 = nullptr,
    Type* raw_moment_3 = nullptr,
    Type* raw_moment_4 = nullptr,
    const std::vector<sycl::event> &dependencies = {});
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

data
Dataset which is used for computation.

dependencies
Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

mean
Pointer to the array of mean values.

raw_moment_2
Optional parameter. Pointer to the array of the 2nd order raw moment values.

raw_moment_3
Optional parameter. Pointer to the array of the 3rd order raw moment values.

raw_moment_4
Optional parameter. Pointer to the array of the 2nd order raw moment values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when mean == nullptr & raw_moment_2 == nullptr & raw_moment_3 == nullptr & raw_moment_4 == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

central_moment

Entry point to compute central moments up to the 4th order.

Description and Assumptions

The oneapi::mkl::stats::central_moment function is used to compute an array of central moments up to the 4th order (central moments for each dataset's dimension).

central_moment supports the following precisions for data:

T
float
double

central_moment (buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = oneapi::mkl::stats::method::fast, typename Type,
    layout ObservationsLayout>
    void central_moment(sycl::queue& queue,
    const dataset<ObservationsLayout, sycl::buffer<Type, 1>>& data,
```

(continues on next page)

(continued from previous page)

```
    sycl::buffer<Type, 1> central_moment_2,
    sycl::buffer<Type, 1> central_moment_3 = {0},
    sycl::buffer<Type, 1> central_moment_4 = {0});
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

data
Dataset which is used for computation.

Output Parameters

central_moment_2
sycl::buffer array of 2nd order central moment values.

central_moment_3
Optional parameter. sycl::buffer array of 3rd order central moment values.

central_moment_4
Optional parameter. sycl::buffer array of 4th order central moment values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when central_moment_2.get_count() == 0 & central_moment_3.get_count() == 0 & central_moment_4.get_count() == 0, or dataset object is invalid

central_moment (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    sycl::event central_moment(sycl::queue& queue,
    const dataset<ObservationsLayout, Type*>& data, Type* central_moment_2,
    Type* central_moment_3 = nullptr, Type* central_moment_4 = nullptr,
    const std::vector<sycl::event> &dependencies = {});
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

data
Dataset which is used for computation.

dependencies
Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

central_moment_2
Pointer to the array of the 2nd order central moment values.

central_moment_3
Optional parameter. Pointer to the array of the 3rd order central moment values.

central_moment_4
Optional parameter. Pointer to the array of the 2nd order central moment values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when central_moment_2 == nullptr & central_moment_3 == nullptr & central_moment_4 == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

central_moment with provided mean

Entry point to compute central moments up to the 4th order with the provided mean values.

Description and Assumptions

The oneapi::mkl::stats::central_moment function is used to compute an array of central moments up to the 4th order (central moments for each dataset's dimension) with the provided mean values.

central_moment with provided mean supports the following precisions for data:

T
float
double

central_moment with provided mean (buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    void central_moment(sycl::queue& queue,
    sycl::buffer<Type, 1> mean,
    const dataset<ObservationsLayout, sycl::buffer<Type, 1>>& data,
    sycl::buffer<Type, 1> central_moment_2,
    sycl::buffer<Type, 1> central_moment_3 = {0},
    sycl::buffer<Type, 1> central_moment_4 = {0});
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

mean
sycl::buffer to the array of provided mean values.

data
Dataset which is used for computation.

Output Parameters

central_moment_2
sycl::buffer array of 2nd order central moment values.

central_moment_3
Optional parameter. sycl::buffer array of 3rd order central moment values.

central_moment_4
Optional parameter. sycl::buffer array of 4th order central moment values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when central_moment_2.get_count() == 0 & central_moment_3.get_count() == 0 & central_moment_4.get_count() == 0, or mean.get_count() == 0, or dataset object is invalid

central_moment with provided mean (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    sycl::event central_moment(sycl::queue& queue,
    Type* mean,
    const dataset<ObservationsLayout, Type*>& data,
    Type* central_moment_2,
    Type* central_moment_3 = nullptr,
    Type* central_moment_4 = nullptr,
```

(continues on next page)

(continued from previous page)

```
    const std::vector<sycl::event> &dependencies = {});
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

mean
Pointer to the array of provided mean values.

data
Dataset which is used for computation.

dependencies
Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

central_moment_2
Pointer to the array of the 2nd order central moment values.

central_moment_3
Optional parameter. Pointer to the array of the 3rd order central moment values.

central_moment_4
Optional parameter. Pointer to the array of the 2nd order central moment values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when central_moment_2 == nullptr & central_moment_3 == nullptr & central_moment_4 == nullptr or mean == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

variation

Entry point to compute variation.

Description and Assumptions

The oneapi::mkl::stats::variation function is used to compute a variation array (variation for each dataset's dimension).

variation supports the following precisions for data:

T
float
double

variation (buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    void variation(sycl::queue& queue,
    const dataset<ObservationsLayout, sycl::buffer<Type, 1>>& data,
    sycl::buffer<Type, 1> variation);
}
```

Template Parameters

Method
: Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type
: Data precision.

ObservationsLayout
: Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

data
Dataset which is used for computation.

Output Parameters

variation
sycl::buffer array of variation values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when variation.get_count() == 0, or dataset object is invalid

variation (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    sycl::event variation(sycl::queue& queue,
    const dataset<ObservationsLayout, Type*>& data,
    Type* variation,
    const std::vector<sycl::event> &dependencies = {});
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

data
Dataset which is used for computation.

dependencies
Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

variation
Pointer to the array of variation values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when variation == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

variation with provided mean

Entry point to compute variation with the provided mean values.

Description and Assumptions

The oneapi::mkl::stats::variation function is used to compute an array of variation (variation for each dataset's dimension) with the provided mean values.

variation with provided mean supports the following precisions for data:

T
float
double

oneapi::mkl::stats::variation (buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    void variation(sycl::queue& queue, sycl::buffer<Type, 1> mean,
    const dataset<ObservationsLayout, sycl::buffer<Type, 1>>& data,
    sycl::buffer<Type, 1> variation);
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

mean
sycl::buffer to the array of provided mean values.

data
Dataset which is used for computation.

Output Parameters

variation
sycl::buffer array of variation values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when variation.get_count() == 0, or mean.get_count() == 0, or dataset object is invalid

variation with provided mean (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    sycl::event variation(sycl::queue& queue,
    Type* mean,
    const dataset<ObservationsLayout, Type*>& data,
    Type* variation,
    const std::vector<sycl::event> &dependencies = {});
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

mean
Pointer to the array of provided mean values.

data
Dataset which is used for computation.

dependencies
Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

variation
Pointer to the array of the variation values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when variation == nullptr, or mean == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

skewness

Entry point to compute skewness.

Description and Assumptions

The oneapi::mkl::stats::skewness function is used to compute a skewness array (skewness for each dataset's dimension).

skewness supports the following precisions for data:

T
float
double

skewness (buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    void skewness(sycl::queue& queue,
    const dataset<ObservationsLayout, sycl::buffer<Type, 1>>& data,
    sycl::buffer<Type, 1> skewness);
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

data
Dataset which is used for computation.

Output Parameters

skewness
sycl::buffer array of skewness values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when skewness.get_count() == 0, or dataset object is invalid

skewness (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    sycl::event skewness(sycl::queue& queue,
    const dataset<ObservationsLayout, Type*>& data,
    Type* skewness,
    const std::vector<sycl::event> &dependencies = {});
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

data
Dataset which is used for computation.

dependencies
Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

skewness
Pointer to the array of skewness values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when skewness == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

skewness with provided mean

Entry point to compute skewness with the provided mean values.

Description and Assumptions

The oneapi::mkl::stats::skewness function is used to compute an array of skewness (skewness for each dataset's dimension) with the provided mean values.

skewness with provided mean supports the following precisions for data:

T
float
double

skewness with provided mean (buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    void skewness(sycl::queue& queue,
    sycl::buffer<Type, 1> mean,
    const dataset<ObservationsLayout, sycl::buffer<Type, 1>>& data,
    sycl::buffer<Type, 1> skewness);
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

mean
sycl::buffer to the array of provided mean values.

data
Dataset which is used for computation.

Output Parameters

skewness
sycl::buffer array of skewness values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when skewness.get_count() == 0, or mean.get_count() == 0, or dataset object is invalid

skewness with provided mean (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    sycl::event skewness(sycl::queue& queue,
    Type* mean,
    const dataset<ObservationsLayout, Type*>& data,
    Type* skewness,
    const std::vector<sycl::event> &dependencies = {});
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

mean
Pointer to the array of provided mean values.

data
Dataset which is used for computation.

dependencies
Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

skewness
Pointer to the array of the skewness values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when skewness == nullptr, or mean == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

kurtosis

Entry point to compute kurtosis.

Description and Assumptions

The oneapi::mkl::stats::kurtosis function is used to compute a kurtosis array (kurtosis for each dataset's dimension).

kurtosis supports the following precisions for data:

T
float
double

kurtosis (buffer version)

Syntax

```
namespace oneapi::mkl::stats {
    template<method Method = method::fast, typename Type, layout ObservationsLayout>
    void kurtosis(sycl::queue& queue,
    const dataset<ObservationsLayout, sycl::buffer<Type, 1>>& data,
    sycl::buffer<Type, 1> kurtosis);
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

data
Dataset which is used for computation.

Output Parameters

kurtosis
sycl::buffer array of kurtosis values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when kurtosis.get_count() == 0, or dataset object is invalid

kurtosis (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    sycl::event kurtosis(sycl::queue& queue,
    const dataset<ObservationsLayout, Type*>& data,
    Type* kurtosis,
    const std::vector<sycl::event> &dependencies = {});
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

data
Dataset which is used for computation.

dependencies
Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

kurtosis
Pointer to the array of kurtosis values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when kurtosis == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

kurtosis with provided mean

Entry point to compute kurtosis with the provided mean values.

Description and Assumptions

The oneapi::mkl::stats::kurtosis function is used to compute an array of kurtosis (kurtosis for each dataset's dimension) with the provided mean values.

kurtosis with provided mean supports the following precisions for data:

T
float
double

kurtosis with provided mean (buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type,
    layout ObservationsLayout>
    void oneapi::mkl::stats::kurtosis(sycl::queue& queue,
    sycl::buffer<Type, 1> mean,
    const oneapi::mkl::stats::dataset<sycl::buffer<Type, 1>, ObservationsLayout>& data,
    sycl::buffer<Type, 1> kurtosis);
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

mean
sycl::buffer to the array of provided mean values.

data
Dataset which is used for computation.

Output Parameters

kurtosis
sycl::buffer array of kurtosis values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when kurtosis.get_count() == 0, or mean.get_count() == 0, or dataset object is invalid

kurtosis with provided mean (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = fast, typename Type, layout ObservationsLayout>
    sycl::event kurtosis(sycl::queue& queue,
    Type* mean,
    const dataset<ObservationsLayout, Type*>& data,
    Type* kurtosis,
    const std::vector<sycl::event> &dependencies = {});
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

mean
Pointer to the array of provided mean values.

data
Dataset which is used for computation.

dependencies
Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

kurtosis
Pointer to the array of the kurtosis values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when kurtosis == nullptr, or mean == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

min

Entry point to compute min values.

Description and Assumptions

The oneapi::mkl::stats::min function is used to compute min arrays (min value for each dataset's dimension).

min supports the following precisions for data:

T
float
double

min (buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = fast, typename Type, layout ObservationsLayout>
    void min(sycl::queue& queue,
    const dataset<ObservationsLayout, sycl::buffer<Type, 1>>& data,
    sycl::buffer<Type, 1> min);
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

data
Dataset which is used for computation.

Output Parameters

min
sycl::buffer array of min values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when min.get_count() == 0, or dataset object is invalid

min (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = fast, typename Type, layout ObservationsLayout>
    sycl::event min(sycl::queue& queue,
    const dataset<ObservationsLayout, Type*>& data,
    Type* min,
    const std::vector<sycl::event> &dependencies = {});
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

data
Dataset which is used for computation.

dependencies
Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

min
Pointer to the array of min values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when min == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

max

Entry point to compute max values.

Description and Assumptions

The oneapi::mkl::stats::max function is used to compute a max values arrays (max value for each dataset's dimension).

max supports the following precisions for data:

T
float
double

max (buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    void max(sycl::queue& queue,
    const dataset<ObservationsLayout, sycl::buffer<Type, 1>>& data,
    sycl::buffer<Type, 1> max);
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

data
Dataset which is used for computation.

Output Parameters

max
sycl::buffer array of max values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when max.get_count() == 0, or dataset object is invalid

max (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    sycl::event max(sycl::queue& queue,
    const dataset<ObservationsLayout, Type*>& data,
    Type* max,
    const std::vector<sycl::event> &dependencies = {});
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

data
Dataset which is used for computation.

dependencies
Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

max
Pointer to the array of max values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when max == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

min_max

Entry point to compute min and max values.

Description and Assumptions

The oneapi::mkl::stats::min_max function is used to compute min and max arrays (min and max values for each dataset's dimension).

min_max supports the following precisions for data:

T
float
double

min_max (buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    void min_max(sycl::queue& queue,
    const dataset<ObservationsLayout, sycl::buffer<Type, 1>>& data,
    sycl::buffer<Type, 1> min,
    sycl::buffer<Type, 1> max);
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

data
Dataset which is used for computation.

Output Parameters

min
sycl::buffer array of min values.

max
sycl::buffer array of max values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when min.get_count() == 0, or max.get_count() == 0, or dataset object is invalid

min_max (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    sycl::event min_max(sycl::queue& queue,
    const dataset<ObservationsLayout, Type*>& data,
    Type* min,
    Type* max,
    const std::vector<sycl::event> &dependencies = {});
}
```

Template Parameters

Method
Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type
Data precision.

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Input Parameters

queue
The queue where the routine should be executed.

data
Dataset which is used for computation.

dependencies
Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

min
Pointer to the array of min values.

max
Pointer to the array of max values.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when min == nullptr, or max == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

Service Routines

Routine	Description
make_dataset	Creates a dataset from the provided parameters

Parent topic: *Summary Statistics*

make_dataset

Entry point to create a dataset from the provided parameters.

Description and Assumptions

The oneapi::mkl::stats::make_dataset function is used to create a dataset from the provided storage of the observations matrix, the number of dimensions and observations, and other parameters.

make_dataset supports the following precisions for data:

T
float
double

make_dataset (buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<layout ObservationsLayout = layout::row_major, typename Type>
    dataset<sycl::buffer<Type, 1>, ObservationsLayout> make_dataset(
    std::int64_t n_dims,
    std::int64_t n_observations,
    sycl::buffer<Type, 1> observations,
    sycl::buffer<Type, 1> weights = {0},
    sycl::buffer<std::int64_t, 1> indices = {0});
}
```

Template Parameters

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Type
Data precision.

Input Parameters

n_dims
The number of dimensions.

n_observations
The number of observations.

observations
Matrix of observations.

weights
Optional parameter. Array of weights of size n_observations. Elements of the array are non-negative members. If the parameter is not specified, each observation has weight equal to 1.

indices
Optional parameter. Array of vector components that are processed. The size of the array is n_dims. If the parameter is not specified, all components are processed.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when :math: *n_dims leq 0*, or :math: *n_observations leq 0*, or observations.get_count() == 0

Return Value

Dataset holding specified parameters.

make_dataset (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<layout ObservationsLayout = layout::row_major, typename Type>
    dataset<Type*, ObservationsLayout> make_dataset(std::nt64_t
    n_dims, std::int64_t n_observations,
    Type* observations, Type* weights = nullptr, std::int64_t* indices = nullptr);
}
```

Template Parameters

ObservationsLayout
Data layout. The specific values are described in *dataset*.

Type
Data precision.

Input Parameters

n_dims
The number of dimensions.

n_observations
The number of observations.

observations
Matrix of observations.

weights
Optional parameter. Array of weights of size n_observations. Elements of the array are non-negative members. If the parameter is not specified, each observation has weight equal to 1.

indices
Optional parameter. Array of vector components that are processed. Size of array is n_dims. If the parameter is not specified, all components are processed.

Throws

oneapi::mkl::invalid_argument
Exception is thrown when $n\_dims \leq 0$, or $n\_observations \leq 0$, or observations == nullptr

Return Value

Dataset holding specified parameters.

Parent topic: *Service Routines*

Parent topic: *Summary Statistics*

9.2.6 Vector Math

oneMKL *Vector Mathematics functions* (VM) compute a mathematical function of each of the vector elements. VM includes a set of functions (arithmetic, power, trigonometric, exponential, hyperbolic, special, and rounding) that operate on vectors of real and complex numbers.

Vector Math

Application programs that improve performance with VM include nonlinear programming software, computation of integrals, financial calculations, computer graphics, and many others.

VM functions fall into the following groups according to the operations they perform:

- *VM Mathematical Functions* compute values of mathematical functions, such as sine, cosine, exponential, or logarithm, on vectors stored contiguously in memory.
- *VM Service Functions* set/get the accuracy modes and the error codes, and create error handlers for mathematical functions.

The VM mathematical functions take an input vector as an argument, compute values of the respective function elementwise, and return the results in an output vector. All the VM mathematical functions can perform in-place operations, where the input and output arrays are at the same memory locations.

- *Special Value Notations*

Special Value Notations

This defines notations of special values for complex functions. The definitions are provided in text, tables, or formulas.

- z, z1, z2, etc. denote complex numbers.
- i, `i2=-1` is the imaginary unit.
- x, X, x1, x2, etc. denote real imaginary parts.
- y, Y, y1, y2, etc. denote imaginary parts.
- **X and Y represent any finite positive IEEE-754 floating point**
  values, if not stated otherwise.
- **Quiet NaN and signaling NaN are denoted with QNAN and SNAN,**
  respectively.
- **The IEEE-754 positive infinities or floating-point numbers are**
  denoted with a + sign before X, Y, etc.

- **The IEEE-754 negative infinities or floating-point numbers are**
  denoted with a - sign before X, Y, etc.

`CONJ(z)` and `CIS(z)` are defined as follows:

`CONJ(x+i·y)=x-i·y`

`CIS(y)=cos(y)+i·sin(y)`.

The special value tables show the result of the function for the z argument at the intersection of the RE(z) column and the i*IM(z) row. If the function encounters a special point for the argument z, the lower part of this cell shows the special point and the VM status value for this element. An empty cell indicates that this argument is normal and the result is well-defined computationally.

Parent topic: *Vector Math*

VM Mathematical Functions

This section describes VM functions that compute values of mathematical functions on real and complex vector arguments with unit increment.

Each function is introduced by its short name, a brief description of its purpose, and the calling sequence for each type of data, as well as a description of the input/output arguments.

The input range of parameters is equal to the mathematical range of the input data type, unless the function description specifies input threshold values, which mark off the precision overflow, as follows:

- `FLT_MAX` denotes the maximum number representable in single precision real data type
- `DBL_MAX` denotes the maximum number representable in double precision real data type

The following tables list the available mathematical functions grouped by category.

Arithmetic Routines	Description
add	Adds vector elements
sub	Subtracts vector elements
sqr	Squares vector elements
mul	Multiplies vector elements
mulbyconj	Multiplies elements of one vector by conjugated elements of the second vector
conj	Conjugates vector elements
abs	Computes the absolute value of vector elements
arg	Computes the argument of vector elements
linearfrac	Performs linear fraction transformation of vectors
fmod	Performs element by element computation of the modulus function of vector `a` with respect to vector `b`
remainder	Performs element by element computation of the remainder function on the elements of vector `a` and the corresponding elements of vector `b`

Power and Root Routines	Description
inv	Inverts vector elements
div	Divides elements of one vector by elements of the second vector
sqrt	Computes the square root of vector elements
invsqrt	Computes the inverse square root of vector elements

continues on next page

Table 5 – continued from previous page

Power and Root Routines	Description
cbrt	Computes the cube root of vector elements
invcbrt	Computes the inverse cube root of vector elements
pow2o3	Computes the cube root of the square of each vector element
pow3o2	Computes the square root of the cube of each vector element
pow	Raises each vector element to the specified power
powx	Raises each vector element to the constant power
powr	Computes `a` to the power `b` for elements of two vectors, where the elements of vector argument `a` are all non-negative
hypot	Computes the square root of sum of squares

Exponential and Logarithmic Routines	Description
exp	Computes the base `e` exponential of vector elements
exp2	Computes the base 2 exponential of vector elements
exp10	Computes the base 10 exponential of vector elements
expm1	Computes the base `e` exponential of vector elements decreased by 1
ln	Computes the natural logarithm of vector elements
log2	Computes the base 2 logarithm of vector elements
log10	Computes the base 10 logarithm of vector elements
log1p	Computes the natural logarithm of vector elements that are increased by 1
logb	Computes the exponents of the elements of input vector `a`

Trigonometric Routines	Description
cos	Computes the cosine of vector elements
sin	Computes the sine of vector elements
sincos	Computes the sine and cosine of vector elements
cis	Computes the complex exponent of vector elements (cosine and sine combined to complex value)
tan	Computes the tangent of vector elements
acos	Computes the inverse cosine of vector elements
asin	Computes the inverse sine of vector elements
atan	Computes the inverse tangent of vector elements
atan2	Computes the four-quadrant inverse tangent of ratios of the elements of two vectors
cospi	Computes the cosine of vector elements multiplied by π
sinpi	Computes the sine of vector elements multiplied by π
tanpi	Computes the tangent of vector elements multiplied by π
acospi	Computes the inverse cosine of vector elements divided by π
asinpi	Computes the inverse sine of vector elements divided by π
atanpi	Computes the inverse tangent of vector elements divided by π
atan2pi	Computes the four-quadrant inverse tangent of the ratios of the corresponding elements of two vectors divided by π
cosd	Computes the cosine of vector elements multiplied by $\pi/180$
sind	Computes the sine of vector elements multiplied by $\pi/180$
tand	Computes the tangent of vector elements multiplied by $\pi/180$

Hyperbolic Routines	Description
cosh	Computes the hyperbolic cosine of vector elements
sinh	Computes the hyperbolic sine of vector elements
tanh	Computes the hyperbolic tangent of vector elements
acosh	Computes the inverse hyperbolic cosine of vector elements
asinh	Computes the inverse hyperbolic sine of vector elements
atanh	Computes the inverse hyperbolic tangent of vector elements.

Special Routines	Description
erf	Computes the error function value of vector elements
erfc	Computes the complementary error function value of vector elements
cdfnorm	Computes the cumulative normal distribution function value of vector elements
erfinv	Computes the inverse error function value of vector elements
erfcinv	Computes the inverse complementary error function value of vector elements
cdfnorminv	Computes the inverse cumulative normal distribution function value of vector elements
lgamma	Computes the natural logarithm for the absolute value of the gamma function of vector elements
tgamma	Computes the gamma function of vector elements
expint1	Computes the exponential integral of vector elements

Rounding Routines	Description
floor	Rounds towards minus infinity
ceil	Rounds towards plus infinity
trunc	Rounds towards zero infinity
round	Rounds to nearest integer
nearbyint	Rounds according to current mode
rint	Rounds according to current mode and reports inexact result status
modf	Computes the integer and fractional parts
frac	Computes the fractional part

Miscellaneous Routines	Description
copysign	Returns vector of elements of one argument with signs changed to match other argument elements
nextafter	Returns vector of elements containing the next representable floating-point values following the values from the elements of one vector in the direction of the corresponding elements of another vector
fdim	Returns vector containing the differences of the corresponding elements of the vector arguments if the first is larger and +0 otherwise
fmax	Returns the larger of each pair of elements of the two vector arguments
fmin	Returns the smaller of each pair of elements of the two vector arguments
maxmag	Returns the element with the larger magnitude between each pair of elements of the two vector arguments

continues on next page

Table 11 – continued from previous page

Miscellaneous Routines	Description
minmag	Returns the element with the smaller magnitude between each pair of elements of the two vector arguments

Parent topic: *Vector Math*

abs

Computes absolute value of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event abs(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<R,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event abs(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        R* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`abs` supports the following precisions.

T	R
`float`	`float`
`double`	`double`
`std::complex<float>`	`float`
`std::complex<double>`	`double`

Description

The abs(a) function computes an absolute value of vector elements.

Argument	Result	Status code
+0	+0	
-0	+0	
+∞	+∞	
-∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

`abs(a) = hypot(RE(a), IM(a))`.

The abs function does not generate any errors.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

acos

Computes inverse cosine of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event acos(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event acos(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});
```

(continues on next page)

(continued from previous page)

```
} // namespace oneapi::mkl::vm
```

`acos` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Description

The acos(a) function computes inverse cosine of vector elements.

Argument	Result	Status code
+0	+π/2	
-0	+π/2	
+1	+0	
-1	+π	
|a| > 1	QNAN	`oneapi::mkl::vm::status::errdom`
+∞	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

RE(a) i·IM(a)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	+3·π/4-i·∞	+π/2-i·∞	+π/2-i·∞	+π/2-i·∞	+π/2-i·∞	+π/4-i·∞	QNAN-i·∞
+i·Y	+π-i·∞					+0-i·∞	QNAN+i·QNAN
+i·0	+π-i·∞		+π/2-i·0	+π/2-i·0		+0-i·∞	QNAN+i·QNAN
-i·0	+π+i·∞		+π/2+i·0	+π/2+i·0		+0+i·∞	QNAN+i·QNAN
-i·Y	+π+i·∞					+0+i·∞	QNAN+i·QNAN
-i·∞	+3π/4+i·∞	+π/2+i·∞	+π/2+i·∞	+π/2+i·∞	+π/2+i·∞	+π/4+i·∞	QNAN+i·∞
+i·NAN	QNAN+i·○	QNAN+i·QNA	+π/2+i·QNA	+π/2+i·QNA	QNAN+i·QNA	QNAN+i·○	QNAN+i·QNAN

Notes:

- `acos(CONJ(a))=CONJ(acos(a))`.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

acosh

Computes inverse hyperbolic cosine (nonnegative) of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event acosh(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event acosh(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`acosh` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

Description

The acosh(a) function computes inverse hyperbolic cosine (nonnegative) of vector elements.

Argument	Result	Status code
+1	+0	
a < +1	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
+∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

RE(a) i·IM(a)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	$+\infty + i \cdot \frac{3\pi}{4}$	+∞+i·π/2	+∞+i·π/2	+∞+i·π/2	+∞+i·π/2	+∞+i·π/4	+∞+i·QNAN
+i·Y	+∞+i·π					+∞+i·0	QNAN+i·QNAN
+i·0	+∞+i·π		+0+i·π/2	+0+i·π/2		+∞+i·0	QNAN+i·QNAN
-i·0	+∞+i·π		+0+i·π/2	+0+i·π/2		+∞+i·0	QNAN+i·QNAN
-i·Y	+∞+i·π					+∞+i·0	QNAN+i·QNAN
-i·∞	$+\infty - i \cdot \frac{3\pi}{4}$	+∞-i·π/2	+∞-i·π/2	+∞-i·π/2	+∞-i·π/2	+∞-i·π/4	+∞+i·QNAN
+i·NAN	+∞+i·QNA	QNAN+i·QN/	QNAN+i·QN/	QNAN+i·QN/	QNAN+i·QN/	+∞+i·QNA	QNAN+i·QNAN

Notes:

- `acosh(CONJ(a))=CONJ(acosh(a))`.

Input Parameters

Buffer API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
: The buffer `a` containing input vector of size `n`.

mode
: Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
: Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
: The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

acospi

Computes the inverse cosine of vector elements divided by π.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event acospi(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
```

(continues on next page)

(continued from previous page)

```
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event acospi(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`acospi` supports the following precisions.

T
`float`
`double`

Description

The acospi(a) function computes the inverse cosine of vector elements divided by π. For an argument a, the function computes acos(a)/π.

Argument	Result	Status code
+0	+1/2	
-0	+1/2	
+1	+0	
-1	+1	
|a| > 1	QNAN	`oneapi::mkl::vm::status::errdom`
$+\infty$	QNAN	`oneapi::mkl::vm::status::errdom` | • ∞	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

add

Performs element by element addition of vector a and vector b.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event add(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event add(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        const T *b,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

add supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

Description

The add(a, b) function performs element by element addition of vector `a` and vector `b`.

Argument 1	Argument 2	Result	Status code
+0	+0	+0	
+0	-0	+0	
-0	+0	+0	
-0	-0	-0	
+∞	+∞	+∞	
+∞	-∞	QNAN	
-∞	+∞	QNAN	
-∞	-∞	-∞	
SNAN	any value	QNAN	
any value	SNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

```
add(x1+i*y1, x2+i*y2) = (x1+x2) + i*(y1+y2)
```

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when all RE(x), RE(y), IM(x), IM(y) arguments are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns ∞ in that part of the result, and sets the VM status code to `oneapi::mkl::vm::status::overflow` (overriding any possible `oneapi::mkl::vm::status::accuracy_warning` status).

Input Parameters

Buffer API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
: The buffer `a` containing 1st input vector of size `n`.

b
: The buffer `b` containing 2nd input vector of size `n`.

mode
: Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
: Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
Pointer `a` to the 1st input vector of size `n`.

b
Pointer `b` to the 2nd input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

arg

Computes argument of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event arg(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<R,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);
```

(continues on next page)

(continued from previous page)

```
} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event arg(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        R* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`arg` supports the following precisions.

T	R
`std::complex<float>`	`float`
`std::complex<double>`	`double`

Description

The arg(a) function computes argument of vector elements.

See *Special Value Notations* for the conventions used in the table below.

RE(a) i·IM(a)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	+3·π/4	+π/2	+π/2	+π/2	+π/2	+π/4	NAN
+i·Y	+π		+π/2	+π/2		+0	NAN
+i·0	+π	+π	+π	+0	+0	+0	NAN
-i·0	-π	-π	-π	-0	-0	-0	NAN
-i·Y	-π		-π/2	-π/2		-0	NAN
-i·∞	-3·π/4	-π/2	-π/2	-π/2	-π/2	-π/4	NAN
+i·NAN	NAN	NAN	NAN	NAN	NAN	NAN	NAN

Note

`arg(a)=atan2(IM(a), RE(a))`

The arg function does not generate any errors.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

asin

Computes inverse sine of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event asin(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event asin(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`asin` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

Description

The asin(a) function computes inverse sine of vector elements.

Argument	Result	Status code
+0	+0	
-0	-0	
+1	+π/2	
-1	-π/2	
|a| > 1	QNAN	`oneapi::mkl::vm::status::errdom`
+∞	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

`asin(a) = -i*asinh(i*z)`.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

asinh

Computes inverse hyperbolic sine of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event asinh(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event asinh(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
```

(continues on next page)

(continued from previous page)

```
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`asinh` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Description

The asinh(a) function computes inverse hyperbolic sine of vector elements.

Argument	Result	Status code
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
QNAN	QNAN	
SNAN	QNAN	

RE(a) i·IM(a)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	-∞+i·π/4	-∞+i·π/2	+∞+i·π/2	+∞+i·π/2	+∞+i·π/2	+∞+i·π/4	+∞+i·QNAN
+i·Y	-∞+i·0					+∞+i·0	QNAN+i·QNAN
+i·0	+∞+i·0		+0+i·0	+0+i·0		+∞+i·0	QNAN+i·QNAN
-i·0	-∞-i·0		-0-i·0	+0-i·0		+∞-i·0	QNAN-i·QNAN
-i·Y	-∞-i·0					+∞-i·0	QNAN+i·QNAN
-i·∞	-∞-i·π/4	-∞-i·π/2	-∞-i·π/2	+∞-i·π/2	+∞-i·π/2	+∞-i·π/4	+∞+i·QNAN
+i·NAN	-∞+i·QNAI	QNAN+i·QN/	QNAN+i·QN/	QNAN+i·QN/	QNAN+i·QN/	+∞+i·QNA	QNAN+i·QNAN

The asinh(a) function does not generate any errors.

Notes:

- `asinh(CONJ(a))=CONJ(asinh(a))`
- `asinh(-a)=-asinh(a)`.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

asinpi

Computes the inverse sine of vector elements divided by π.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event asinpi(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event asinpi(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`asinpi` supports the following precisions.

T
float
double

Description

The asinpi(a) function computes the inverse sine of vector elements divided by π. For an argument `a`, the function computes asinpi(`a`)/π.

Argument	Result	Status code
+0	+0	
-0	-0	
+1	+1/2	
-1	-1/2	
|a| > 1	QNAN	`oneapi::mkl::vm::status::errdom`
$+\infty$	QNAN	`oneapi::mkl::vm::status::errdom`
$-\infty$	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

atan

Computes inverse tangent of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event atan(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event atan(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
```

(continues on next page)

(continued from previous page)

```
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`atan` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Description

The atan(a) function computes inverse tangent of vector elements.

Argument	Result	Status code
+0	+0	
-0	-0	
$+\infty$	$+\pi/2$	
$-\infty$	$-\pi/2$	
QNAN	QNAN	
SNAN	QNAN	

The atan function does not generate any errors.

Specifications for special values of the complex functions are defined according to the following formula

`atan(a) = -i*atanh(i*a)`.

Input Parameters

Buffer API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
: The buffer `a` containing input vector of size `n`.

mode
: Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue
: The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

atan2

Computes four-quadrant inverse tangent of elements of two vectors.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event atan2(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event atan2(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        const T *b,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

ad2d supports the following precisions.

T
float
double

Description

The atan2(a, b) function computes four-quadrant inverse tangent of elements of two vectors.

The elements of the output vector are computed as the four-quadrant arctangent of `a[i] / b[i]`.

Argument 1	Argument 2	Result	Status code
-∞	-∞	-3*π/4	
-∞	b < +0	-π/2	
-∞	-0	-π/2	
-∞	+0	-π/2	
-∞	b > +0	-π/2	
-∞	+∞	-π/4	
a < +0	-∞	-π	
a < +0	-0	-π/2	
a < +0	+0	-π/2	
a < +0	+∞	-0	
-0	-∞	-π	
-0	b < +0	-π	
-0	-0	-π	
-0	+0	-0	
-0	b > +0	-0	
-0	+∞	-0	
+0	-∞	+π	
+0	b < +0	+π	
+0	-0	+π	
+0	+0	+0	
+0	b > +0	+0	
+0	+∞	+0	
a > +0	-∞	+π	

continues on next page

Table 12 – continued from previous page

Argument 1	Argument 2	Result	Status code
a > +0	-0	+π/2	
a > +0	+0	+π/2	
a > +0	+∞	+0	
+∞	-∞	+3*π/4	
+∞	b < +0	+π/2	
+∞	-0	+π/2	
+∞	+0	+π/2	
+∞	b > +0	+π/2	
+∞	+∞	+π/4	
a > +0	QNAN	QNAN	
a > +0	SNAN	QNAN	
QNAN	b > +0	QNAN	
SNAN	b > +0	QNAN	
QNAN	QNAN	QNAN	
QNAN	SNAN	QNAN	
SNAN	QNAN	QNAN	
SNAN	SNAN	QNAN	

The atan2(a, b) function does not generate any errors.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing 1st input vector of size `n`.

b
The buffer `b` containing 2nd input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the 1st input vector of size `n`.

b
Pointer `b` to the 2nd input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

atan2pi

Computes the four-quadrant inverse tangent of the ratios of the corresponding elements of two vectors divided by π.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event atan2pi(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event atan2pi(
        sycl::queue& exec_queue,
        std::int64_t n,
```

(continues on next page)

(continued from previous page)

```
        const T *a,
        const T *b,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`atan2pi` supports the following precisions.

T
`float`
`double`

Description

The atan2pi(a, b) function computes the four-quadrant inverse tangent of the ratios of the corresponding elements of two vectors divided by π.

For the elements of the output vector `y`, the function computers the four-quadrant arctangent of a_i/b_i, with the result divided by π.

Argument 1	Argument 2	Result	Status code
-∞	-∞	-3/4	
-∞	b < +0	-1/2	
-∞	-0	+1/2	
-∞	+0	-1/2	
-∞	x > +0	-1/2	
-∞	+∞	-1/4	
a < +0	-∞	-1	
a < +0	-0	-1/2	
a < +0	+0	-1/2	
a < +0	+∞	-0	
-0	-∞	-1	
-0	b < +0	-1	
-0	-0	-1	
-0	+0	-0	
-0	b > +0	-0	
-0	+∞	-0	
+0	-∞	+1	
+0	b < +0	+1	
+0	-0	+1	
+0	+0	+0	
+0	b > +0	+0	
+0	+∞	+0	
a > +0	-∞	+1	
a > +0	-0	+1/2	
x > +0	+0	+1/2	
a > +0	+∞	+1/4	

continues on next page

Table 13 – continued from previous page

Argument 1	Argument 2	Result	Status code
+∞	-∞	+3/4	
+∞	b < +0	+1/2	
+∞	-0	+1/2	
+∞	+0	+1/2	
+∞	b > +0	+1/2	
+∞	+∞	+1/4	
a > +0	QNAN	QNAN	
a > +0	SNAN	QNAN	
QNAN	b > +0	QNAN	
SNAN	x > +0	QNAN	
QNAN	QNAN	QNAN	
QNAN	SNAN	QNAN	
SNAN	QNAN	QNAN	
SNAN	SNAN	QNAN	

The atan2pi(a, b) function does not generate any errors.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing 1st input vector of size `n`.

b
The buffer `b` containing 2nd input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the 1st input vector of size `n`.

b
Pointer `b` to the 2nd input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for

possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

atanh

Computes inverse hyperbolic tangent of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event atanh(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event atanh(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
```

(continues on next page)

(continued from previous page)

```
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`atanh` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Description

The atanh(a) function computes inverse hyperbolic tangent of vector elements.

Argument	Result	Status code
+1	+∞	`oneapi::mkl::vm::status::sing`
-1	-∞	`oneapi::mkl::vm::status::sing`
|a| > 1	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
+∞	+∞	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

RE(a) i·IM(a)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	-0+i·π/2	-0+i·π/2	-0+i·π/2	+0+i·π/2	+0+i·π/2	+0+i·π/2	+0+i·π/2
+i·Y	-0+i·π/2					+0+i·π/2	QNAN+i·QNAN
+i·0	-0+i·π/2		-0+i·0	+0+i·0		+0+i·π/2	QNAN+i·QNAN
-i·0	-0-i·π/2		-0-i·0	+0-i·0		+0-i·π/2	QNAN-i·QNAN
-i·Y	-0-i·π/2					+0-i·π/2	QNAN+i·QNAN
-i·∞	-0-i·π/2	-0-i·π/2	-0-i·π/2	+0-i·π/2	+0-i·π/2	+0-i·π/2	+0-i·π/2
+i·NAN	-0+i·QNAN	QNAN+i·QNA	-0+i·QNAN	+0+i·QNAN	QNAN+i·QNA	+0+i·QNAN	QNAN+i·QNAN

Notes:

- `atanh(±1±i*0)`=±∞±`i*0`, and oneapi::mkl::vm::status::sing error is generated
- `atanh(CONJ(a))=CONJ(atanh(a))`
- `atanh(-a)=-atanh(a)`.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

atanpi

Computes the inverse tangent of vector elements divided by π.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event atanpi(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event atanpi(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`atanpi` supports the following precisions.

T
float
double

Description

The atanpi(a) function computes the inverse tangent of vector elements divided by π. For an argument `a`, the function computes atan(`a`)/π.

Argument	Result	Status code
+0	+0	
-0	-0	
$+\infty$	+1/2	
$-\infty$	-1/2	
QNAN	QNAN	
SNAN	QNAN	

The atanpi function does not generate any errors.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

cbrt

Computes a cube root of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event cbrt(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event cbrt(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`cbrt` supports the following precisions.

T
`float`
`double`

Description

The cbrt(a)function computes a cube root of vector elements.

Argument	Result	Status code
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
QNAN	QNAN	
SNAN	QNAN	
+0	+0	

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

cdfnorm

Computes the cumulative normal distribution function values of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event cdfnorm(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event cdfnorm(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});
```

(continues on next page)

(continued from previous page)

```
} // namespace oneapi::mkl::vm
```

`cdfnorm` supports the following precisions.

T
`float`
`double`

Description

The cdfnorm function computes the cumulative normal distribution function values for elements of the input vector `a` and writes them to the output vector `y`.

The cumulative normal distribution function is defined as given by:

$$\mathrm{cdfnorm}(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{t^2}{2}} \, \mathrm{d}x$$

Useful relations for these functions:

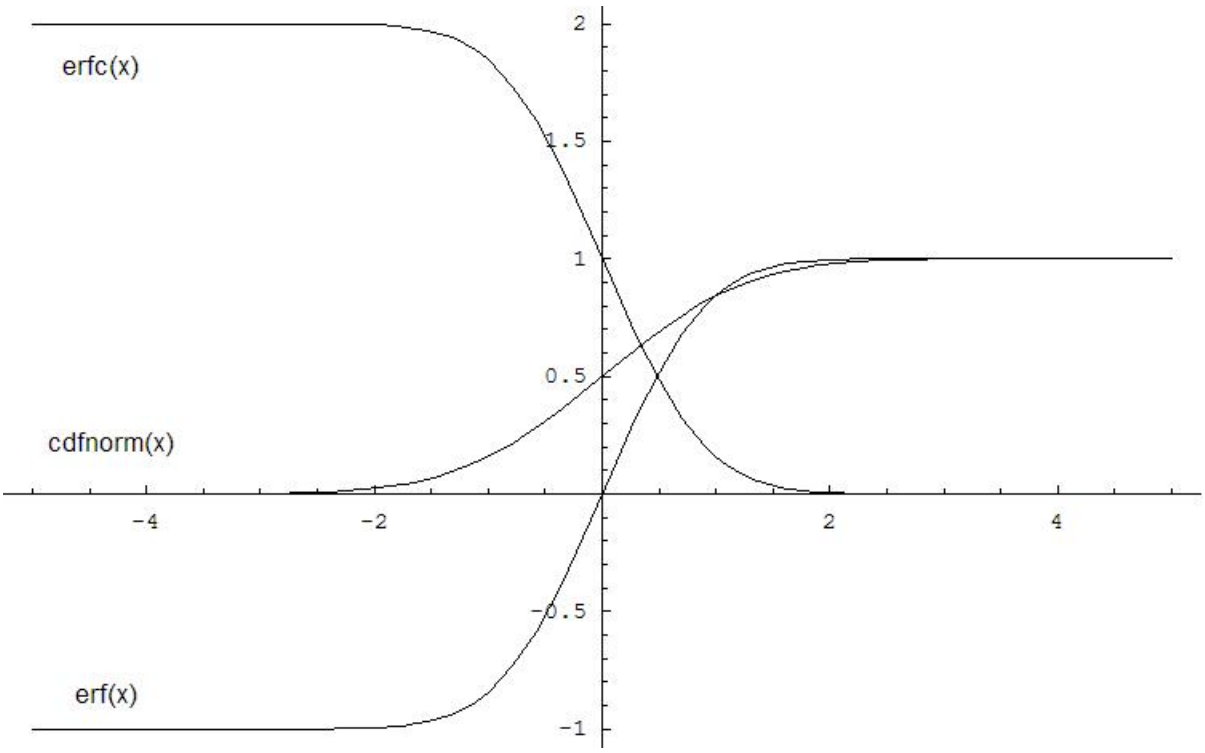
$$\mathrm{erf}(x) + \mathrm{erfc}(x) = 1$$

$$\begin{aligned} \mathrm{cdfnorm}(x) &= \frac{1}{2}\left(1 + \mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right) \\ &= 1 - \frac{1}{2}\,\mathrm{erfc}\left(\frac{x}{\sqrt{2}}\right) \end{aligned}$$

where erf and erfc are the error and complementary error functions, respectively.

The following figure illustrates the relationships among the family of error functions (erf, erfc, cdfnorm).

cdfnorm Family Functions Relationship |



Argument	Result	Status code
a < underflow	+0	`oneapi::mkl::vm::status::underflow`
+∞	+1	
-∞	+0	
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

cdfnorminv

Computes the inverse cumulative normal distribution function values of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event cdfnorminv(
        sycl::queue& exec_queue,
        std::int64_t n,
```

(continues on next page)

(continued from previous page)

```
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event cdfnorminv(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`cdfnorminv` supports the following precisions.

T
float
double

Description

The cdfnorminv(a) function computes the inverse cumulative normal distribution function values for elements of the input vector `a` and writes them to the output vector `y`.

The inverse cumulative normal distribution function is defined as given by:

$$\text{cdfnorminv}(x) = \text{cdfnorm}^{-1}(x)$$

where $\text{cdfnorm}(x)$ denotes the cumulative normal distribution function.

Useful relations:

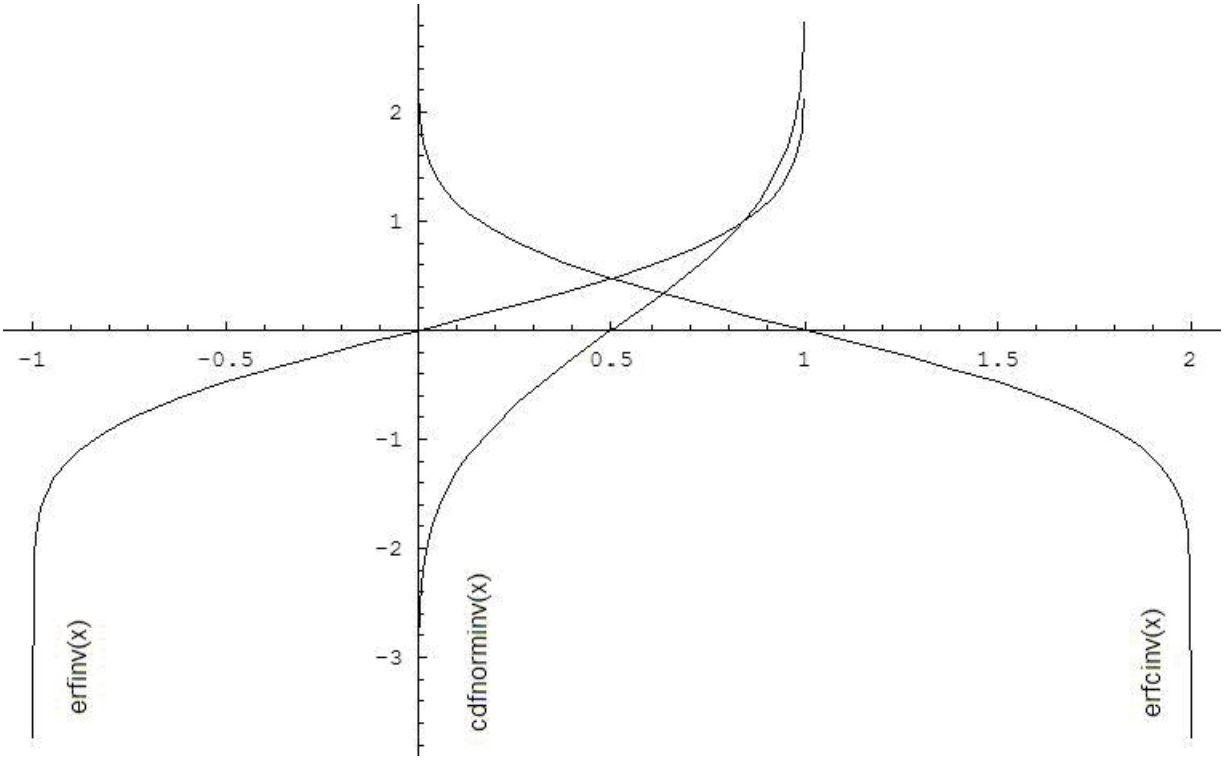
$$\text{erfcinv}(x) = \text{erfinv}(1 - x)$$

$$\begin{aligned}\text{cdfnorminv}(x) &= \sqrt{2}\,\text{erfinv}(2x - 1) \\ &= \sqrt{2}\,\text{erfcinv}(2 - 2x)\end{aligned}$$

where $\text{erfinv}(x)$ denotes the inverse error function and $\text{erfcinv}(x)$ denotes the inverse complementary error function.

The following figure illustrates the relationships among erfinv family functions (erfinv, erfcinv, cdfnorminv).

cdfnorminv Family Functions Relationship |



Argument	Result	Status code
+0.5	+0	
+1	+∞	`oneapi::mkl::vm::status::sing`
-0	-∞	`oneapi::mkl::vm::status::sing`
+0	-∞	`oneapi::mkl::vm::status::sing`
a < -0	QNAN	`oneapi::mkl::vm::status::errdom`
a > +1	QNAN	`oneapi::mkl::vm::status::errdom`
+∞	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

ceil

Computes an integer value rounded towards plus infinity for each vector element.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event ceil(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event ceil(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`ceil` supports the following precisions.

T
`float`
`double`

Description

The ceil(a) function computes an integer value rounded towards plus infinity for each vector element.

$$y_i = \lceil a_i \rceil$$

Argument	Result	Status code
+0	+0	
-0	-0	
$+\infty$	$+\infty$	
$-\infty$	$-\infty$	
QNAN	QNAN	
SNAN	QNAN	

The ceil function does not generate any errors.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

cis

Computes complex exponent of real vector elements (cosine and sine of real vector elements combined to complex value).

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event cis(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<R,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event cis(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        R* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`cis` supports the following precisions.

T	R
float	std::complex<float>
double	std::complex<double>

Description

The cis(a) function computes complex exponent of real vector elements (cosine and sine of real vector elements combined to complex value).

Argument	Result	Status code
• 0	+1+i·0	
• 0	+1-i·0	
• ∞	QNAN+i·QNAN	`oneapi::mkl::vm::status::errdom`
• ∞	QNAN+i·QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN+i·QNAN	
SNAN	QNAN+i·QNAN	

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for

possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

conj

Performs element by element conjugation of the vector.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event conj(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event conj(
        sycl::queue& exec_queue,
```

(continues on next page)

(continued from previous page)

```
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`conj` supports the following precisions.

T
std::complex<float>
std::complex<double>

Description

The conj function performs element by element conjugation of the vector.

No special values are specified. The conj function does not generate any errors.

Input Parameters

Buffer API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
: The buffer `a` containing input vector of size `n`.

mode
: Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
: Pointer `a` to the input vector of size `n`.

depends
: Vector of dependent events (to wait for input data to be ready).

mode
: Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

copysign

Returns vector of elements of one argument with signs changed to match other argument elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event copysign(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event copysign(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        const T *b,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);
```

(continues on next page)

(continued from previous page)

```
} // namespace oneapi::mkl::vm
```

`copysign` supports the following precisions.

T
`float`
`double`

Description

The copysign(a, b) function returns the first vector argument elements with the sign changed to match the sign of the second vector argument's corresponding elements.

Argument 1	Argument 2	Result	Status code
any value	positive value	+any value	
any value	negative value	-any value	

The copysign(a, b) function does not generate any errors.

Input Parameters

Buffer API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
: The buffer `a` containing 1st input vector of size `n`.

b
: The buffer `b` containing 2nd input vector of size `n`.

mode
: Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
: Pointer `a` to the 1st input vector of size `n`.

b
: Pointer `b` to the 2nd input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

cos

Computes cosine of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event cos(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event cos(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`cos` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Description

The cos(a) function computes cosine of vector elements.

Note that arguments `abs(a[i])` $\leq$ `213` and `abs(a[i])` $\leq$ `216` for single and double precisions, respectively, are called fast computational path. These are trigonometric function arguments for which VM provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VM High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VM Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Argument	Result	VM status code
+0	+1	
-0	+1	
+∞	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

`Cos(z) = Cosh(i*z)`.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

cosd

Computes the cosine of vector elements multiplied by $\pi/180$.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event cosd(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event cosd(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`cosd` supports the following precisions.

T
float
double

Description

The cosd(a) function is a degree argument trigonometric function. It computes the cosine of vector elements multiplied by $\pi/180$. For an argument `a`, the function computes cos(π*`a`/180).

Note that arguments abs(a_i) $\leq 2^{24}$ for single precision or abs(a_i) $\leq 2^{52}$ for double precision, they belong to the *fast computational path*: trigonometric function arguments for which VM provides the best possible performance. Avoid arguments with do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

Argument	Result	Status code
+0	+1	
-0	+1	
+∞	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

cosh

Computes hyperbolic cosine of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event cosh(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event cosh(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
```

(continues on next page)

(continued from previous page)

```
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`cosh` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Description

The cosh(a) function computes hyperbolic cosine of vector elements.

Data Type	Threshold Limitations on Input Parameters
single precision	`-Log(FLT_MAX)-Log2 <a[i] < Log(FLT_MAX)+Log2`
double precision	`-Log(DBL_MAX)-Log2 <a[i] < Log(DBL_MAX)+Log2`

Argument	Result	Status code
+0	+1	
-0	+1	
X > overflow	+∞	`oneapi::mkl::vm::status::overflow`
X < -overflow	+∞	`oneapi::mkl::vm::status::overflow`
+∞	+∞	
-∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

+i·∞	+∞+i·QNAN	QNAN+i·QN	QNAN-i·0	QNAN+i·0	QNAN+i·QN	+∞+i·QNA	QNAN+i·QNAN
+i·Y	+∞·Cos(Y)-i·∞·Sin(Y)					+∞·CIS(Y	QNAN+i·QNAN
+i·0	+∞-i·0		+1-i·0	+1+i·0		+∞+i·0	QNAN+i·0
-i·0	+∞+i·0		+1+i·0	+1-i·0		+∞-i·0	QNAN-i·0
-i·Y	+∞·Cos(Y)-i·∞·Sin(Y)					+∞·CIS(Y	QNAN+i·QNAN
-i·∞	+∞+i·QNAN	QNAN+i·QN	QNAN+i·0	QNAN-i·0	QNAN+i·QN	+∞+i·QNA	QNAN+i·QNAN
+i·NAN	+∞+i·QNAN	QNAN+i·QN	QNAN+i·QN	QNAN-i·QNAN	QNAN+i·QN	+∞+i·QNA	QNAN+i·QNAN

Notes:

- **The complex cosh(a) function sets the VM status code to**
  `oneapi::mkl::vm::status::overflow` in the case of overflow, that is, when RE(a), IM(a) are finite non-zero numbers, but the real or imaginary part of the exact result is so large that it does not meet the target precision.
- `cosh(CONJ(a))=CONJ(cosh(a))`
- `cosh(-a)=cosh(a)`.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

cospi

Computes the cosine of vector elements multiplied by π.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event cospi(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event cospi(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});
```

(continues on next page)

(continued from previous page)

```
} // namespace oneapi::mkl::vm
```

`cospi` supports the following precisions.:

T
`float`
`double`

Description

The cospi(a) function computes the cosine of vector elements multiplied by π. For an argument `a`, the function computes cos(π*`a`).

Argument	Result	Status code
+0	+1	
-0	+1	
`n` + 0.5, for any integer `n` where `n` + 0.5 is representable	+0	
+∞	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

If arguments abs(a_i) $\leq 2^{22}$ for single precision or abs(a_i) $\leq 2^{51}$ for double precision, they belong to the *fast computational path*: arguments for which VM provides the best possible performance. Avoid arguments which do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

Input Parameters

Buffer API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
: The buffer `a` containing input vector of size `n`.

mode
: Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
: Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

div

Performs element by element division of vector `a` by vector `b`

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event div(
        sycl::queue& exec_queue,
        std::int64_t n,
```

(continues on next page)

(continued from previous page)

```
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event div(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        const T *b,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`div` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Description

The div(a, b) function performs element by element division of vector `a` by vector `b`.

Argument 1	Argument 2	Result	VM status code
X > +0	+0	+∞	`oneapi::mkl::vm::status::sing`
X > +0	-0	-∞	`oneapi::mkl::vm::status::sing`
X < +0	+0	-∞	`oneapi::mkl::vm::status::sing`
X < +0	-0	+∞	`oneapi::mkl::vm::status::sing`
+0	+0	QNAN	`oneapi::mkl::vm::status::sing`
-0	-0	QNAN	`oneapi::mkl::vm::status::sing`
X > +0	+∞	+0	
X > +0	-∞	-0	
+∞	+∞	QNAN	
-∞	-∞	QNAN	
QNAN	QNAN	QNAN	
SNAN	SNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

`Div(x1+i*y1, x2+i*y2) = (x1+i*y1)*(x2-i*y2)/(x2*x2+y2*y2)`.

Overflow in a complex function occurs when `x2+i*y2` is not zero, x1, x2, y1, y2 are finite numbers, but the real or imaginary part of the exact result is so large that it does not fit the target precision. In that case, the function returns ∞ in that part of the result, and sets the VM status code to `oneapi::mkl::vm::status::overflow`.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing 1st input vector of size `n`.

b
The buffer `b` containing 2nd input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the 1st input vector of size `n`.

b
Pointer `b` to the 2nd input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

erf

Computes the error function value of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event erf(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event erf(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`erf` supports the following precisions.

T
`float`
`double`

Description

The erf function computes the error function values for elements of the input vector `a` and writes them to the output vector `y`.

The error function is defined as given by:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} \, \mathrm{d}t$$

Useful relations:

$$\operatorname{erfc}(x) = 1 - \operatorname{erf}(x)$$

where erfc is the complementary error function.

$$\Phi(x) = \frac{1}{2} \left(1 + \operatorname{erf} \left(x/\sqrt{2} \right) \right)$$

where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x \exp(-t^2/2) \, \mathrm{d}t$$

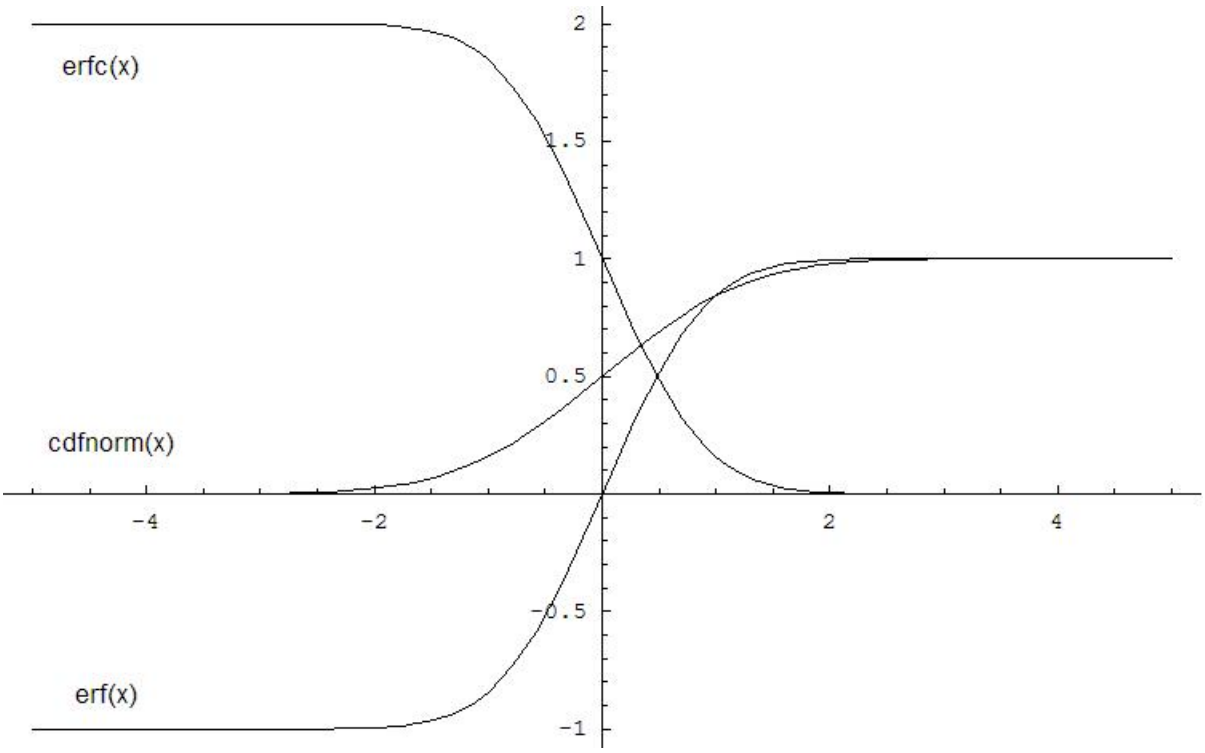
is the cumulative normal distribution function.

$$\Phi^{-1}(x) = \sqrt{2} \operatorname{erf}^{-1}(2x - 1)$$

where $\Phi^{-1}(x)$ and $\operatorname{erf}^{-1}(x)$ are the inverses to $\Phi(x)$ and $\operatorname{erf}(x)$, respectively.

The following figure illustrates the relationships among erf family functions (erf, erfc, cdfnorm).

erf Family Functions Relationship |



Useful relations for these functions:

$$\mathrm{erf}(x) + \mathrm{erfc}(x) = 1$$

$$\begin{aligned}\mathrm{cdfnorm}(x) &= \frac{1}{2}\left(1 + \mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right) \\ &= 1 - \frac{1}{2}\,\mathrm{erfc}\left(\frac{x}{\sqrt{2}}\right)\end{aligned}$$

Argument	Result	Status code
+∞	+1	
-∞	-1	
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

erfc

Computes the complementary error function value of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event erfc(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event erfc(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`erfc` supports the following precisions.

T
float
double

Description

The erfc function computes the complementary error function values for elements of the input vector `a` and writes them to the output vector `y`.

The complementary error function is defined as follows:

$$\operatorname{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} \, \mathrm{d}t$$

Useful relations:

$$\operatorname{erfc}(x) = 1 - \operatorname{erf}(x)$$

$$\Phi(x) = \frac{1}{2}\left(1 + \operatorname{erf}\left(x/\sqrt{2}\right)\right)$$

where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x \exp(-t^2/2)\,\mathrm{d}t$$

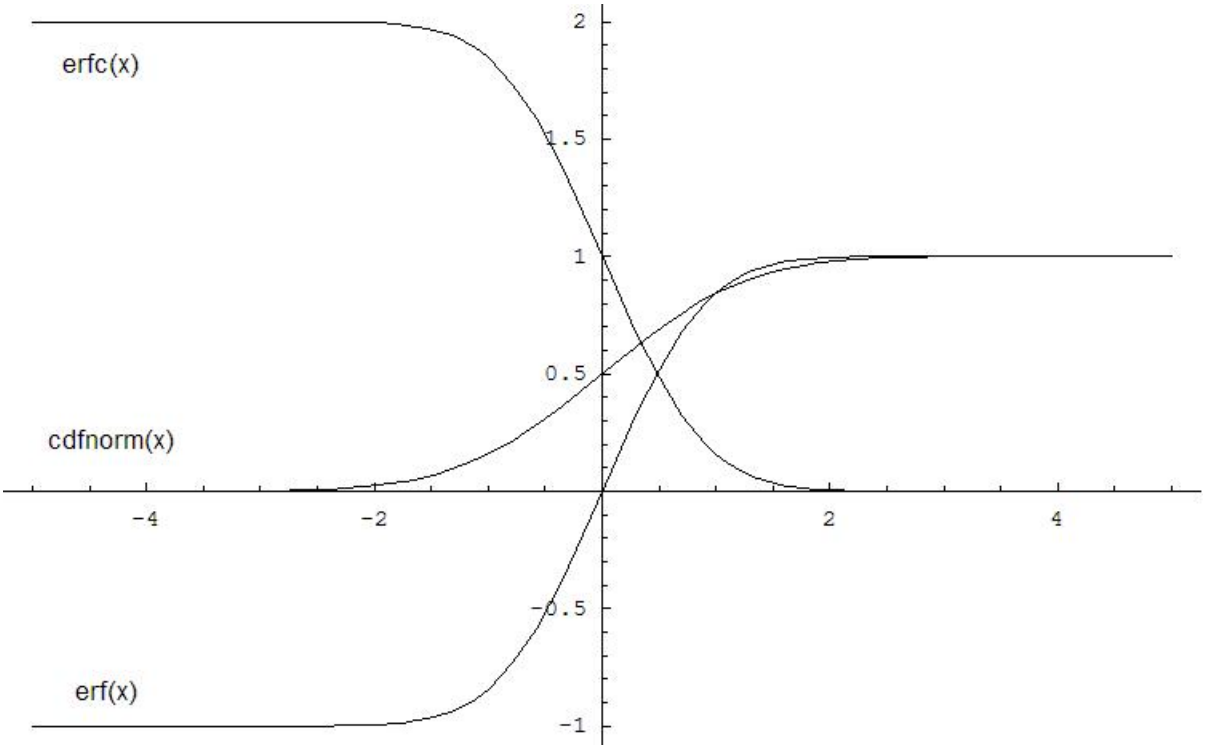
is the cumulative normal distribution function.

$$\Phi^{-1}(x) = \sqrt{2}\,\operatorname{erf}^{-1}(2x - 1)$$

where $\Phi^{-1}(x)$ and $\operatorname{erf}^{-1}(x)$ are the inverses to $\Phi(x)$ and $\operatorname{erf}(x)$, respectively.

The following figure illustrates the relationships among erf family functions (erf, erfc, cdfnorm).

erfc Family Functions Relationship |



Useful relations for these functions:

$$\operatorname{erf}(x) + \operatorname{erfc}(x) = 1$$

$$\begin{aligned}\operatorname{cdfnorm}(x) &= \frac{1}{2}\left(1 + \operatorname{erf}\left(\frac{x}{\sqrt{2}}\right)\right) \\ &= 1 - \frac{1}{2}\operatorname{erfc}\left(\frac{x}{\sqrt{2}}\right)\end{aligned}$$

Argument	Result	Status code
a > underflow	+0	`oneapi::mkl::vm::status::underflow`
+∞	+0	
-∞	+2	
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

erfcinv

Computes the inverse complementary error function value of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event erfcinv(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event erfcinv(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});
```

(continues on next page)

(continued from previous page)

```
} // namespace oneapi::mkl::vm
```

`erfcinv` supports the following precisions.

T
float
double

Description

The erfcinv(a) function computes the inverse complimentary error function values for elements of the input vector `a` and writes them to the output vector `y`.

The inverse complementary error function is defined as given by:

$$\mathrm{erfcinv}(x) = \mathrm{erfinv}(1 - x)$$

Useful relations for these functions:

$$\mathrm{erfcinv}(x) = \mathrm{erfinv}(1 - x)$$

$$\begin{aligned}\mathrm{cdfnorminv}(x) &= \sqrt{2}\,\mathrm{erfinv}(2x - 1) \\ &= \sqrt{2}\,\mathrm{erfcinv}(2 - 2x)\end{aligned}$$

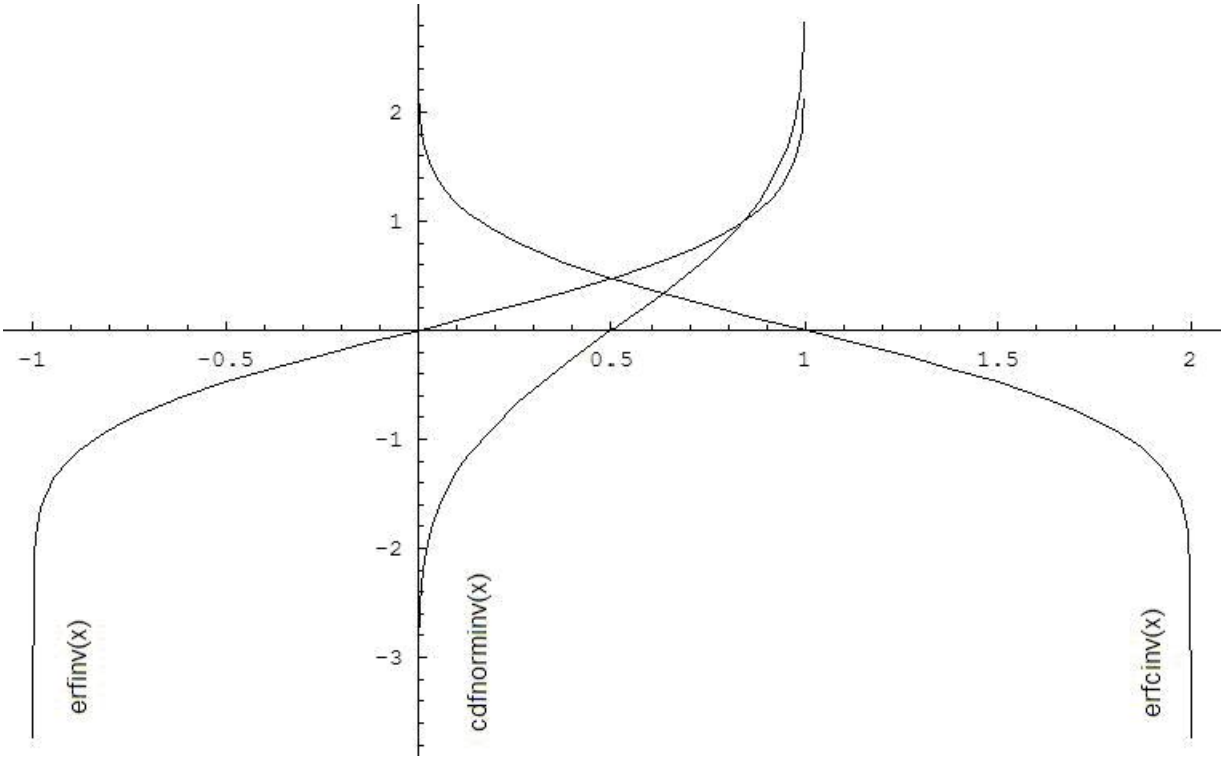
$$\mathrm{erfinv}(x) = \mathrm{erf}^{-1}(x)$$

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\,\mathrm{d}t$$

where $\mathrm{erf}(x)$ denotes the error function and $\mathrm{erfinv}(x)$ denotes the inverse error function.

The following figure illustrates the relationships among erfinv family functions (erfinv, erfcinv, cdfnorminv).

erfcinv Family Functions Relationship |



Argument	Result	Status code
+1	+0	
+2	-∞	`oneapi::mkl::vm::status::sing`
-0	+∞	`oneapi::mkl::vm::status::sing`
+0	+∞	`oneapi::mkl::vm::status::sing`
a < -0	QNAN	`oneapi::mkl::vm::status::errdom`
a > +2	QNAN	`oneapi::mkl::vm::status::errdom`
+∞	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

erfinv

Computes inverse error function value of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event erfinv(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event erfinv(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`erfinv` supports the following precisions.

T
float
double

Description

The erfinv(a) function computes the inverse error function values for elements of the input vector `a` and writes them to the output vector `y`.

$$y_i = \mathrm{erf}^{-1}(a)$$

where $\mathrm{erf}(x)$ is the error function defined as given by:

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\,\mathrm{d}t$$

Useful relations for these functions:

$$\mathrm{erfcinv}(x) = \mathrm{erfinv}(1 - x)$$

$$\begin{aligned}\mathrm{cdfnorminv}(x) &= \sqrt{2}\,\mathrm{erfinv}(2x-1)\\ &= \sqrt{2}\,\mathrm{erfcinv}(2-2x)\\ \mathrm{erf}^{-1}(x) &= \mathrm{erfc}^{-1}(1-x)\end{aligned}$$

where erfc is the complementary error function.

$$\Phi(x) = \frac{1}{2}\left(1 + \mathrm{erf}\left(x/\sqrt{2}\right)\right)$$

where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}}\int_0^x \exp(-t^2/2)\,\mathrm{d}t$$

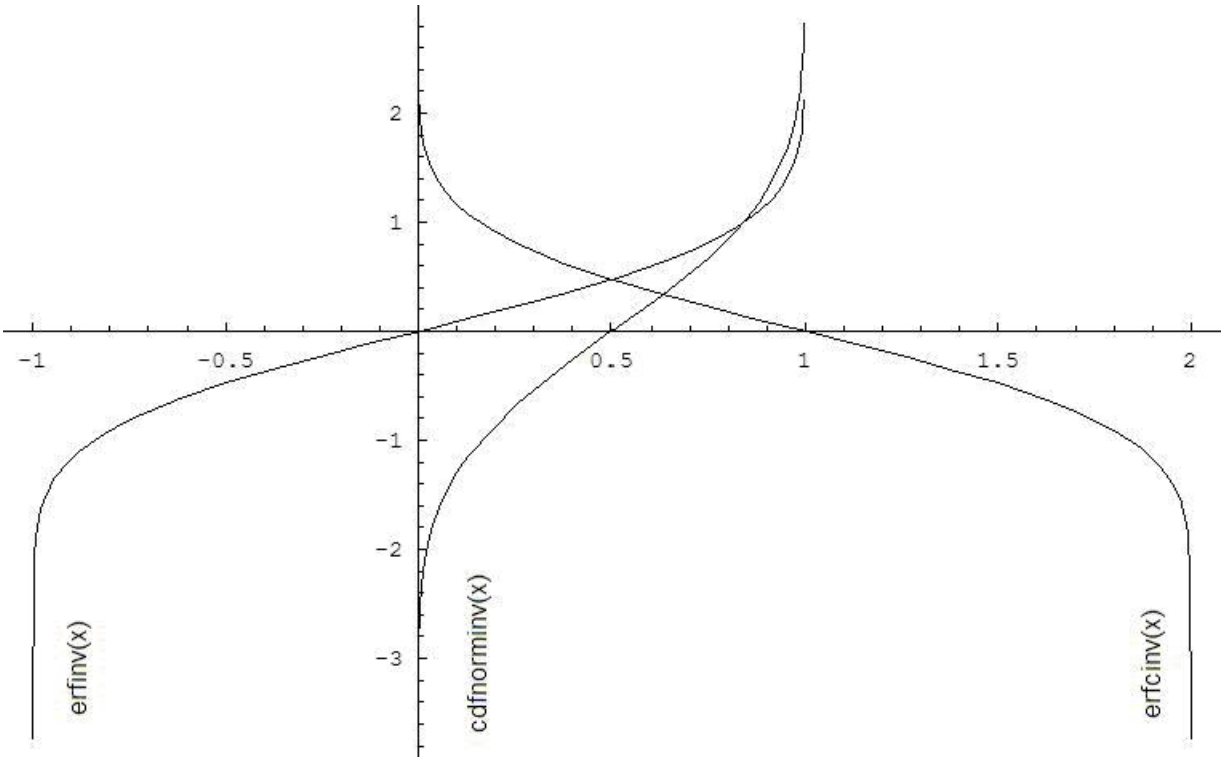
is the cumulative normal distribution function.

$$\Phi^{-1}(x) = \sqrt{2}\,\mathrm{erf}^{-1}(2x-1)$$

where $\Phi^{-1}(x)$ and $\mathrm{erf}^{-1}(x)$ are the inverses to $\Phi(x)$ and $\mathrm{erf}(x)$, respectively.

The following figure illustrates the relationships among erfinv family functions (erfinv, erfcinv, cdfnorminv).

erfinv Family Functions Relationship |



Argument	Result	Status code
+0	+0	
-0	-0	
+1	+∞	`oneapi::mkl::vm::status::sing`
-1	-∞	`oneapi::mkl::vm::status::sing`
|a| > 1	QNAN	`oneapi::mkl::vm::status::errdom`
+∞	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

exp

Computes an exponential of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event exp(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event exp(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

exp supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

Description

The exp(a) function computes an exponential of vector elements.

Data Type	Threshold Limitations on Input Parameters
single precision	`a[i] < Log( FLT_MAX )`
double precision	`a[i] < Log( DBL_MAX )`

Argument	Result	Status code
+0	+1	
-0	+1	
a > overflow	+∞	`oneapi::mkl::vm::status::overflow`
a < underflow	+0	`oneapi::mkl::vm::status::overflow`
+∞	+∞	
-∞	+0	
QNAN	QNAN	
SNAN	QNAN	

+i·∞
+i·Y
+i·0
-i·0
-i·Y
-i·∞
+i·NAN

Notes:

- **The complex exp(z) function sets the VM status code to** `oneapi::mkl::vm::status::overflow` in the case of overflow, that is, when both `RE(z)` and `IM(z)` are finite non-zero numbers, but the real or imaginary part of the exact result is so large that it does not meet the target precision.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

exp10

Computes the base 10 exponential of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event exp10(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event exp10(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`exp10` supports the following precisions.

T
`float`
`double`

Description

The exp10(a) function computes the base 10 exponential of vector elements.

Data Type	Threshold Limitations on Input Parameters
single precision	$a_i < \log_{10}(\text{FLT\_MAX})$
double precision	$a_i < \log_{10}(\text{DBL\_MAX})$

Argument	Result	VM status code
+0	+1	
-0	+1	
a > overflow	+∞	`oneapi::mkl::vm::status::overflow`
a < underflow	+0	`oneapi::mkl::vm::status::underflow`
+∞	+∞	
-∞	+0	
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

exp2

Computes the base 2 exponential of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event exp2(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event exp2(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});
```

(continues on next page)

(continued from previous page)

```
} // namespace oneapi::mkl::vm
```

`exp2` supports the following precisions.

T
`float`
`double`

Description

The exp2 function computes the base 2 exponential of vector elements.

Data Type	Threshold Limitations on Input Parameters
single precision	$a_i < \log_2(\text{FLT\_MAX})$
double precision	$a_i < \log_2(\text{DBL\_MAX})$

Argument	Result	Status code
+0	+1	
-0	+1	
a > overflow	+∞	`oneapi::mkl::vm::status::overflow`
a < underflow	+0	`oneapi::mkl::vm::status::underflow`
+∞	+∞	
-∞	+0	
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
: The buffer `a` containing input vector of size `n`.

mode
: Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
: Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

expint1

Computes the exponential integral of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event expint1(
        sycl::queue& exec_queue,
        std::int64_t n,
```

(continues on next page)

(continued from previous page)

```
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event expint1(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`expint1` supports the following precisions.

T
`float`
`double`

Description

The expint1(a) function computes the exponential integral of vector elements of the input vector `a` and writes them to the output vector `y`.

For positive real values `x`, this can be written as:

$$E_1(x) = \int_x^\infty \frac{e^{-t}}{t}\,\mathrm{d}t = \int_1^\infty \frac{e^{-xt}}{t}\,\mathrm{d}t$$

For negative real values `x`, the result is defined as NAN.

Argument	Result	Status code
x < +0	QNAN	`oneapi::mkl::vm::status::errdom`
+0	+∞	`oneapi::mkl::vm::status::sing`
-0	+∞	`oneapi::mkl::vm::status::sing`
+∞	+0	
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

expm1

Computes an exponential of vector elements decreased by 1. `exp(a[i]) - 1`

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event expm1(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event expm1(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`expm1` supports the following precisions.

T
float
double

Description

The expm1(a) function computes an exponential of vector elements decreased by 1.

Argument	Result	Status code
+0	+1	
-0	+1	
a > overflow	+∞	`oneapi::mkl::vm::status::overflow`
+∞	+∞	
-∞	-0	
QNAN	QNAN	
SNAN	QNAN	

Data Type	Threshold Limitations on Input Parameters
single precision	`a[i] < Log( FLT_MAX )`
double precision	`a[i] < Log( DBL_MAX )`

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for

possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

fdim

Returns vector containing the differences of the corresponding elements of the vector arguments if the first is larger and +0 otherwise.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event fdim(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event fdim(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        const T *b,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`fdim` supports the following precisions.

T
`float`
`double`

Description

The fdim(a, b) function returns a vector containing the differences of the corresponding elements of the first and second vector arguments if the first element is larger, and +0 otherwise.

Argument 1	Argument 2	Result	Status code
any	QNAN	QNAN	
any	SNAN	QNAN	
QNAN	any	QNAN	
SNAN	any	QNAN	

The fdim(a, b) function does not generate any errors.

Input Parameters

Buffer API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
: The buffer `a` containing 1st input vector of size `n`.

b
: The buffer `b` containing 2nd input vector of size `n`.

mode
: Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the 1st input vector of size `n`.

b
Pointer `b` to the 2nd input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

floor

Computes an integer value rounded towards minus infinity for each vector element.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event floor(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event floor(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`floor` supports the following precisions.

T
`float`
`double`

Description

The floor(a)function computes an integer value rounded towards minus infinity for each vector element.

$$y_i = \lfloor a_i \rfloor$$

Argument	Result	Status code
+0	+0	
-0	-0	
$+\infty$	$+\infty$	
$-\infty$	$-\infty$	
QNAN	QNAN	
SNAN	QNAN	

The floor function does not generate any errors.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

fmax

Returns the larger of each pair of elements of the two vector arguments.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event fmax(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event fmax(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        const T *b,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`fmax` supports the following precisions.

T
float
double

Description

The fmax(a, b) function returns a vector with element values equal to the larger value from each pair of corresponding elements of the two vectors `a` and `b`: if `a` < `b`fmax(a, b) returns `b`, otherwise fmax(a, b) returns `a`.

Argument 1	Argument 2	Result	Status code
`a` not NAN	NAN	`a`	
NAN	`b` not NAN	`b`	
NAN	NAN	NAN	

The fmax(a, b) function does not generate any errors.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing 1st input vector of size `n`.

b
The buffer `b` containing 2nd input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the 1st input vector of size `n`.

b
Pointer `b` to the 2nd input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

fmin

Returns the smaller of each pair of elements of the two vector arguments.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event fmin(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event fmin(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        const T *b,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);
```

(continues on next page)

(continued from previous page)

```
} // namespace oneapi::mkl::vm
```

`fmin` supports the following precisions.

T
`float`
`double`

Description

The fmin(a, b) function returns a vector with element values equal to the smaller value from each pair of corresponding elements of the two vectors `a` and `b`: if `a` > `b`fmin(a, b) returns `b`, otherwise fmin(a, b) returns `a`.

Argument 1	Argument 2	Result	Status code
`a` not NAN	NAN	`a`	
NAN	`b` not NAN	`b`	
NAN	NAN	NAN	

The fmin(a, b) function does not generate any errors.

Input Parameters

Buffer API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
: The buffer `a` containing 1st input vector of size `n`.

b
: The buffer `b` containing 2nd input vector of size `n`.

mode
: Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
: Pointer `a` to the 1st input vector of size `n`.

b
: Pointer `b` to the 2nd input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

fmod

The fmod function performs element by element computation of the modulus function of vector `a` with respect to vector `b`.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event fmod(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event fmod(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        const T *b,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`fmod` supports the following precisions.

T
`float`
`double`

Description

The fmod (a, b) function computes the modulus function of each element of vector `a`, with respect to the corresponding elements of vector `b`:

`ai - bi*trunc(ai/bi)`

In general, the modulus function `fmod (ai, bi)` returns the value `ai - n*bi` for some integer `n` such that if b_i is nonzero, the result has the same sign as a_i and a magnitude less than the magnitude of b_i.

Argument 1	Argument 2	Result	Status code
a not NAN	± 0	NAN	`oneapi::mkl::vm::status::sing`
$\pm\infty$	b not NAN	NAN	`oneapi::mkl::vm::status::sing`
± 0	b$\neq 0$, not NAN	± 0	
a finite	$\pm\infty$	a	
NAN	b		
a	NAN	NAN	

Input Parameters

Buffer API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
: The buffer `a` containing 1st input vector of size `n`.

b
The buffer `b` containing 2nd input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the 1st input vector of size `n`.

b
Pointer `b` to the 2nd input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

frac

Computes a signed fractional part for each vector element.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event frac(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event frac(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`frac` supports the following precisions.

T
float
double

Description

The frac(a) function computes a signed fractional part for each vector element.

$$y_i = \begin{cases} a_i - \lfloor a_i \rfloor, a_i \geq 0 \\ a_i - \lceil a_i \rceil, a_i < 0 \end{cases}$$

Argument	Result	Status code
+0	+0	
-0	-0	
$+\infty$	+0	
$-\infty$	-0	
QNAN	QNAN	
SNAN	QNAN	

The frac function does not generate any errors.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

hypot

Computes a square root of sum of two squared elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event hypot(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event hypot(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        const T *b,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);
```

(continues on next page)

(continued from previous page)

```
} // namespace oneapi::mkl::vm
```

`hypot` supports the following precisions.

T
`float`
`double`

Description

The function hypot(a, b) computes a square root of sum of two squared elements.

Argument 1	Argument 2	Result	Status code
+0	+0	+0	
-0	-0	+0	
+∞	any value	+∞	
any value	+∞	+∞	
SNAN	any value	QNAN	INVALID
any value	SNAN	QNAN	INVALID
QNAN	any value	QNAN	
any value	QNAN	QNAN	

Data Type	Threshold Limitations on Input Parameters
single precision	`abs(a[i]) < sqrt(FLT_MAX)abs(b[i]) < sqrt(FLT_MAX)`
double precision	`abs(a[i]) < sqrt(DBL_MAX)abs(b[i]) < sqrt(DBL_MAX)`

The hypot(a, b) function does not generate any errors.

Input Parameters

Buffer API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
: The buffer `a` containing 1st input vector of size `n`.

b
: The buffer `b` containing 2nd input vector of size `n`.

mode
: Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the 1st input vector of size `n`.

b
Pointer `b` to the 2nd input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

inv

Performs element by element inversion of the vector.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event inv(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
```

(continues on next page)

(continued from previous page)

```
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event inv(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`inv` supports the following precisions.

T
`float`
`double`

Description

The inv(a) function performs element by element inversion of the vector.

Argument	Result	VM status code
+0	+∞	`oneapi::mkl::vm::status::sing`
-0	-∞	`oneapi::mkl::vm::status::sing`
+∞	+0	
-∞	-0	
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

invcbrt

Computes an inverse cube root of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event invcbrt(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event invcbrt(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`invcbrt` supports the following precisions.

T
float
double

Description

The invcbrt(a)function computes an inverse cube root of vector elements.

Argument	Result	Status code
+0	+∞	`oneapi::mkl::vm::status::sing`
-0	-∞	`oneapi::mkl::vm::status::sing`
+∞	+0	
-∞	-0	
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

invsqrt

Computes an inverse square root of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event invsqrt(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event invsqrt(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});
```

(continues on next page)

(continued from previous page)

```
} // namespace oneapi::mkl::vm
```

`invsqrt` supports the following precisions.

T
`float`
`double`

Description

The invsqrt(a) function computes an inverse square root of vector elements.

Argument	Result	VM status code
a < +0	QNAN	`oneapi::mkl::vm::status::errdom`
+0	+∞	`oneapi::mkl::vm::status::sing`
-0	-∞	`oneapi::mkl::vm::status::sing`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
+∞	+0	
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
: The buffer `a` containing input vector of size `n`.

mode
: Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
: Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
: Pointer `a` to the 1st input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

lgamma

Computes the natural logarithm of the absolute value of gamma function for vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event lgamma(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event lgamma(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`lgamma` supports the following precisions.

T
`float`
`double`

Description

The lgamma(a) function computes the natural logarithm of the absolute value of gamma function for elements of the input vector `a` and writes them to the output vector `y`. Precision overflow thresholds for the lgamma function are beyond the scope of this document. If the result does not meet the target precision, the function sets the VM status code to `oneapi::mkl::vm::status::overflow`.

Argument	Result	VM status code
+1	+0	
+2	+0	
+0	+∞	`oneapi::mkl::vm::status::sing`
-0	+∞	`oneapi::mkl::vm::status::sing`
negative integer	+∞	`oneapi::mkl::vm::status::sing`
-∞	+∞	
+∞	+∞	
a > overflow	+∞	`oneapi::mkl::vm::status::overflow`
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
: The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

linearfrac

Performs linear fraction transformation of vectors a and b with scalar parameters.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event linearfrac(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        T scalea,
        T shifta,
        T scaleb,
        T shiftb,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event linearfrac(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        const T *b,
        T scalea,
        T shifta,
        T scaleb,
        T shiftb,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`linearfrac` supports the following precisions.

T
float
double

Description

The linearfrac(a, b, scalea, shifta, scaleb, shiftb) function performs a linear fraction transformation of vector a by vector b with scalar parameters: scaling multipliers scalea, scaleb and shifting addends shifta, shiftb:

y[i]=(scalea·a[i]+shifta)/(scaleb·b[i]+shiftb), i=1,2 … n

The linearfrac function is implemented in the EP accuracy mode only, therefore no special values are defined for this function. If used in HA or LA mode, linearfrac sets the VM status code to oneapi::mkl::vm::status::accuracy_warning. Correctness is guaranteed within the threshold limitations defined for each input parameter (see the table below); otherwise, the behavior is unspecified.

Threshold Limitations on Input Parameters
2EMIN/2≤ |scalea| ≤ 2(EMAX-2)/2
2EMIN/2≤ |scaleb| ≤ 2(EMAX-2)/2
|shifta| ≤ 2EMAX-2
|shiftb| ≤ 2EMAX-2
2EMIN/2≤a[i] ≤ 2(EMAX-2)/2
2EMIN/2≤b[i] ≤ 2(EMAX-2)/2
a[i] ≠ - (shifta/scalea)*(1-δ1), \|δ1| ≤ 21-(p-1)/2
b[i] ≠ - (shiftb/scaleb)*(1-δ2), \|δ2| ≤ 21-(p-1)/2

EMINand EMAX are the minimum and maximum exponents and p is the number of significant bits (precision) for the corresponding data type according to the ANSI/IEEE Standard 754-2008 ([*Bibliography*]):

- for single precisionEMIN = -126, EMAX = 127, p = 24
- **for double precisionEMIN = -1022, EMAX = 1023, p =** 53

The thresholds become less strict for common cases with scalea=0 and/or scaleb=0:

- **ifscalea=0, there are no limitations for the values of a[i]** and shifta.
- **ifscaleb=0, there are no limitations for the values of b[i]** and shiftb.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer a containing 1st input vector of size n.

b
The buffer b containing 2nd input vector of size n.

scalea
Constant value for scaling multipliers of vector a

shifta
Constant value for shifting addend of vector `a`

scaleb
Constant value for scaling multipliers of vector `b`

shiftb
Constant value for shifting addend of vector `b`

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The pointer `a` to the 1st input vector of size `n`.

b
The pointer `b` to the 2nd input vector of size `n`.

scalea
Constant value for scaling multipliers of vector `a`

shifta
Constant value for shifting addend of vector `a`

scaleb
Constant value for scaling multipliers of vector `b`

shiftb
Constant value for shifting addend of vector `b`

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

ln

Computes natural logarithm of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event ln(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event ln(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});
```

(continues on next page)

(continued from previous page)

```
} // namespace oneapi::mkl::vm
```

ln supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

Description

The ln(a)function computes natural logarithm of vector elements.

Argument	Result	Status code
+1	+0	
a <+0	QNAN	oneapi::mkl::vm::status::errdom
+0	-∞	oneapi::mkl::vm::status::sing
-0	-∞	oneapi::mkl::vm::status::sing
-∞	QNAN	oneapi::mkl::vm::status::errdom
+∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

RE(a) i·IM(a)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	$+\infty + i \cdot \frac{3\pi}{4}$	+∞+i·π/2	+∞+i·π/2	+∞+i·π/2	+∞+i·π/2	+∞+i·π/4	+∞+i·QNAN
+i·Y	+∞-i·π					+∞+i·0	QNAN+i·QNAN
+i·0	+∞-i·π		-∞+i·π	-∞-i·0		+∞+i·0	QNAN+i·QNAN
-i·0	+∞-i·π		-∞+i·π	-∞-i·0		+∞-i·0	QNAN+i·QNAN
-i·Y	+∞-i·π					+∞-i·0	QNAN+i·QNAN
-i·∞	$+\infty - i \cdot \frac{3\pi}{4}$	+∞-i·π/2	+∞-i·π/2	+∞-i·π/2	+∞-i·π/2	+∞-i·π/4	+∞+i·QNAN
+i·NAN	+∞+i·QNA	QNAN+i·QN/	QNAN+i·QN/	QNAN+i·QN/	QNAN+i·QN/	+∞+i·QNA	QNAN+i·QNAN

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

log10

Computes the base 10 logarithm of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event log10(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event log10(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`log10` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

Description

The log10(a) function computes the base 10 logarithm of vector elements.

Argument	Result	Status code
+1	+0	
a <+0	QNAN	`oneapi::mkl::vm::status::errdom`
+0	-∞	`oneapi::mkl::vm::status::sing`
-0	-∞	`oneapi::mkl::vm::status::sing`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
+∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

RE(a) i·IM(a)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	$+\infty + i\frac{3}{4}\frac{\pi}{\ln 10}$	$+\infty + i\frac{\pi}{2}\frac{1}{\ln 10}$	$+\infty + i\frac{\pi}{2}\frac{1}{\ln 10}$	$+\infty + i\frac{\pi}{2}\frac{1}{\ln 10}$	$+\infty + i\frac{\pi}{2}\frac{1}{\ln 10}$	$+\infty + i\frac{\pi}{4}\frac{1}{\ln 10}$	+∞+i·QNAN
+i·Y	$+\infty + i\frac{\pi}{\ln 10}$					+∞+i·0	QNAN+i·QNAN
+i·0	$+\infty + i\frac{\pi}{\ln 10}$		$-\infty + i\frac{\pi}{\ln 10}$	-∞+i·0		+∞+i·0	QNAN+i·QNAN
-i·0	$+\infty - i\frac{\pi}{\ln 10}$		$-\infty - i\frac{\pi}{\ln 10}$	-∞-i·0		+∞-i·0	QNAN-i·QNAN
-i·Y	$+\infty - i\frac{\pi}{\ln 10}$					+∞-i·0	QNAN+i·QNAN
-i·∞	$+\infty + i\frac{3}{4}\frac{\pi}{\ln 10}$	$+\infty - i\frac{\pi}{2}\frac{1}{\ln 10}$	$+\infty - i\frac{\pi}{2}\frac{1}{\ln 10}$	$+\infty - i\frac{\pi}{2}\frac{1}{\ln 10}$	$+\infty - i\frac{\pi}{2}\frac{1}{\ln 10}$	$+\infty - i\frac{\pi}{4}\frac{1}{\ln 10}$	+∞+i·QNAN
+i·NAN	+∞+i·QNA	QNAN+i·QN	QNAN+i·QN	QNAN+i·QN	QNAN+i·QN	+∞+i·QNA	QNAN+i·QNAN

Input Parameters

Buffer API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
: The buffer `a` containing input vector of size `n`.

mode
: Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
: Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

log1p

Computes a natural logarithm of vector elements that are increased by 1. `log(a[i] + 1)`

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event log1p(
        sycl::queue& exec_queue,
        std::int64_t n,
```

(continues on next page)

(continued from previous page)

```
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event log1p(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`log1p` supports the following precisions.

T
`float`
`double`

Description

The log1p(a) function computes a natural logarithm of vector elements that are increased by 1.

Argument	Result	VM status code
-1	-∞	`oneapi::mkl::vm::status::sing`
a <-1	QNAN	`oneapi::mkl::vm::status::errdom`
+0	+0	
-0	-0	
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
+∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

log2

Computes the base 2 logarithm of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event log2(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event log2(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`log2` supports the following precisions.

T
float
double

Description

The log2(a) function computes the base 2 logarithm of vector elements.

Argument	Result	Status code
+1	+0	
a < +0	QNAN	`oneapi::mkl::vm::status::errdom`
+0	-∞	`oneapi::mkl::vm::status::sing`
-0	-∞	`oneapi::mkl::vm::status::sing`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
+∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

logb

Computes the exponents of the elements of input vector `a`.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event logb(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event logb(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});
```

(continues on next page)

(continued from previous page)

```
} // namespace oneapi::mkl::vm
```

`logb` supports the following precisions.

T
`float`
`double`

Description

The logb(a) function computes the exponents of the elements of the input vector `a`. For each element a_i of vector `a`, this is the integral part of $\log_2|a_i|$. The returned value is exact and is independent of the current rounding direction mode.

Argument	Result	VM status code
+0	+∞	`oneapi::mkl::vm::status::errdom`
-0	-∞	`oneapi::mkl::vm::status::errdom`
-∞	+∞	
+∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
: The buffer `a` containing input vector of size `n`.

mode
: Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
: Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
: Pointer `a` to the 1st input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

maxmag

Returns the element with the larger magnitude between each pair of elements of the two vector arguments.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event maxmag(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event maxmag(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        const T *b,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`maxmag` supports the following precisions.

T
`float`
`double`

Description

The maxmag(a, b) function returns a vector with element values equal to the element with the larger magnitude from each pair of corresponding elements of the two vectors `a` and `b`:

- **If |a| > |b| maxmag(a, b) returns a,**
  otherwise maxmag(a, b) returns `b`.
- **If |b| > |a| maxmag(a, b) returns b,**
  otherwise maxmag(a, b) returns `a`.
- Otherwise maxmag(a, b) behaves like fmax.

Argument 1	Argument 2	Result	Status code
a not NAN	NAN	a	
NAN	b not NAN	b	
NAN	NAN	NAN	

The maxmag(a, b) function does not generate any errors.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing 1st input vector of size `n`.

b
The buffer `b` containing 2nd input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the 1st input vector of size `n`.

b
Pointer `b` to the 2nd input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

minmag

Returns the element with the smaller magnitude between each pair of elements of the two vector arguments.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event minmag(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event minmag(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        const T *b,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`minmag` supports the following precisions.

T
float
double

Description

The minmag(a, b) function returns a vector with element values equal to the element with the smaller magnitude from each pair of corresponding elements of the two vectors a and b:

- **If |a| < |b| minmag(a, b) returns a,**
  otherwise minmag(a, b) returns b.
- **If |b| < |a| minmag(a, b) returns b,**
  otherwise minmag(a, b) returns a.
- Otherwise minmag behaves like fmin.

Argument 1	Argument 2	Result	Status code
a not NAN	NAN	a	
NAN	b not NAN	b	
NAN	NAN	NAN	

The minmag(a, b) function does not generate any errors.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing 1st input vector of size `n`.

b
The buffer `b` containing 2nd input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the 1st input vector of size `n`.

b
Pointer `b` to the 2nd input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

modf

Computes a truncated integer value and the remaining fraction part for each vector element.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event modf(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        sycl::buffer<T,1>& z,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event modf(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        T* z,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);
```

(continues on next page)

(continued from previous page)

```
} // namespace oneapi::mkl::vm
```

`modf` supports the following precisions.

T
`float`
`double`

Description

The modf(a) function computes a truncated integer value and the remaining fraction part for each vector element.

$$a_i \geq 0, \begin{cases} y_i = \lfloor a_i \rfloor \\ z_i = a_i - \lfloor a_i \rfloor \end{cases}$$

$$a_i < 0, \begin{cases} y_i = \lceil a_i \rceil \\ z_i = a_i - \lceil a_i \rceil \end{cases}$$

Argument	Result 1	Result 2	Status code
+0	+0	+0	
-0	-0	-0	
+∞	+∞	+0	
-∞	-∞	-0	
SNAN	QNAN	QNAN	
QNAN	QNAN	QNAN	

The modf function does not generate any errors.

Input Parameters

Buffer API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
: The buffer `a` containing input vector of size `n`.

mode
: Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue
: The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n` for truncated integer values.

z
The buffer `z` containing the output vector of size `n` for remaining fraction parts.

USM API:

y
Pointer `y` to the output vector of size `n` for truncated integer values.

z
Pointer `z` to the output vector of size `n` for remaining fraction parts.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

mul

Performs element by element multiplication of vector `a` and vector `b`.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event mul(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
```

(continues on next page)

(continued from previous page)

```
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event mul(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        const T *b,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`mul` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

Description

The mul(a, b) function performs element by element multiplication of vector `a` and vector `b`.

Argument 1	Argument 2	Result	Status code
+0	+0	+0	
+0	-0	-0	
-0	+0	-0	
-0	-0	+0	
+0	+∞	QNAN	
+0	-∞	QNAN	
-0	+∞	QNAN	
-0	-∞	QNAN	
+∞	+0	QNAN	
+∞	-0	QNAN	
-∞	+0	QNAN	
-∞	-0	QNAN	
+∞	+∞	+∞	
+∞	-∞	-∞	
-∞	+∞	-∞	
-∞	-∞	+∞	
SNAN	any value	QNAN	
any value	SNAN	QNAN	
QNAN	non-SNAN	QNAN	
non-SNAN	QNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

```
mul(x1+i*y1, x2+i*y2) = (x1*x2-y1*y2) + i*(x1*y2+y1*x2)
```

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when all RE(x), RE(y), IM(x), IM(y) arguments are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns ∞ in that part of the result, and sets the VM status code to `oneapi::mkl::vm::status::overflow` (overriding any possible `oneapi::mkl::vm::status::accuracy_warning` status).

Input Parameters

Buffer API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
: The buffer `a` containing 1st input vector of size `n`.

b
: The buffer `b` containing 2nd input vector of size `n`.

mode
: Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
: Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the 1st input vector of size `n`.

b
Pointer `b` to the 2nd input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

mulbyconj

Performs element by element multiplication of vector `a` element and conjugated vector `b` element.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event mulbyconj(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event mulbyconj(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        const T *b,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`mulbyconj` supports the following precisions.

T
std::complex<float>
std::complex<double>

Input Parameters

Buffer API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
: The buffer `a` containing 1st input vector of size `n`.

b
: The buffer `b` containing 2nd input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the 1st input vector of size `n`.

b
Pointer `b` to the 2nd input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

nearbyint

Computes a rounded integer value in the current rounding mode for each vector element.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event nearbyint(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event nearbyint(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`nearbyint` supports the following precisions.

T
`float`
`double`

Description

The nearbyint(a) function computes a rounded integer value in a current rounding mode for each vector element.

Argument	Result	Status code
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
QNAN	QNAN	
SNAN	QNAN	

The nearbyint function does not generate any errors.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

nextafter

Returns vector of elements containing the next representable floating-point values following the values from the elements of one vector in the direction of the corresponding elements of another vector.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event nextafter(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event nextafter(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        const T *b,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`nextafter` supports the following precisions.

T
float
double

Description

The nextafter(a, b) function returns a vector containing the next representable floating-point values following the first vector argument elements in the direction of the second vector argument's corresponding elements.

Arguments/Results	Status code
Input vector argument element is finite and the corresponding result vector element value is infinite	`oneapi::mkl::vm::status::overflow`
Result vector element value is subnormal or zero, and different from the corresponding input vector argument element	`oneapi::mkl::vm::status::underflow`

Even though underflow or overflow can occur, the returned value is independent of the current rounding direction mode.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing 1st input vector of size `n`.

b
The buffer `b` containing 2nd input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the 1st input vector of size `n`.

b
Pointer `b` to the 2nd input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

pow

Computes `a` to the power `b` for elements of two vectors.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event pow(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event pow(
        sycl::queue& exec_queue,
        std::int64_t n,
```

(continues on next page)

(continued from previous page)

```
        const T *a,
        const T *b,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

pow supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

Description

The pow(a, b) function computes a to the power b for elements of two vectors.

The real function pow has certain limitations on the input range of a and b parameters. Specifically, if a[i] is positive, then b[i] may be arbitrary. For negative a[i], the value of b[i] must be an integer (either positive or negative).

The complex function pow has no input range limitations.

Argument 1	Argument 2	Result	Status code
+0	neg. odd integer	+∞	oneapi::mkl::vm::status::errdom
-0	neg. odd integer	-∞	oneapi::mkl::vm::status::errdom
+0	neg. even integer	+∞	oneapi::mkl::vm::status::errdom
-0	neg. even integer	+∞	oneapi::mkl::vm::status::errdom
+0	neg. non-integer	+∞	oneapi::mkl::vm::status::errdom
-0	neg. non-integer	+∞	oneapi::mkl::vm::status::errdom
-0	pos. odd integer	+0	
-0	pos. odd integer	-0	
+0	pos. even integer	+0	
-0	pos. even integer	+0	
+0	pos. non-integer	+0	
-0	pos. non-integer	+0	
-1	+∞	+1	
-1	-∞	+1	
+1	any value	+1	
+1	+0	+1	
+1	-0	+1	
+1	+∞	+1	
+1	-∞	+1	
+1	QNAN	+1	
any value	+0	+1	
+0	+0	+1	
-0	+0	+1	

continues on next page

Table 14 – continued from previous page

Argument 1	Argument 2	Result	Status code
+∞	+0	+1	
-∞	+0	+1	
QNAN	+0	+1	
any value	-0	+1	
+0	-0	+1	
-0	-0	+1	
+∞	-0	+1	
-∞	-0	+1	
QNAN	-0	+1	
a < +0	non-integer	QNAN	`oneapi::mkl::vm::status::errdom`
|a| < 1	-∞	+∞	
+0	-∞	+∞	`oneapi::mkl::vm::status::errdom`
-0	-∞	+∞	`oneapi::mkl::vm::status::errdom`
|a| > 1	-∞	+0	
+∞	-∞	+0	
-∞	-∞	+0	
|a| < 1	+∞	+0	
+0	+∞	+0	
-0	+∞	+0	
|a| > 1	+∞	+∞	
+∞	+∞	+∞	
-∞	+∞	+∞	
-∞	neg. odd integer	-0	
-∞	neg. even integer	+0	
-∞	neg. non-integer	+0	
-∞	pos. odd integer	-∞	
-∞	pos. even integer	+∞	
-∞	pos. non-integer	+∞	
+∞	b < +0	+0	
+∞	b > +0	+∞	
Big finite value*	Big finite value*	+/-∞	`oneapi::mkl::vm::status::overflow`
QNAN	QNAN	QNAN	
QNAN	SNAN	QNAN	
SNAN	QNAN	QNAN	
SNAN	SNAN	QNAN	

* Overflow in a real function is supported only in the HA/LA accuracy modes. The overflow occurs when x and y are finite numbers, but the result is too large to fit the target precision. In this case, the function:

1. Returns ∞ in the result.
2. Sets the VM status code to `oneapi::mkl::vm::status::overflow`.

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when all RE(x), RE(y), IM(x), IM(y) arguments are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns ∞ in that part of the result, and sets the VM status code to `oneapi::mkl::vm::status::overflow` (overriding any possible `oneapi::mkl::vm::status::accuracy_warning` status).

The complex double precision versions of this function are implemented in the EP accuracy mode only. If used in HA or LA mode, the functions set the VM status code to `oneapi::mkl::vm::status::accuracy_warning`.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing 1st input vector of size `n`.

b
The buffer `b` containing 2nd input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the 1st input vector of size `n`.

b
Pointer `b` to the 2nd input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

pow2o3

Computes the cube root of the square of each vector element.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event pow2o3(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event pow2o3(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

pow2o3 supports the following precisions.

T
float
double

Description

The pow2o3(a)function computes the cube root of the square of each vector element.

Argument	Result	Status code
+0	+0	
-0	+0	
+∞	+∞	
-∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

pow3o2

Computes the square root of the cube of each vector element.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event pow3o2(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event pow3o2(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});
```

(continues on next page)

(continued from previous page)

```
} // namespace oneapi::mkl::vm
```

pow3o2 supports the following precisions.

T
float
double

Description

The pow3o2(a)function computes the square root of the cube of each vector element.

Data Type	Threshold Limitations on Input Parameters
single precision	$\|a_i\|$ < (FLT_MAX)2/3
double precision	$\|a_i\|$ < (FLT_MAX)2/3

Argument	Result	VM status code
a < +0	QNAN	oneapi::mkl::vm::status::errdom
+0	+0	
-0	-0	
-∞	QNAN	oneapi::mkl::vm::status::errdom
+∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
: The buffer a containing input vector of size n.

mode
: Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is oneapi::mkl::vm::mode::not_defined.

errhandler
: Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

powr

Computes `a` to the power `b` for elements of two vectors, where the elements of vector argument `a` are all non-negative.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event powr(
        sycl::queue& exec_queue,
        std::int64_t n,
```

(continues on next page)

(continued from previous page)

```
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event powr(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        const T *b,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`powr` supports the following precisions.

T
`float`
`double`

Description

The powr(a, b) function raises each element of vector `a` by the corresponding element of vector `b`. The elements of `a` are all nonnegative ($a_i \geq 0$).

Data Type	Threshold Limitations on Input Parameters
single precision	$a_i < (FLT\_MAX)^{1/`b`}i`$
double precision	$a_i < (DBL\_MAX)^{1/`b`}i`$

Special values and VM status code treatment for v?Powr function are the same as for pow, unless otherwise indicated in this table:

Argument 1	Argument 2	Result	Status code
a < 0	any value b	NAN	`oneapi::mkl::vm::status::errdom`
0 < a < ∞	±0	1	
±0	-∞ < b < 0	+∞	
±0	-∞	+∞	
±0	b > 0	+0	
1	-∞ < b < ∞	1	
±0	±0	NAN	
+∞	±0	NAN	
1	+∞	NAN	
a≥ 0	NAN	NAN	
NAN	any value b	NAN	
0 < a <1	-∞	+∞	
a > 1	-∞	+0	
0 ≤a < 1	+∞	+0	
a > 1	+∞	+∞	
+∞	b < +0	+0	
+∞	b > +0	+∞	
QNAN	QNAN	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	SNAN	QNAN	`oneapi::mkl::vm::status::errdom`
SNAN	QNAN	QNAN	`oneapi::mkl::vm::status::errdom`
SNAN	SNAN	QNAN	`oneapi::mkl::vm::status::errdom`

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing 1st input vector of size `n`.

b
The buffer `b` containing 2nd input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the 1st input vector of size `n`.

b
Pointer `b` to the 2nd input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

powx

Computes vector `a` to the scalar power `b`.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event powx(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        T b,
        sycl::buffer<T,1>& y,
```

(continues on next page)

(continued from previous page)

```
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event powx(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T b,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

powx supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

Description

The powx function computes a to the power b for a vector a and a scalar b.

The real function powx has certain limitations on the input range of a and b parameters. Specifically, if a[i] is positive, then b may be arbitrary. For negative a[i], the value of b must be an integer (either positive or negative).

The complex function powx has no input range limitations.

Special values and VM status code treatment are the same as for the pow function.

Input Parameters

Buffer API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
: The buffer a containing 1st input vector of size n.

b
Fixed value of power `b`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the 1st input vector of size `n`.

b
Fixed value of power `b`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

remainder

Performs element by element computation of the remainder function on the elements of vector `a` and the corresponding elements of vector `b`.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event remainder(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event remainder(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        const T *b,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`remainder` supports the following precisions.

T
float
double

Description

The `remainder(a)` function computes the remainder of each element of vector `a`, with respect to the corresponding elements of vector `b`: compute the values of `n` such that

```
n = ai - n*bi
```

where `n` is the integer nearest to the exact value of a_i/b_i. If two integers are equally close to a_i/b_i, `n` is the even one. If `n` is zero, it has the same sign as a_i.

Argument 1	Argument 2	Result	VM status code
a not NAN	± 0	NAN	oneapi::mkl::vm::status::errdom
$\pm\infty$	b not NAN	NAN	
± 0	b$\neq 0$, not NAN	± 0	
a finite	$\pm\infty$	a	
NAN	b	NAN	
a	NAN	NAN	

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing 1st input vector of size `n`.

b
The buffer `b` containing 2nd input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the 1st input vector of size `n`.

b
Pointer `b` to the 2nd input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

rint

Computes a rounded integer value in the current rounding mode.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event rint(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event rint(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`rint` supports the following precisions.

T
`float`
`double`

Description

The rint(a) function computes a rounded floating-point integer value using the current rounding mode for each vector element.

The rounding mode affects the results computed for inputs that fall between consecutive integers. For example:

- **`f(0.5) = 0`, for rounding modes set to round to nearest round** toward zero or to minus infinity.
- `f(0.5) = 1`, for rounding modes set to plus infinity.
- **`f(-1.5) = -2`, for rounding modes set to round to nearest or to** minus infinity.
- **`f(-1.5) = -1`, for rounding modes set to round toward zero or to** plus infinity.

Argument	Result	Status code
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
QNAN	QNAN	
SNAN	QNAN	

The rint function does not generate any errors.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

round

Computes a value rounded to the nearest integer for each vector element.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event round(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event round(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

round supports the following precisions.

T
float
double

Description

The round(a) function computes a value rounded to the nearest integer for each vector element. Input elements that are halfway between two consecutive integers are always rounded away from zero regardless of the rounding mode.

Argument	Result	Status code
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
QNAN	QNAN	
SNAN	QNAN	

The round(a) function does not generate any errors.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

sin

Computes sine of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event sin(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event sin(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});
```

(continues on next page)

(continued from previous page)

```
} // namespace oneapi::mkl::vm
```

`sin` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Description

The sin(a)function computes sine of vector elements.

Note that arguments `abs(a[i])` $\leq$ `213` and `abs(a[i])` $\leq$ `216` for single and double precisions respectively are called fast computational path. These are trigonometric function arguments for which VM provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VM High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VM Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Argument	Result	VM status code
+0	+0	
-0	-0	
+∞	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

`Sin(z) = -i*Sinh(i*z)`.

Input Parameters

Buffer API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
: The buffer `a` containing input vector of size `n`.

mode
: Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

sincos

Computes sine and cosine of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event sincos(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        sycl::buffer<T,1>& z,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event sincos(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        T* z,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`sincos` supports the following precisions.

T
float
double

Description

The sincos(a) function computes sine and cosine of vector elements.

Note that arguments `abs(a[i])` $\leq$ `213`and `abs(a[i])` $\leq$ `216`for single and double precisions respectively are called fast computational path. These are trigonometric function arguments for which VM provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VM High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VM Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Argument	Result 1	Result 2	Status code
+0	+0	+1	
-0	-0	+1	
+∞	QNAN	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	QNAN	
SNAN	QNAN	QNAN	

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output sine vector of size `n`.

z
The buffer `z` containing the output cosine vector of size `n`.

USM API:

y
Pointer `y` to the output sine vector of size `n`.

z
The buffer `z` containing the output cosine vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

sind

Computes the sine of vector elements multiplied by $\pi/180$.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event sind(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event sind(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
```

(continues on next page)

(continued from previous page)

```
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`sind` supports the following precisions.

T
`float`
`double`

Description

The sind(a) function is a degree argument trigonometric function. It computes the sine of vector elements multiplied by $\pi/180$. For an argument `a`, the function computes sin(π*a/180).

Note that arguments abs(a_i) $\leq 2^{24}$ for single precision or abs(a_i) $\leq 2^{52}$ for double precision, they belong to the *fast computational path*: trigonometric function arguments for which VM provides the best possible performance. Avoid arguments with do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

Argument	Result	Status code
+0	+0	
-0	-0	
+∞	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
: The buffer `a` containing input vector of size `n`.

mode
: Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
: Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

sinh

Computes hyperbolic sine of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event sinh(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event sinh(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`sinh` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

Description

The sinh(a) function computes hyperbolic sine of vector elements.

Data Type	Threshold Limitations on Input Parameters
single precision	-Log(FLT_MAX)-Log(2) <a[i] < Log(FLT_MAX)+Log(2)
double precision	-Log(DBL_MAX)-Log(2) <a[i] < Log(DBL_MAX)+Log(2)

Argument	Result	Status code
+0	+0	
-0	-0	
a > overflow	+∞	`oneapi::mkl::vm::status::overflow`
a < -overflow	-∞	`oneapi::mkl::vm::status::overflow`
+∞	+∞	
-∞	-∞	
QNAN	QNAN	
SNAN	QNAN	

+i·∞	-∞+i·QNAN	QNAN+i·QNA	-0+i·QNAN	+0+i·QNA	QNAN+i·QNA	+∞+i·QNA	QNAN+i·QNAN
+i·Y	-∞·Cos(Y)+ i·∞·Sin(Y)					+∞·CIS(Y)	QNAN+i·QNAN
+i·0	-∞+i·0		-0+i·0	+0+i·0		+∞+i·0	QNAN+i·0
-i·0	-∞-i·0		-0-i·0	+0-i·0		+∞-i·0	QNAN-i·0
-i·Y	-∞·Cos(Y)+ i·∞·Sin(Y)					+∞·CIS(Y)	QNAN+i·QNAN
-i·∞	-∞+i·QNAN	QNAN+i·QNA	-0+i·QNAN	+0+i·QNA	QNAN+i·QNA	+∞+i·QNA	QNAN+i·QNAN
+i·NAN	-∞+i·QNAN	QNAN+i·QNA	-0+i·QNAN	+0+i·QNA	QNAN+i·QNA	+∞+i·QNA	QNAN+i·QNAN

Notes:

- **The complex sinh(a) function sets the VM status code to**
  oneapi::mkl::vm::status::overflow in the case of overflow, that is, when RE(a), IM(a) are finite non-zero numbers, but the real or imaginary part of the exact result is so large that it does not meet the target precision.
- `sinh(CONJ(a))=CONJ(sinh(a))`
- `sinh(-a)=-sinh(a)`.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

sinpi

Computes the sine of vector elements multiplied by π.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event sinpi(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event sinpi(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`sinpi` supports the following precisions.

T
`float`
`double`

Description

The sinpi(a) function computes the sine of vector elements multiplied by π. For an argument `a`, the function computes sin(π*a).

Argument	Result	Status code
+0	+0	
-0	-0	
+n, positive integer	+0	
-n, negative integer	-0	
$+\infty$	QNAN	`oneapi::mkl::vm::status::errdom`
$-\infty$	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

If arguments abs(a_i) $\leq 2^{22}$ for single precision or abs(a_i) $\leq 2^{51}$ for double precision, they belong to the *fast computational path*: arguments for which VM provides the best possible performance. Avoid arguments which do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

sqr

Performs element by element squaring of the vector.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event sqr(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event sqr(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`sqr` supports the following precisions.

T
`float`
`double`

Description

The sqr() function performs element by element squaring of the vector.

Argument	Result	Status code
+0	+0	
-0	+0	
+∞	+∞	
-∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

The `sqr` function does not generate any errors.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing the input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

sqrt

Computes a square root of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event sqrt(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event sqrt(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});
```

(continues on next page)

(continued from previous page)

```
} // namespace oneapi::mkl::vm
```

`sqrt` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Description

The sqrt function computes a square root of vector elements.

Argument	Result	VM status code
a< +0	QNAN	`oneapi::mkl::vm::status::errdom`
+0	+0	
-0	-0	
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
+∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

+i·∞	+∞+i·∞	+∞+i·∞	+∞+i·∞	+∞+i·∞	+∞+i·∞	+∞+i·∞	+∞+i·∞
+i·Y	+0+i·∞					+∞+i·0	
+i·0	+0+i·∞		+0+i·0	+0+i·0		+∞+i·0	
-i·0	+0-i·∞		+0-i·0	+0-i·0		+∞-i·0	
-i·Y	+0-i·∞					+∞-i·0	
-i·∞	+∞-i·∞	+∞-i·∞	+∞-i·∞	+∞-i·∞	+∞-i·∞	+∞-i·∞	+∞-i·∞
+i·NAN							

Notes:

- `Sqrt(CONJ(z))=CONJ(Sqrt(z))`.

Input Parameters

Buffer API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
: The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the 1st input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

sub

Performs element by element subtraction of vector b from vector a.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event sub(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
.. code-block:: cpp
```

namespace oneapi::mkl::vm {

sycl::event sub(
sycl::queue& exec_queue, std::int64_t n, const T *a, const T *b, T* y, std::vector<sycl::event> const & depends = {}, oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined, oneapi::mkl::vm::error_handler<T> errhandler - {});

} // namespace oneapi::mkl::vm

sub supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

Description

The sub(a, b) function performs element by element subtraction of vector a and vector b.

Argument 1	Argument 2	Result	Status code
+0	+0	+0	
+0	-0	+0	
-0	+0	+0	
-0	-0	-0	
+∞	+∞	QNAN	
+∞	-∞	+∞	
-∞	+∞	-∞	
-∞	-∞	QNAN	
SNAN	any value	QNAN	
any value	SNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

```
sub(x1+i*y1, x2+i*y2) = (x1-x2) + i*(y1-y2)
```

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when all RE(x), RE(y), IM(x), IM(y) arguments are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns ∞ in that part of the result, and sets the VM status code to `oneapi::mkl::vm::status::overflow` (overriding any possible `oneapi::mkl::vm::status::accuracy_warning` status).

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing 1st input vector of size `n`.

b
The buffer `b` containing 2nd input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the 1st input vector of size `n`.

b
Pointer `b` to the 2nd input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

tan

Computes tangent of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event tan(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event tan(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`tan` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Description

The tan(a) function computes tangent of vector elements.

Note that arguments `abs(a[i])` $\leq$ `213`and `abs(a[i])` $\leq$ `216`for single and double precisions respectively are called fast computational path. These are trigonometric function arguments for which VM provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VM High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VM Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Argument	Result	Status code
+0	+0	
-0	-0	
+∞	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

`Tan(z) = -i*Tanh(i*z)`.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

tand

Computes the tangent of vector elements multiplied by $\pi/180$.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event tand(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event tand(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`tand` supports the following precisions.

T
float
double

Description

The tand(a) function computes the tangent of vector elements multiplied by $\pi/180$. For an argument `x`, the function computes tan(π*`x`/180).

Note that arguments abs(a_i) $\leq 2^{38}$ for single precision or abs(a_i) $\leq 2^{67}$ for double precision, they belong to the *fast computational path*: trigonometric function arguments for which VM provides the best possible performance. Avoid arguments with do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

Argument	Result	Status code
+0	+1	
-0	+1	
$\pm\infty$	QNAN	`oneapi::mkl::vm::status::errdom`
$\pm\infty$	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

tanh

Computes hyperbolic tangent of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event tanh(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event tanh(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
```

(continues on next page)

(continued from previous page)

```
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`tanh` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Description

The tanh(a) function computes hyperbolic tangent of vector elements.

Argument	Result	Erro Code
+0	+0	
-0	-0	
+∞	+1	
-∞	-1	
QNAN	QNAN	
SNAN	QNAN	

+i·∞	-1+i·0	QNAN+i·QNA	QNAN+i·QNA	QNAN+i·QNA	QNAN+i·QNA	+1+i·0	QNAN+i·QNAN
+i·Y	-1+i·0·Tan(Y					+1+i·0·Tan(Y	QNAN+i·QNAN
+i·0	-1+i·0		-0+i·0	+0+i·0		+1+i·0	QNAN+i·0
-i·0	-1-i·0		-0-i·0	+0-i·0		+1-i·0	QNAN-i·0
-i·Y	-1+i·0·Tan(Y					+1+i·0·Tan(Y	QNAN+i·QNAN
-i·∞	-1-i·0	QNAN+i·QNA	QNAN+i·QNA	QNAN+i·QNA	QNAN+i·QNA	+1-i·0	QNAN+i·QNAN
+i·NAN	-1+i·0	QNAN+i·QNA	QNAN+i·QNA	QNAN+i·QNA	QNAN+i·QNA	+1+i·0	QNAN+i·QNAN

Notes:

- `tanh(CONJ(a))=CONJ(tanh(a))`
- `tanh(-a)=-tanh(a)`.

The tanh(a) function does not generate any errors.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

tanpi

Computes the tangent of vector elements multiplied by π.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event tanpi(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event tanpi(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`tanpi` supports the following precisions.

T
float
double

Description

The tanpi(a) function computes the tangent of vector elements multiplied by π. For an argument a, the function computes tan(π*a).

Argument	Result	Status code
+0	+0	
-0	+0	
n, even integer	*copysign(0.0, n)	
n, odd integer	*copysign(0.0, -n)	
n + 0.5, for n even integer and n + 0.5 representable	+∞	
n + 0.5, for n odd integer and n + 0.5 representable	-∞	
+∞	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

The copysign(x, y) function returns the first vector argument x with the sign changed to match that of the second argument y.

If arguments abs(a_i) $\leq 2^{13}$ for single precision or abs(a_i) $\leq 2^{67}$ for double precision, they belong to the *fast computational path*: arguments for which VM provides the best possible performance. Avoid arguments with do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer a containing input vector of size n.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

tgamma

Computes the gamma function of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event tgamma(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event tgamma(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`tgamma` supports the following precisions.

T
`float`
`double`

Description

The tgamma(a) function computes the gamma function for elements of the input vector `a` and writes them to the output vector `y`. Precision overflow thresholds for the tgamma function are beyond the scope of this document. If the result does not meet the target precision, the function raises sets the VM status code to oneapi::mkl::vm::status::sing.

Argument	Result	Status code
+0	+∞	`oneapi::mkl::vm::status::sing`
-0	-∞	`oneapi::mkl::vm::status::sing`
negative integer	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
+∞	+∞	
a > overflow	+∞	`oneapi::mkl::vm::status::sing`
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue
: The queue where the routine should be executed.

n
: Specifies the number of elements to be calculated.

a
: The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler
Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

trunc

Computes an integer value rounded towards zero for each vector element.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event trunc(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event trunc(
        sycl::queue& exec_queue,
        std::int64_t n,
        const T *a,
        T* y,
        std::vector<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`trunc` supports the following precisions.

T
float
double

Description

The trunc(a) function computes an integer value rounded towards zero for each vector element.

$$y_i = \begin{cases} \lfloor a_i \rfloor, a_i \geq 0 \\ \lceil a_i \rceil, a_i < 0 \end{cases}$$

Argument	Result	Status code
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
QNAN	QNAN	
SNAN	QNAN	

The trunc function does not generate any errors.

Input Parameters

Buffer API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
The buffer `a` containing input vector of size `n`.

mode
Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue
The queue where the routine should be executed.

n
Specifies the number of elements to be calculated.

a
Pointer `a` to the input vector of size `n`.

depends
Vector of dependent events (to wait for input data to be ready).

mode
Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y
The buffer `y` containing the output vector of size `n`.

USM API:

y
Pointer `y` to the output vector of size `n`.

return value (event)
Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

VM Service Functions

The VM Service functions enable you to set/get the accuracy mode and error code. These functions are available both in the Fortran and C interfaces. The table below lists available VM Service functions and their short description.

Function Short Name	Description
set_mode	Sets the VM mode for given queue
get_mode	Gets the VM mode for given queue
set_status	Sets the VM status code for given queue
get_status	Gets the VM status code for given queue
clear_status	Clears the VM status code for given queue
create_error_handler	Creates the local VM error handler for a function

Parent topic: *Vector Math*

set_mode

Sets a new mode for VM functions according to the `mode` parameter and returns the previous VM mode.

Syntax

```
namespace oneapi::mkl::vm {

    oneapi::mkl::vm::mode set_mode(
        sycl::queue& exec_queue,
        oneapi::mkl::vm::mode new_mode);

} // namespace oneapi::mkl::vm
```

Description

The set_mode function sets a new mode for VM functions according to the `new_mode` parameter and returns the previous VM mode. The mode change has a global effect on all the VM functions within a queue.

The `mode` parameter is designed to control accuracy for a given queue.

Value of mode	Description
Accuracy Control	
`oneapi::mkl::vm::mode::ha`	High accuracy versions of VM functions.
`oneapi::mkl::vm::mode::la`	Low accuracy versions of VM functions.
`oneapi::mkl::vm::mode::ep`	Enhanced performance accuracy versions of VM functions.
`oneapi::mkl::vm::mode::not_defined`	VM mode not defined. This has no effect.

The assumed value of the `mode` parameter for a new queue, if `set_mode` is not called is `oneapi::mkl::vm::mode::ha`.

Input Parameters

exec_queue
The queue where the routine should be executed.

new_mode
Specifies the VM mode to be set.

Output Parameters

return value (old_mode)
Specifies the former VM mode.

Parent topic: *VM Service Functions*

get_mode

Gets the VM mode.

Syntax

```
namespace oneapi::mkl::vm {

    oneapi::mkl::vm::mode get_mode(
        sycl::queue& exec_queue);

} // namespace oneapi::mkl::vm
```

Description

The function get_mode function returns the global VM `mode` parameter that controls accuracy for a given queue.

Value of mode	Description
Accuracy Control	
`oneapi::mkl::vm::mode::`	High accuracy versions of VM functions.
`oneapi::mkl::vm::mode::`	Low accuracy versions of VM functions.
`oneapi::mkl::vm::mode::`	Enhanced performance accuracy versions of VM functions.
`oneapi::mkl::vm::mode::`	VM mode not defined. It means that no special provisions for accuracy have been made for this queue. See *set_mode* for details.

Input Parameters

exec_queue
The queue where the routine should be executed.

Output Parameters

return value
The current global VM mode for the queue `exec_queue`.

Parent topic: *VM Service Functions*

set_status

Sets the global VM status according to new value and returns the previous VM status.

Syntax

```
namespace oneapi::mkl::vm {

    oneapi::mkl::vm::status set_status(
        sycl::queue& exec_queue,
        oneapi::mkl::vm::status new_status);

} // namespace oneapi::mkl::vm
```

Description

The set_status function sets the global VM status to new value and returns the previous VM status code for a given queue.

The global VM status is a single value and it registers the bitwise-OR of status codes that happened inside VM functions run on the specific queue. For performance reasons, it might be done in non-atomic manner. The possible status codes are listed in the table below.

Status	Description
Successful Execution	
`oneapi::mkl::vm::status::succ`	VM function execution completed successfully
`oneapi::mkl::vm::status::not_`	VM status not defined
Warnings	
`oneapi::mkl::vm::status::accu`	VM function execution completed successfully in a different accuracy mode
Computational status codes	
`oneapi::mkl::vm::status::errd`	Values are out of a range of definition producing invalid (QNaN) result
`oneapi::mkl::vm::status::sing`	Values cause divide-by-zero (singularity) computational errors and produce and invalid (QNaN or Inf) result
`oneapi::mkl::vm::status::over`	An overflow happened during the calculation process
`oneapi::mkl::vm::status::unde`	An underflow happened during the calculation process

Input Parameters

exec_queue
The queue where the routine should be executed.

new_status
Specifies the VM status to be set.

Output Parameters

return value (old_status)
Specifies the former VM status.

Parent topic: *VM Service Functions*

get_status

Gets the VM status.

Syntax

```
namespace oneapi::mkl::vm {

    oneapi::mkl::vm::status get_status(
        sycl::queue& exec_queue);

} // namespace oneapi::mkl::vm
```

Description

The get_status function gets the VM status for a given queue.

The global VM status is a single value and it registers the bitwise-OR of status codes that happened inside VM functions run on the specific queue. For performance reasons, it might be done in non-atomic manner. The possible status codes are listed in the table below.

Status	Description
Successful Execution	
`oneapi::mkl::vm::status::succ`	VM function execution completed successfully
`oneapi::mkl::vm::status::not_`	VM status not defined
Warnings	
`oneapi::mkl::vm::status::accu`	VM function execution completed successfully in a different accuracy mode
Computational status codes	
`oneapi::mkl::vm::status::errd`	Values are out of a range of definition producing invalid (QNaN) result
`oneapi::mkl::vm::status::sing`	Values cause divide-by-zero (singularity) computational errors and produce and invalid (QNaN or Inf) result
`oneapi::mkl::vm::status::over`	An overflow happened during the calculation process
`oneapi::mkl::vm::status::unde`	An underflow happened during the calculation process

Input Parameters

exec_queue
The queue where the routine should be executed.

Output Parameters

return value (status)
Specifies the VM status.

Parent topic: *VM Service Functions*

clear_status

Resets the global VM status to `oneapi::mkl::vm::status::success` and returns the previous VM status code.

Syntax

```
namespace oneapi::mkl::vm {

    oneapi::mkl::vm::status clear_status(
        sycl::queue& exec_queue);

} // namespace oneapi::mkl::vm
```

Description

The clear_status function sets the VM status code to `oneapi::mkl::vm::status::success` and returns the previous VM status code for a given queue.

The global VM status is a single value and it registers the bitwise-OR of status codes that happened inside VM functions run on the specific queue. For performance reasons, it might be done in non-atomic manner. The possible status codes are listed in the table below.

Status code	Description
Successful Execution	
`oneapi::mkl::vm::status::succ`	VM function execution completed successfully
`oneapi::mkl::vm::status::not_`	VM status not defined
Warnings	
`oneapi::mkl::vm::status::accu`	VM function execution completed successfully in a different accuracy mode
Computational status codes	
`oneapi::mkl::vm::status::errd`	Values are out of a range of definition producing invalid (QNaN) result
`oneapi::mkl::vm::status::sing`	Values cause divide-by-zero (singularity) computational errors and produce and invalid (QNaN or Inf) result
`oneapi::mkl::vm::status::over`	An overflow happened during the calculation process
`oneapi::mkl::vm::status::unde`	An underflow happened during the calculation process

Input Parameters

exec_queue
The queue where the routine should be executed.

Output Parameters

return value (old_status)
Specifies the VM status code before the call.

Parent topic: *VM Service Functions*

create_error_handler

Creates an error handler for VM functions that support computational error handling.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

oneapi::mkl::vm::error_handler<T> create_error_handler(
    sycl::buffer<oneapi::mkl::vm::status, 1> & status_array,
    int64_t length = 1,
    oneapi::mkl::vm::status status = oneapi::mkl::vm::status::not_defined,
    T fixup = {},
    bool copysign = false);
```

(continues on next page)

(continued from previous page)

```
} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

oneapi::mkl::vm::error_handler<T> create_error_handler(
    oneapi::mkl::vm::status* status_array,
    int64_t length = 1,
    oneapi::mkl::vm::status status = oneapi::mkl::vm::status::not_defined,
    T fixup = {},
    bool copysign = false);

} // namespace oneapi::mkl::vm
```

`create_error_handler` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

Description

create_error_handler creates an computational error handler to be passed to VM functions that support computational error handling.

A VM computational error handler supports three modes:

- **Single status mode**: all computational errors that happened during the execution of a function are being written into one single status variable.

  After the execution, the single value is either un-changed if no errors happened or contains bitwise-OR of initial value and non-success status codes occurred during computation.

  To enable this mode, `status_array` must point to any `status`-type array or buffer of 1 or more elements and `length` must be 1.

- **Multiple status mode**: each non-successful status code is saved in `status_array` at the same index as the argument causing the non-success status code.

  Success status codes are not written to `status_array`. This means the array needs to be allocated and initialized before function execution.

  To enable this mode, `status_array` must have at least the same length as the argument and result vectors, and `length` must be set to this length.

- **Fixup mode**: for all arguments that caused a specific error status, results are overwritten by a user-defined value.

  To enable this mode, the target `status` and `fixup` values must be set. The fixup value is written to results for each argument for which calculation resulted in the `status` status code.

  To fix multiple error status codes, `status` can be provided with bitwise-OR of status codes.

If `copysign` is set to true then the sign of `fixup` is set to the same sign as the argument that caused the status code – a suitable option for symmetric math functions.

The following table lists the possible computational status code values.

Status code	Description
Successful Execution	
`oneapi::mkl::vm::status::succ`	VM function execution completed successfully
`oneapi::mkl::vm::status::not_`	VM status not defined
Warnings	
`oneapi::mkl::vm::status::accu`	VM function execution completed successfully in a different accuracy mode
Computational Errors	
`oneapi::mkl::vm::status::errd`	Values are out of a range of definition producing invalid (QNaN) result
`oneapi::mkl::vm::status::sing`	Values cause divide-by-zero (singularity) computational errors and produce and invalid (QNaN or Inf) result
`oneapi::mkl::vm::status::over`	An overflow happened during the calculation process
`oneapi::mkl::vm::status::unde`	An underflow happened during the calculation process

Notes:

- `status_array` must be allocated and initialized before calling VM functions in multiple status error handling mode.

  The array should be large enough to contain `n` status codes, where `n` is the same as the input/output vector size for the VM function.

- If no arguments are passed to `create_error_handler`, then an empty object is created with all three error handling modes disabled.

  In this case, the VM math functions set the global status code only.

Input Parameters

status_array
Array to store status codes (should be a buffer for buffer API).

length
Length of the errarray. This is an optional argument, default value is `1`.

status_code
Status code to match and fix the results. This is an optional argument, default value is `oneapi::mkl::vm::status::not_defined`.

fixup
Fixup value for results. This is an optional argument, default value is `0.0`.

copysign
Flag for setting the fixup value's sign the same as the argument's. This is an optional argument, default value `false`.

Output Parameters

return value
Specifies the error handler object to be created.

Parent topic: *VM Service Functions*

Exceptions

All VM mathematical functions throw exceptions in exceptional cases. The following table summarizes the conditions.

exception	when thrown

oneapi::mkl::invalid_argument	**buffer API:**
	n < 0;
	y.get_count() < n;
	z.get_count() < n; // for sincos

	USM API: n < 0; any pointer argument is nullptr
oneapi::mkl::host_bad_alloc	**USM API:** when internal copying to and from host memory is used and corresponding allocation fails
oneapi::mkl::device_bad_alloc	**USM API:** when internal copying to and from device memory is used and corresponding allocation fails

Bibliography

For more information about the VM functionality, refer to the following publications:

- **VM**

[IEEE754]
IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std 754-2008.

Parent topic: *Vector Math*

9.3 oneMKL Appendix

9.3.1 Future considerations

The following items are being considered for future versions of this specification:

- Encapsulation of matrix and vector information in classes. Matrix storage information could also be encapsulated.
- More human-readable names for linear algebra functionality, aligned with the P1673 C++ proposal.
- Broader support for row major layout.
- Alternative handling of computational failures.

9.3.2 Acknowledgment

The oneMKL Technical Advisory Board members provided valuable feedback to the specification and are thanked for their contributions.

CHAPTER

TEN

LEGAL NOTICES AND DISCLAIMERS

BIBLIOGRAPHY

[OpenCLSpec] Khronos OpenCL Working Group, The OpenCL Specification Version:2.1 Document Revision:24 Available from opencl-2.1.pdf

[SYCLSpec] Khronos®OpenCL™ Working Group — SYCL™ subgroup, SYCL™ Specification SYCL™ integrates OpenCL™ devices with modern C++, Version 1.2.1 Available from sycl-1.2.1.pdf

[Lloyd82] Stuart P Lloyd. *Least squares quantization in PCM*. IEEE Transactions on Information Theory 1982, 28 (2): 1982pp: 129–137.

[Bro07] Bro, R.; Acar, E.; Kolda, T. *Resolving the sign ambiguity in the singular value decomposition*. SANDIA Report, SAND2007-6422, Unlimited Release, October, 2007.

[Bentley80] J. L. Bentley. Multidimensional Divide and Conquer. Communications of the ACM, 23(4):214–229, 1980.

[Friedman17] J. Friedman, T. Hastie, R. Tibshirani. *The Elements of Statistical Learning Data Mining, Inference, and Prediction.* Springer, 2017.

[Zhang04] T. Zhang. Solving Large Scale Linear Prediction Problems Using Stochastic Gradient Descent Algorithms. ICML 2004: Proceedings Of The Twenty-First International Conference On Machine Learning, 919–926, 2004.

[Lang87] S. Lang. *Linear Algebra*. Springer-Verlag New York, 1987.

[Ping14] Ping Tak Peter and Eric Polizzi. *FEAST as a Subspace Iteration Eigensolver Accelerated by Approximate Spectral Projection.* 2014.

[Demmel90] J. W. Demmel and W. Kahan. *Accurate singular values of bidiagonal matrices*. SIAM J. Sci. Stat. Comput., 11 (1990), pp. 873-912.

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X