array_ref<>

Bryce Adelstein Lelbach, Lawrence Berkeley National Laboratory

H. Carter Edwards, Sandia National Laboratory

The case for array_ref<>

The case for array_ref<>

- Multi-dimensional arrays are foundational data structure for many domains.
  - Graphics, gaming, big data, science, engineering.

The case for array_ref<>

- Multi-dimensional arrays are implemented by many existing libraries.
  - Blitz++, Eigen, Boost.MultiArray, NT2, Kokkos.

The case for array_ref<>

```
FArrayBox
FArrayBox
array3d<>
vector4d<>
grid3d <>
grid
DataArray4D<>
DenseTile<>
nt2::table<>
Eigen::Matrix<>
Kokkos::View<>
```

The case for array_ref<>

- Standardization would facilitate:
  - Interoperability between libraries and with other languages.
  - Portability and code reuse.

The case for array_ref<>

- Modern hardware provides multiple types of memory and multiple mechanisms for accessing memory.
    - We need a hardware-agnostic abstraction which can utilize these features via hardware-specific policies.

C-Style Arrays

- `T a[N];`

C++-Style Arrays

- `array<T, N> a;`

C-Style Arrays

```
• T a[N];
• T* a = new T[N];
```

C++-Style Arrays

```
• array<T, N> a;
• vector<T> a(N);
```

C-Style Arrays

- T a[N];
- T* a = new T[N];
- T a[N][M];

C++-Style Arrays

- array<T, N> a;
- vector<T> a(N);
- ???

C-Style Arrays

- `T a[N];`
- `T* a = new T[N];`
- `T a[N][M];`

C++-Style Arrays

- `array<T, N> a;`
- `vector<T> a(N);`
- `array<T, N, M> a;`

C-Style Arrays

- T a[N];
- T* a = new T[N];
- T a[N][M];

C++-Style Arrays

- array<T, N> a;
- vector<T> a(N);
- ~~array<T, N, M> a;~~

C-Style Arrays

- `T a[N];`
- `T* a = new T[N];`
- `T a[N][M];`

C++-Style Arrays

- `array<T, N> a;`
- `vector<T> a(N);`
- `array<array<T, M>, N> a;`

C-Style Arrays

- T a[N];
- T* a = new T[N];
- T a[N][M];
- T** a = new T[N][M];

C++-Style Arrays

- array<T, N> a;
- vector<T> a(N);
- array<array<T, M>, N> a;
- ???

C-Style Arrays

- T a[N];
- T* a = new T[N];
- T a[N][M];
- T** a = new T[N][M];

C++-Style Arrays

- array<T, N> a;
- vector<T> a(N);
- array<array<T, M>, N> a;
- vector<T> a(N, M);

C-Style Arrays

- `T a[N];`
- `T* a = new T[N];`
- `T a[N][M];`
- ~~`T** a = new T[N][M];`~~

C++-Style Arrays

- `array<T, N> a;`
- `vector<T> a(N);`
- `array<array<T, M>, N> a;`
- ~~`vector<T> a(N, M);`~~

C-Style Arrays

- `T a[N];`
- `T* a = new T[N];`
- `T a[N][M];`
- ```
  T** a = new T*[N];
  for (int i; i < N; ++i)
    a[i] = new T[M];
  ```

C++-Style Arrays

- `array<T, N> a;`
- `vector<T> a(N);`
- `array<array<T, M>, N> a;`
- ???
  ```

C-Style Arrays

- `T a[N];`
- `T* a = new T[N];`
- `T a[N][M];`
- ```
  T** a = new T*[N];
  for (int i; i < N; ++i)
    a[i] = new T[M];
  ```

C++-Style Arrays

- `array<T, N> a;`
- `vector<T> a(N);`
- `array<array<T, M>, N> a;`
- ```
  vector<vector<T> > a
     (N, vector<T>(M));
  ```

C-Style Array Declaration

- `T a[N];`
- `T* a = new T[N];`

C-Style Array Element Access

- `a[i]      // *(a+i)`
- `a[i]      // *(a+i)`

C-Style Array Declaration

- `T a[N];`
- `T* a = new T[N];`
- `T a[N][M];`

C-Style Array Element Access

- `a[i]      // *(a+i)`
- `a[i]      // *(a+i)`
- `a[i][j]   // *(a+i*M+j)`

C-Style Array Declaration

- `T a[N];`
- `T* a = new T[N];`
- `T a[N][M];`
- `T** a = new T*[N];`

```
for (int i; i < N; ++i)
  a[i] = new T[M];
```

C-Style Array Element Access

- `a[i]     // *(a+i)`
- `a[i]     // *(a+i)`
- `a[i][j] // *(a+i*M+j)`
- `a[i][j] // 2 indirections`

C++-Style Array Declaration

```
• array<T, N> a;
• vector<T> a(N);
• array<array<T, M>, N> a;
• vector<vector<T> > a
     (N, vector<T>(M));
```

C++-Style Array Element Access

```
• a[i]      // *(a.data()+i)
• a[i]      // *(a.data()+i)
• a[i][j]   // *(a.data()+i*M+j)
• a[i][j]   // 2 indirections
```

C-Style Arrays

- `T a[N];`
- `T* a = new T[N];`
- `T a[N][M];`
- `T** a = new T*[N];`

```
for (int i; i < N; ++i)
  a[i] = new T[M];
```

C++-Style Arrays

- `array<T, N> a;`
- `vector<T> a(N);`
- `array<array<T, M>, N> a;`
- `vector<vector<T> > a`

```
  (N, vector<T>(M));
```

C-Style Arrays

```
• T a[N];
• T* a = new T[N];
• T a[N][M];
• T** a = new T*[N];
  for (int i; i < N; ++i)
     a[i] = new T[M];
```

C++-Style Arrays

```
• array<T, N> a;
• vector<T> a(N);
• array<array<T, M>, N> a;
• vector<vector<T> > a
     (N, vector<T>(M));
```

C-Style Arrays

```
T a[N];
T* a = new T[N];
T a[N][M];
T* a = new T[N * M];
```

C++-Style Arrays

```
array<T, N> a;
vector<T> a(N);
array<array<T, M>, N> a;
vector<T> a(N * M);
```

C-Style Array Declaration

- `T a[N];`
- `T* a = new T[N];`
- `T a[N][M];`
- `T* a = new T[N * M];`

C-Style Array Element Access

- `a[i]       // *(a+i)`
- `a[i]       // *(a+i)`
- `a[i][j]    // *(a+i*M+j)`
- `a[i*M+j]   // *(a+i*M+j)`

C++-Style Array Declaration

- `array<T, N> a;`
- `vector<T> a(N);`
- `array<array<T, M>, N> a;`
- `vector<T> a(N * M);`

C++-Style Array Element Access

- `a[i]        // *(a.data()+i)`
- `a[i]        // *(a.data()+i)`
- `a[i][j]     // *(a.data()+i*M+j)`
- `a[i*M+j]    // *(a.data()+i*M+j)`

C-Style Arrays

```
• T a[N];
• T* a = new T[N];
• T a[N][M];
• T* a = new T[N * M];
```

C++-Style Arrays

```
• array<T, N> a;
• vector<T> a(N);
• array<array<T, M>, N> a;
• vector<T> a(N * M);
```

C-Style Arrays

- T a[N];
- T* a = **new** T[N];
- T a[N][M];
- T* a = **new** T[N * M];

C++-Style Arrays

- array<T, N> a;
- vector<T> a(N);
- ~~array<array<T, M>, N> a;~~
- vector<T> a(N * M);

C-Style Arrays

```
T a[N];
T* a = new T[N];
T a[N][M];
T* a = new T[N * M];
```

C++-Style Arrays

```
array<T, N> a;
vector<T> a(N);
array<T, N * M> a;
vector<T> a(N * M);
```

Array Layout: A mapping from an index to linear storage.

Row-Major AKA Right

- C++, NumPy (default)
- Last dimension is contiguous

Index	Element
0	a_{11}
1	a_{12}
2	a_{21}
3	a_{22}

`A[i*M + j]`

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

Column-Major AKA Left

- Fortran, MATLAB
- First dimension is contiguous

Index	Element
0	a_{11}
1	a_{21}
2	a_{12}
3	a_{22}

`A[i + j*N]`

Row-Major AKA Right

- C++, NumPy (default)
- Last dimension is contiguous

Column-Major AKA

- Fortran, MATLAB
- First dimension is contiguous

$$x_r + D_r\big(x_{r-1} + D_{r-1}(\cdots + x_1)\big)$$

x := index

r := rank

D := dimensions

$$x_0 + D_0\big(x_1 + D_1(\cdots + x_r)\big)$$

```
A[i*M   + j]
A[i*M*L + j*L + k]
```

```
A[i + j*N]
A[i + j*N + k*N*M]
```

Element access should be independent of layout.

E.g.

```
a[i][j]
a(i, j)
```

not

```
a[i * M + j]
a[slice(j, N, M)][i]
```

E.g.

```
a[i][j]
a(i, j)
```

not

```
a[i * M + j]
a[slice(j, N, M)][i]
```

`valarray<>`

- Container for numeric types.
  - Slicing interface
  - "Optimized" numerical operations (**`operator+=`**, **`operator*=`**, etc)
- Problems:
  - Poor support for multidimensional arrays.
  - Numerical operations uses proxy objects.
    - Expression template approaches (Blitz++, Eigen, NT2) outperform `valarray<>`.
    - Proxy objects can be unwieldy to work with.
  - Original authors abandoned the proposal before it was added to the standard.

Design Goals

- Non-owning.
- Performant.
- Multi-dimensional (with static rank).
- Unified API for static and dynamic extents.
- User and vendor extensible through compile-time policies.
- Element access is independent of policies.
- Slicing and striding facilities.
- No numerical operations.

Design Goals

- Non-owning.
- Performant.
- Static rank.
- Unified API for static and dynamic extents.
- User and vendor extensible through policies.
- Element access is independent of policies.
- Slicing and striding facilities.
- No numerical operations.

```
namespace std {
namespace experimental {

template <class T, class... Properties>

class array_ref;

}}
```

```
void f(T A[N]);                         // C-Style API
void f(const T A[N]);                   // C-Style API

void f(array<T, N>& A);                 // C++-Style API
void f(array<T, N> const& A);           // C++-Style API

void f(array_ref<T[N]> A);              // Reference API
void f(array_ref<const T[N]> A);        // Reference API
```

```
void f(T* A, size_t N);           // C-Style API
void f(const T* A, size_t N);     // C-Style API

void f(vector<T>& A);              // C++-Style API
void f(vector<T> const& A);        // C++-Style API

void f(array_ref<T[]> A);          // Reference API
void f(array_ref<const T[]> A);   // Reference API
```

```
array_ref<T[N]> A;    // Static extents
array_ref<T[ ]> B;    // Dynamic extents
```

```
array_ref<T[N]> A;    // Static dimensions
array_ref<T[ ]> B;    // Dynamic dimensions

array_ref<T[N][M]> C; // Static dimensions
array_ref<T[ ][ ]> D; // Dynamic dimensions

array_ref<T[ ][M]> E; // 1 static/1 dynamic dimensions
array_ref<T[N][ ]> F; // 1 static/1 dynamic dimensions
```

Only the first dimension of an array type may have dynamic size.

```
array_ref<T[N]> A;     // Static extents
array_ref<T[ ]> B;     // Dynamic extents

array_ref<T[N][M]> C; // Static extents
array_ref<T[ ][ ]> D; // Dynamic extents

array_ref<T[ ][M]> E; // 1 static and 1 dynamic extent
array_ref<T[N][ ]> F; // 1 static and 1 dynamic extent
```

```
array_ref<T[N][M][L]> A; // Ok
array_ref<T[ ][ ][ ]> B; // Bad

array_ref<T[ ][M][L]> C; // Ok
array_ref<T[N][M][ ]> D; // Bad

array_ref<T[ ][ ][L]> E; // Bad
array_ref<T[ ][M][ ]> F; // Bad
array_ref<T[N][ ][ ]> G; // Bad
```

```
namespace std {
namespace experimental {
namespace array_property {

template <size_t... Dims>
class dimensions;

}}}
```

```
array_ref<T, dimensions<N, M> >           A(buf0);
array_ref<T, dimensions<dynamic_extent,
                        dynamic_extent> > B(buf1, N, M, L);
```

array_ref<T, dimensions<N, M> > A(buf0);

array_ref<T, dimensions<~~dynamic_extent~~,
~~dynamic_extent~~> > B(buf0, N, M, L);

```
array_ref<T, dimensions<N, M> >          A(buf0);
array_ref<T, dimensions<0, 0> >          B(buf0, N, M, L);
```

```
array_ref<T, dimensions<N,            A(buf0);
array_ref<T, dimensions<              B(buf0, N, M, L);
```

Array Type Specifier

- T[N]
- T[ ]
- T[N][M]
- T[ ][ ]
- T[ ][M]
- T[N][ ]

dimensions<> Specifier

- T, dimensions<N>
- T, dimensions<0>
- T, dimensions<N, M>
- T, dimensions<0, 0>
- T, dimensions<0, M>
- T, dimensions<N, 0>

Other Dimension Specifier Approaches

- array_ref<T[N][M][0]>
- array_ref<T[N][M][dynamic_extent]>
- array_ref<T, N, M, 0>
- array_ref<T, N, M, dynamic_extent>

```
template <size_t N>
void f()
{
    T buf0[N];
    array_ref<T[N]> A(buf0);

    array<T, N> buf1;
    array_ref<T[N]> B(buf1);
}
```

```
void f(size_t N)
{
    T buf0[N];
    array_ref<T[]> A(buf0, N);

    T* buf1 = new T[N];
    array_ref<T[]> B(buf1, N);

    vector<T> buf2(N);
    array_ref<T[]> C(buf2);
}
```

```
template <size_t N, size_t M>
void f()
{
    array<T, N * M> buf0;
    array_ref<T, dimensions<N, M> > A(buf0);
}

void g(size_t N, size_t M)
{
    vector<T> buf1(N * M);
    array_ref<T, dimensions<0, 0> > B(buf1, N, M);
}
```

```
template <size_t N, size_t L>
void f(size_t M)
{
    vector<T> buf(N * M * L);
    array_ref<T, dimensions<N, 0, L> > A(buf, M);
}
```

```
void f(size_t N, size_t M)
{
    vector<T> buf1(N * M);
    dimensions<0, 0> d(N, M);
    array_ref<T, dimensions<0, 0> > A(buf0, d);
}
```

```
array_ref<T, dimensions<N, 0, L> > A(buf, M);

A.rank()         == 3
A.rank_dynamic() == 1
A.extent(0)      == N
A.extent(1)      == M
A.extent(2)      == L
A.size()         == A.extent(0) * A.extent(1) * A.extent(2)
```

```
array_ref<T, dimensions<N, 0, L> > A(buf, M);

A.rank()         == 3
A.rank_dynamic() == 1
A.extent(0)      == N
A.extent(1)      == M
A.extent(2)      == L
A.size()         == N * M * L
```

```
array_ref<T, dimensions<N, 0, L> > A(buf, M);

A.stride(0) == M * L;
A.stride(1) == L;
A.stride(2) == 1;
A.span()    == A.size();
```

```
array_ref<T[N]> A(buf0);

A[i]
A(i)

array_ref<T, dimensions<N, M, L> > B(buf1);

B(i, j, k)
```

Iterators

- An `array_ref<>` which is contiguous provides random access iterators.
- All other `array_ref<>`s provide no iterators.

- No “multi-dimensional” iterators.
- Multi-dimensional iterators are hard.

```
// ...
struct md_iterator
{
    array_ref</* ... */> ar;
    array<size_t, /* ... */> idx;
};
```

```cpp
void md_iterator::increment()
{
    ++idx[rank - 1];

    if (idx[rank - 1] == ar.extent(rank - 1))
    {
        idx[rank - 1] = 0;

        ++idx[rank - 2];

        // ...
    }
}
```

```
for (int j = 0; j < M; ++j)
    for (int i = 4; i < N - 4; ++i)
        u(i, j) = v(i, j)
                + c0 * (v(i+1, j) + v(i-1, j))
                + c1 * (v(i+2, j) + v(i-2, j))
                + c2 * (v(i+3, j) + v(i-3, j))
                + c3 * (v(i+4, j) + v(i-4, j));
```

```
for (md_iterator v = /* ... */)
{
    u[0] = v[0]
         + c0 * (v[1] + v[-1])
         + c1 * (v[2] + v[-2])
         + c2 * (v[3] + v[-3])
         + c3 * (v[4] + v[-4]);
}
```

Iterators

- A possible fast multi-dimensional iterator:
  - Stores only the current position in the reference memory region (a 1D index), and a copy of the array_ref<>.
  - Increment becomes fast.
  - Multi-dimensional index cannot be recovered cheaply.
    - Recovery involves integer divides.
  - Relative indexing is possible though.
- What about a proxy iterator which iterates the index space?
  - Performance concerns relating to auto-vectorization.
  - I'm not sold on the usefulness of such an iterator.

```
namespace std {
namespace experimental {

template <class T, class... Properties>

class array_ref;

}}
```

```
namespace std {
namespace experimental {

template <class T, class Dimensions, class Layout,
          class... Properties>
class array_ref;

}}
```

```cpp
namespace std {
namespace experimental {
namespace array_property {

class layout_left;  // Column-major, e.g. Fortran/MATLAB
class layout_right; // Row-major (default), e.g. C++
template <size_t... Ordering> class layout_order;
class layout_stride;

}}}
```

```cpp
namespace std {
namespace experimental {
namespace array_property {

template <class Striding, class Padding>
class basic_layout_left;

}}}
```

```
array_ref<T, dimensions<N, M, L> >               A(buf0);
array_ref<T, dimensions<N, M, L>, layout_left>  B(buf1);
array_ref<T, dimensions<N, M, L>, layout_right> C(buf2);

A(i, j, k) // *(A.data() + i*M*L + j*L + k)
B(i, j, k) // *(B.data() + i*M*L + j*L + k)
C(i, j, k) // *(C.data() + i + j*N + k*N*M)
```

```
namespace std {
namespace experimental {
namespace array_property {

class bounds_checking;

}}}

array_ref<T[N], bounds_checking> A(buf);
```

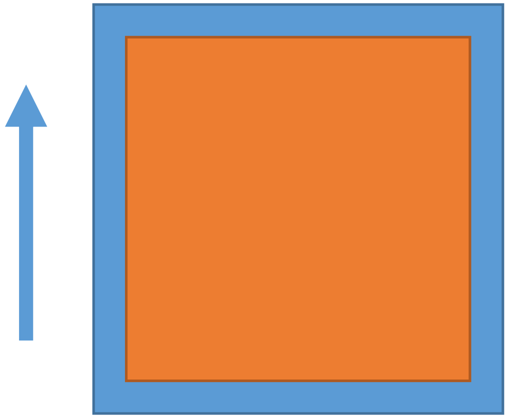
```
using bounds_checking_if_debug =
    conditional_t<DEBUG, bounds_checking, void>;

array_ref<T[N], bounds_checking_if_debug> A(buf);
```

```
namespace std {
namespace experimental {
namespace array_property {

class no_aliasing;

}}}
```

```
namespace std {
namespace experimental {
namespace array_property {

constexpr /* unspecified */ all = /* ... */;

}

template <class T, class... Properties,
          class... SliceSpecs>
/* unspecified array_ref<> */
subarray(array_ref<T, Properties...> ar,
         SliceSpecs... specs) noexcept;

}}
```

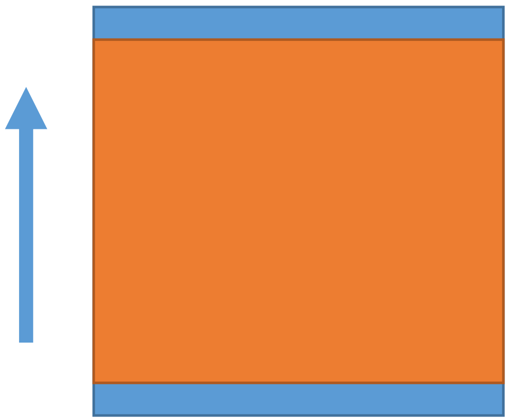
```cpp
// N > 2, M > 2
array_ref<T, dimensions<N, M> > A(buf);

auto B = subarray(A, {1, N-2}, {1, M-2});

B.rank()            == 2
B.is_contiguous() == false
```

```
// N > 2, M > 2
array_ref<T, dimensions<N, M> > A(buf);

auto B = subarray(A, {1, N-2}, {1, M-2});

B.rank()          == 2
B.is_contiguous() == false
```



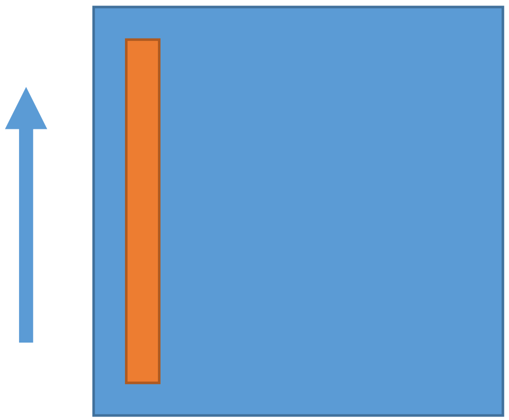
```
// N > 2, M > 2
array_ref<T, dimensions<N, M> > A(buf);

auto B = subarray(A, all, {1, M-2});

B.rank()          == 2
B.is_contiguous() == false
```

```cpp
// N > 2, M > 2
array_ref<T, dimensions<N, M> > A(buf);

auto B = subarray(A, all, {1, M-2});

B.rank()          == 2
B.is_contiguous() == false
```



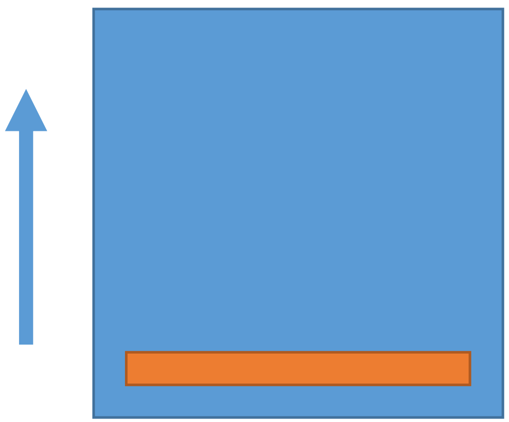
```
// N > 2, M > 2
array_ref<T, dimensions<N, M> > A(buf);

auto B = subarray(A, 1, {1, M-2});

B.rank()          == 1
B.is_contiguous() == true
```

```
// N > 2, M > 2
array_ref<T, dimensions<N, M> > A(buf);

auto B = subarray(A, 1, {1, M-2});

B.rank()          == 1
B.is_contiguous() == true
```



```
// N > 2, M > 2
array_ref<T, dimensions<N, M> > A(buf);

auto B = subarray(A, {1, N-2}, 1);

B.rank()            == 1
B.is_contiguous() == false
```

```
// N > 2, M > 2
array_ref<T, dimensions<N, M> > A(buf);

auto B = subarray(A, {1, N-2}, 1);

B.rank()            == 1
B.is_contiguous() == false
```



```cpp
array_ref<double, dimensions<0>, layout_right> u;
array_ref<double const, dimensions<0, 0>, layout_right> A;
array_ref<double const, dimensions<0>, layout_right> x;

for (int j = 0; j < A.extent(1); ++j)
    for (int i = 0; i < A.extent(0); ++i)
        u(i) += A(i, j) * x(j);
```

```cpp
array_ref<double, dimensions<0>, layout_right> u;
array_ref<double const, dimensions<0, 0>, layout_right> A;
array_ref<double const, dimensions<0>, layout_right> x;

for (int j = 0; j < A.extent(1); ++j)
{
   auto a = subarray(A, all, j);

   for (int i = 0; i < A.extent(0); ++i)
       u(i) += a(i) * x(j);
}
```

Things I'd Like

- A better dimension specifier mechanism.
- Some sort of multi-dimensional container wrapper.
- A performant abstraction to replace my raw multi-dimensional loops.

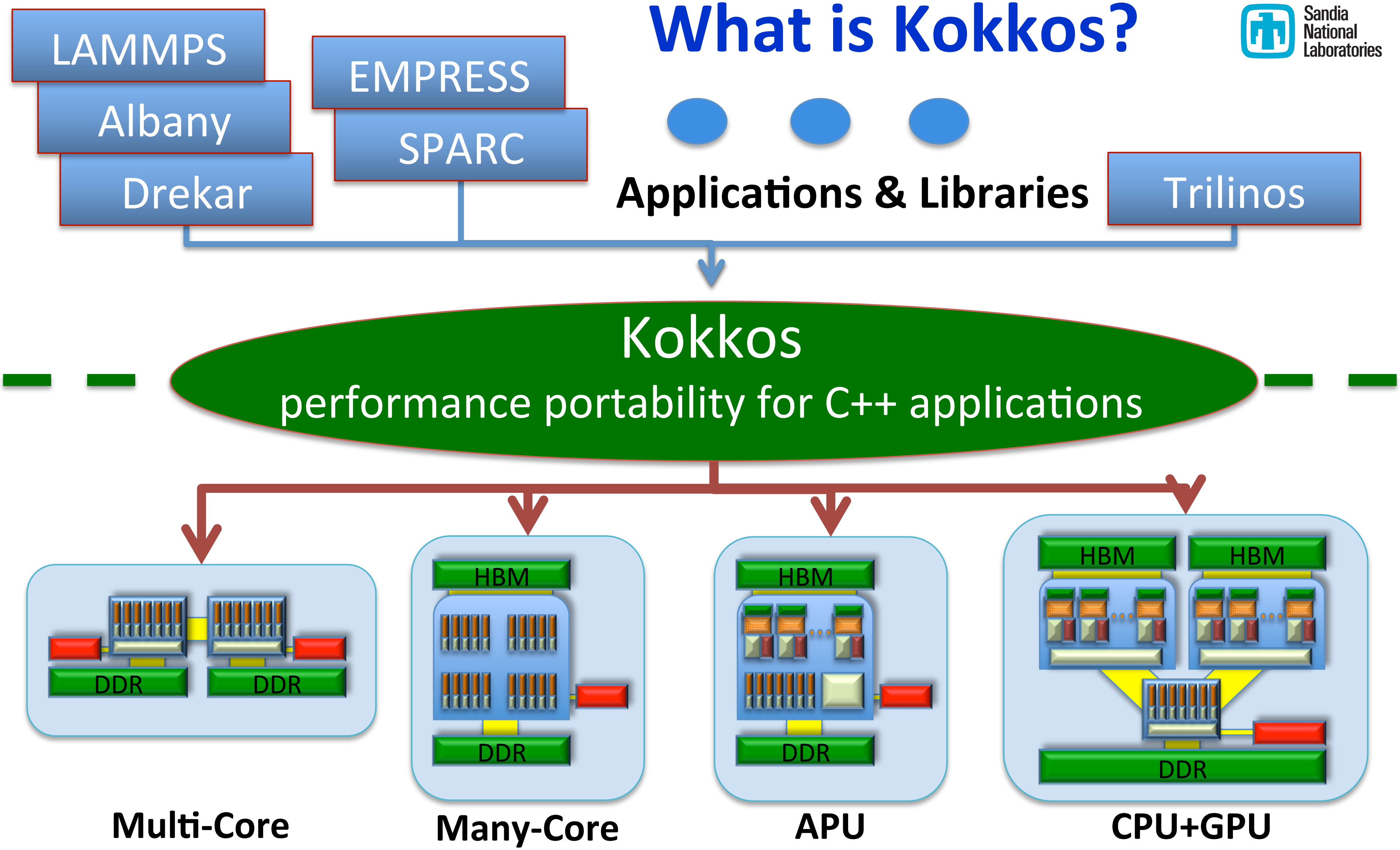
Part 1: Kokkos

Inspiration for array_ref

Part 2: Future Directions for array_ref

Six years of lessons learned with Kokkos

What is Kokkos?



Cornerstone for performance portability across next generation HPC architectures at multiple DOE laboratories, and other organizations.

What is *Kokkos*?

Sandia National Laboratories

- **κόκκος** (Greek, not an acronym)
  - Translation: "granule" or "grain" ; *like grains of sand on a beach*
- **Performance Portable Thread-Parallel Programming Model**
  - E.g., "X" in "MPI+X" ; **not** a distributed-memory programming model
  - Application identifies its parallelizable grains of computations *and* data
  - Kokkos maps those computations onto cores *and* that data onto memory
- **Fully Performance Portable Library Implementation using C++11**
  - ***Not* a language extension (e.g., OpenMP, OpenACC, OpenCL, …)**
  - **Open source at https://github.com/kokkos/kokkos**

✓ **Multicore CPU** - including NUMA architectural concerns

✓ **Intel Xeon Phi (KNC)** – toward DOE's Trinity (ATS-1) supercomputer

✓ **NVIDIA GPU (Kepler)** – toward DOE's Sierra (ATS-2) supercomputer

✧ **IBM Power 8** – toward DOE's Sierra (ATS-2) supercomputer

✧ **AMD Fusion** – back-end in collaboration with AMD via HCC

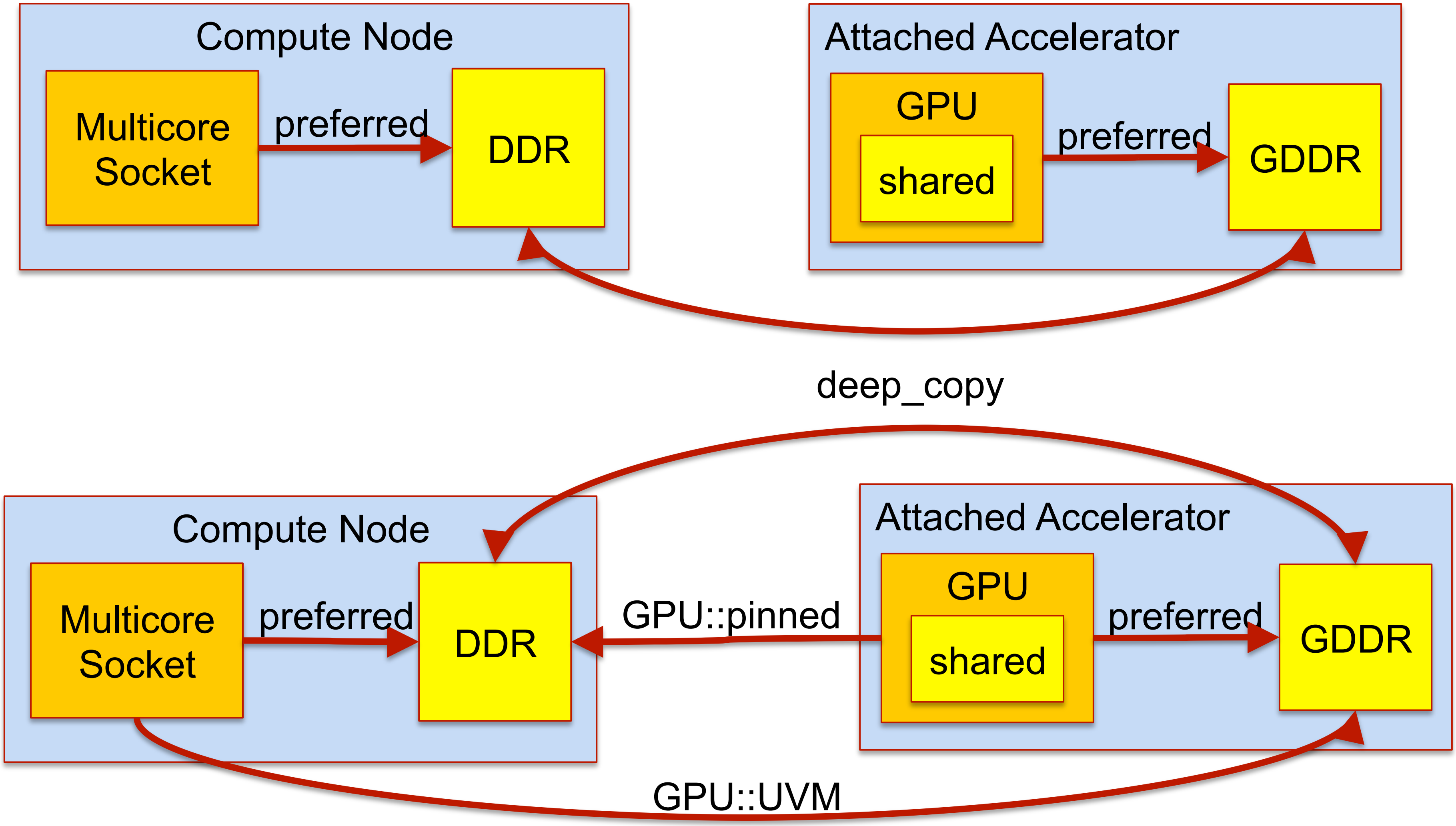
✓ Regularly tested

✧ Ramping up testing

Abstractions: Patterns, Policies, and Spaces

- Parallel Pattern of user's computations
  - parallel_for, parallel_reduce, parallel_scan, task-graph, ... *(extensible)*
- Execution Policy tells *how* user computation will be executed
  - Static scheduling, dynamic scheduling, thread-teams, ... *(extensible)*
- Execution Space tells *where* user computations will execute
  - Which cores, numa region, GPU, ... *(extensible)*
- Memory Space tells *where* user data resides
  - Host memory, GPU memory, high bandwidth memory, ... *(extensible)*
- Layout (policy) tells *how* user array data is laid out in memory
  - Row-major, column-major, array-of-struct, struct-of-array … *(extensible)*
- **Differentiating: Layout and Memory Space**
  - Versus other programming models (OpenMP, OpenACC, …)
  - Critical for performance portability …

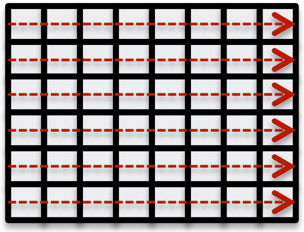
Examples of Execution and Memory Spaces

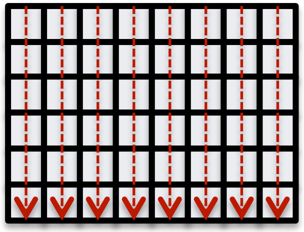


Layout Abstraction: Multidimensional Array

- **Classical (50 years!) data pattern for science & engineering codes**
  - Computer languages hard-wire multidimensional array layout mapping
  - Problem: different architectures *require* different layouts for performance
  - ➢ **Leads to architecture-specific versions of code to obtain performance**
  - E.g., “Array of Structure” ↔ “Structure of Array” redesigns

e.g., “row-major” CPU caching





e.g., “column-major” GPU coalescing

- **Kokkos *separates* layout from user’s computational code**
  - *Choose* layout for architecture-specific memory access pattern
    - ➢ **Without modifying user’s computational code**
  - **Polymorphic** layout via C++ template meta-programming *(extensible)*
    - ➢ **e.g., Hierarchical Tiling layout** (array of structure of array)
- **Bonus: easy/transparent use of special data access hardware**
  - Atomic operations, GPU texture cache, ... *(extensible)*

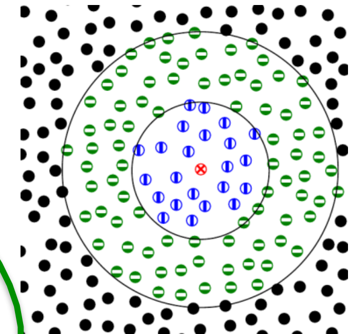
Performance Impact of Data Layout

- **Molecular dynamics computational kernel in miniMD**
- **Simple Lennard Jones force model:**

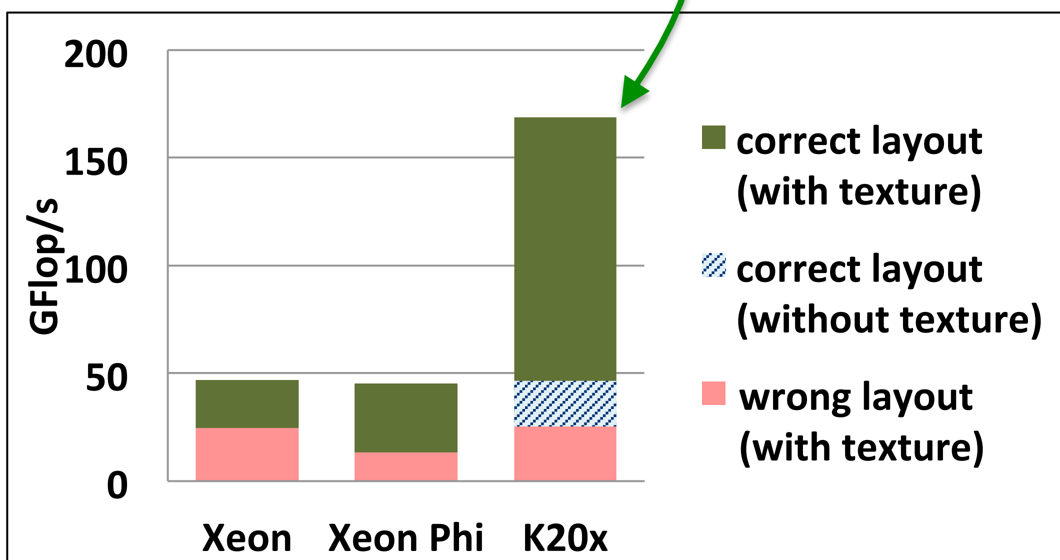
$$F_i = \sum_{j,\, r_{ij} < r_{cut}} 6\varepsilon \left[\left(\frac{\varsigma}{r_{ij}} \right)^7 - 2 \left(\frac{\varsigma}{r_{ij}} \right)^{13} \right]$$

- **Atom neighbor list to avoid N^2 computations**

```
pos_i = pos(i);
for( jj = 0; jj < num_neighbors(i); jj++) {
  j = neighbors(i,jj);
  r_ij = pos(i,0..2) – pos(j,0..2); // random read 3 floats
  if (|r_ij| < r_cut) f_i += 6*e*((s/r_ij)^7 – 2*(s/r_ij)^13)
}
f(i) = f_i;
```



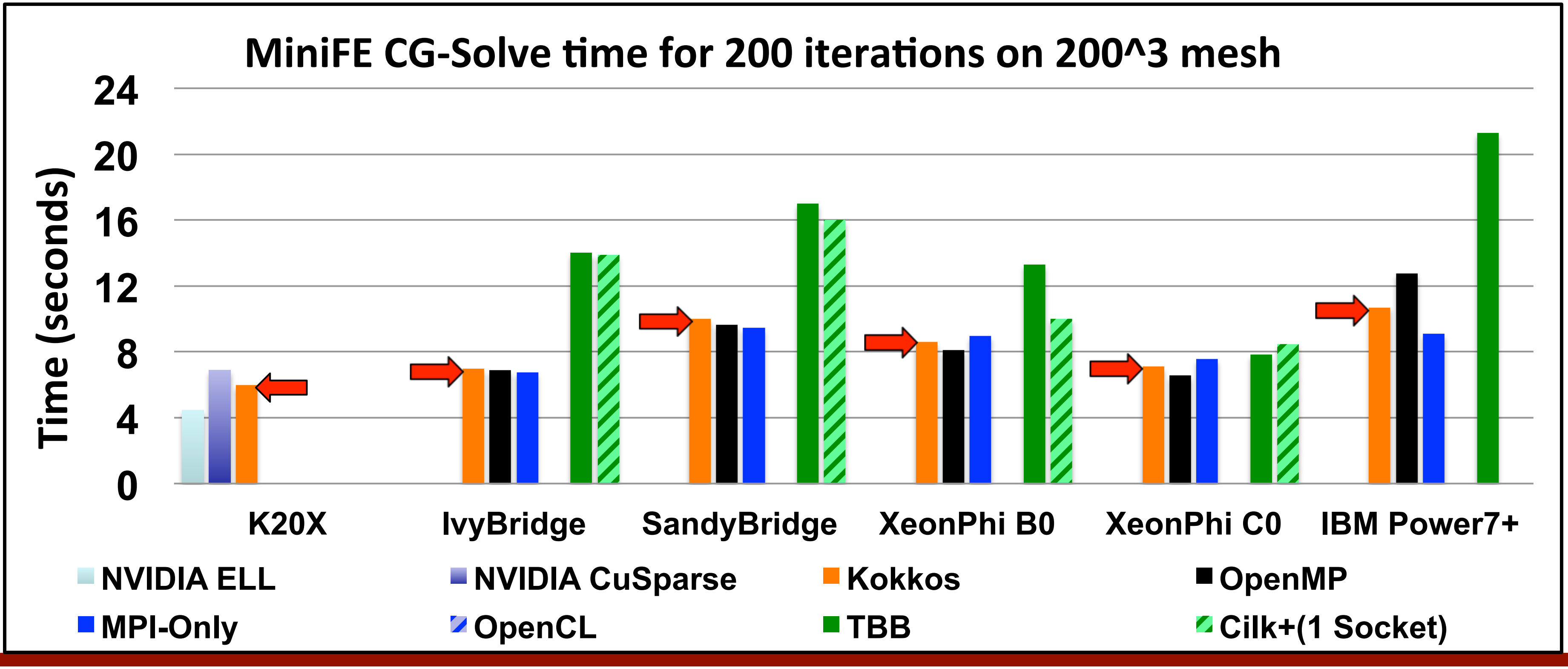
- **Test Problem**
  - 864k atoms, ~77 neighbors
  - 2D neighbor array
  - Different layouts CPU vs GPU
  - Random read 'pos' through GPU texture cache
- **Large performance loss with wrong data layout**



Performance Overhead?

Kokkos is competitive with other programming models

- **Regularly performance-test mini-applications on Sandia's ASC/CSSE test beds**
- **MiniFE: finite element linear system iterative solver mini-app**
  - **Compare to versions with architecture-specialized programming models**



Performance Portability & Future Proofing

Integrated mapping of users' parallel computations *and* data through abstractions of patterns, policies, spaces, *and* layout.

- **Versus other thread parallel programming models (mechanisms)**
  - OpenMP, OpenACC, OpenCL, ... have parallel execution
  - OpenMP 4 finally has execution spaces; when memory spaces ??
  - ➢ **All of these neglect data layout mapping**
    - Requiring significant code refactoring to change data access patterns
    - Cannot provide *performance* portability
  - ➢ **All require language and compiler changes for extension**
- **Kokkos extensibility "future proofing" wrt evolving architectures**
  - Library extensions, not compiler extensions
  - E.g., Intel KNL high bandwidth memory ← just another memory space

Mapping Parallel Computations

- *Pattern* composed with *policy* drives execution of *closure*

pattern policy closure

```
Kokkos::parallel_for ( N , [=]( int i ) { /* body */ } );
```

- Data parallel patterns
  - Kokkos::parallel_for
  - Kokkos::parallel_reduce
  - Kokkos::parallel_scan
- Data parallel execution policies
  - Kokkos::RangePolicy< *ExecSpace* >(*integral_begin* , *integral_end*)
  - Kokkos::TeamPolicy< *ExecSpace* >(*league_size* , *team_size*)
  - N *implies* Kokkos::RangePolicy< *DefaultExecSpace* >(0 , N)
- Simplicity of use is comparable to OpenMP
  - Reduce is far simpler to customize than OpenMP
  - Scan is not even an option in OpenMP

Mapping Execution onto ExecSpace

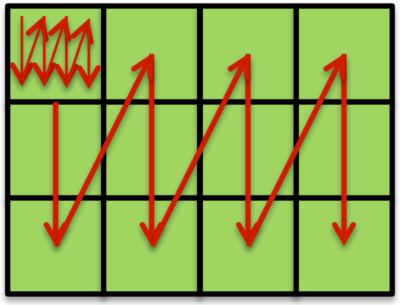
- Markups for ExecSpace Portability
  - CUDA: #define KOKKOS_FUNCTION __device__ __host__
    - Lambda capture markup supported in CUDA 8.0,
      came about through intense prodding of NVIDIA by DOE laboratories
    - Exposed the now resolved (C++17) lambda-capture-*this issue
  - HCC: #define KOKKOS_FUNCTION __attribute__((amp,cpu))
  - CPU: #define KOKKOS_FUNCTION /* nothing needed */
- Mapping RangePolicy $i \in [0..N)$
  - CUDA Space: i = threadIdx + blockDim * blockIdx ; strided partitions
  - CPU Space: $i \in [begin,end)_{Th}$; contiguous partitions to threads
- Inter-thread computations for value of reduce and scan
  - Thread-local values for partial sums (or other reduction operator)
  - Inter-thread *join* of thread-local values
  - User extensible *type*, *init*, and *join* of reduction value

Kokkos' Multidimensional Array <u>View</u>

- **Development started 2010, predates array_ref proposal**
- **Kokkos::View< double**[3][8] , Space > a("a",N,M);**
  - **Allocate array data in memory Space with dimensions [N][M][3][8]**
  - **For compact syntax dynamic dimensions denoted by ***
    - **Initially got away with [] until warnings-as-errors**
- **a(i,j,k,l) : User's access to array datum**
  - **"Space" accessibility enforced; e.g., GPU code cannot access CPU memory**
  - **Optional array bounds checking of indices for debugging**
- ***View* Semantics: View<double**[3][8],Space> b = a ;**
  - **Analogous to std::shared_ptr**
  - **A <u>shallow</u> assignment: 'a' and 'b' are *references* to the same allocated data**
- **Kokkos::deep_copy(destination_view , source_view);**
  - **Copy data from 'source_view' to 'destination_view'**
  - **Kokkos policy: make expensive deep copy operations very obvious**

Polymorphic Multidimensional Array Layout

- Layout mapping : a(i,j,k,l) → memory location
  - Layout is polymorphic, defined at compile time
  - Kokkos chooses default array layout appropriate for "Space"
  - E.g., row-major, column-major, Morton ordering, dimension padding, ...
- User can specify Layout : View< ArrayType, *Layout*, Space >
  - Override Space's preferred array layout
  - Why? For compatibility with legacy code, algorithmic performance tuning, ...
- Example Tiling Layout
  - View<double**,Tile<8,8>,Space> m("matrix",N,N);
  - Tiling layout transparent to user code : m(i,j) unchanged
  - Layout-aware algorithm extracts tile subview



Multidimensional Array Subview & Properties

- **Array subview of array view**
  - **Y = subview(X , *...ranges_and_indices_argument_list...*);**
  - **View of same data, with the appropriate layout and index map**
  - **Each index argument eliminates a dimension**
  - **Each range [begin,end) argument contracts a dimension**
    - **pair<iType,iType>(begin,end) or { begin , end }**

- **Access intent Properties**

  View< ArrayType, Layout, Space, *AccessProperties* >

  - **How user intends to access datum**
  - **Example, View with const and random access intension**
    - **View< double ** , Cuda > a("mymatrix", N, M);**
    - **View< const double **, Cuda, RandomAccess > b = a ;**
    - ➢ **Kokkos implements b(i,j) with GPU texture cache**

Managing Memory Access Pattern:

Compose Parallel Execution ○ Array Layout

- **Recall mapping of parallel execution**
  - **Maps calls to closure(iw) onto threads**
  - **GPU: iw = threadIdx + blockDim * blockIds**
  - **CPU: iw ∈[begin,end)$_{Th}$; contiguous partitions among threads**
- **Choose array layout**
  - **Leading dimension is parallel work dimension**
    - **Leading multi-index is 'iw' : a(iw , j, k, l)**
  - **Choose appropriate array layout for space's architecture**
    - **E.g., row-major for CPU and column-major for GPU**
- **Fine-tune Array Layout**
  - **E.g., padding dimensions for cache line alignment**

Part 1: Kokkos

Inspiration for array_ref

Part 2: Future Directions for array_ref

Six years of lessons learned with Kokkos

Compact (Relaxed) Array Type Declaration

Sandia National Laboratories

- **array_ref< T , array_property::dimension< ...dims > >**
  - Very user unfriendly
  - Especially for mathematicians, engineers, and scientists – target stakeholders
  - Especially if using array_property::dynamic_extent
    - experience a rank-6 array_ref with 5 dynamic extents
- **array_ref< T[][][][][][3] > // preferred syntax**
  - Original syntax for Kokkos worked well, until warnings-as-errors
  - Kokkos users universally preferred this syntax
  - LEWG had consensus on preferring this syntax
- **Preferred syntax requires trivial change to language**
  - One line change to Clang to stop generating an error
  - Accepted by gcc until v5 (without warnings-as-errors)
  - Well-defined change to Arrays paragraph : n4567 p8.3.4.p3
  - ➢ Omission of any static bound after the first defines an incomplete object type

When array_ref::reference does not alias an lvalue reference

- Recall: Kokkos::View< const *ArrayType* , Cuda, RandomAccess >
  - operator()(i,j,…) reads data through GPU texture cache
  - return by value, not by const lvalue reference
  - not lvalue reference => disallowed to &a(i,j,k)
- Another use case: Kokkos::View< *ArrayType* , *Space* , Atomic >
  - operator()(i,j,…) returns an atomic view concept (P0019) proxy
  - allowed to a(i,j,k).fetch_add(value)
- Perhaps for convenience:

```
array_ref {
 static constexpr bool is_lvalue_reference_v =
  std::is_lvalue_reference_v< reference > ;
}
```

Shared Ownership and Allocating Constructor

- **Users appreciate View's shared ownership and allocating ctor**
  - Reference to array data semantics preserved
  - Users have a single interface, avoid juggling multiple objects
  - Avoids multistep allocation process: compute size, allocate, wrap
- **array_ref< … , array_property::Shared >**

```
template < class D , class A > array_ref( D , A , pointer , dimensions... );
```

  - Conformal to std::shared_ptr deleter and allocator

```
template < class A > array_ref( A , dimensions… );
```

  - Allocate, initialize, destroy, and deallocate via A

```
use_count() const noexcept ;
```

  - Conformal to std::shared_ptr
- **As if data member was std::shared_ptr instead of pointer**

Memory Space (Memory Resource)

- **Modern architectures have non-trivial memory spaces**
  - **DDR NUMA regions on CPU**
  - **GDDR and programmable L1 (a.k.a., __shared__ memory) on GPU**
  - **HBM, NVRAM, …**
  - **… with kernel properties; e.g., GPU UVM, pinned**
- **Use *concept* of C++17 memory_resource for memory spaces**
  - **Safety and performant utilization requires type information**
    - **When can/cannot be accessed, specialized instructions**
    - **is_memory_resource< *Space* >**
- **array_ref< … , *Space* >**

array_ref(*Space* , dimensions…);

  - **Allocate and deallocate via *Space***

Space & memory_resource() const noexcept ;

Performance Hint Properties

- **In the current scope …**
- **array_ref< … , array_property::Restrict >**
  - **Declares exclusive reference to array elements**
- **array_ref< … , array_property::Once >**
  - **Declares elements are accessed only once and need not be cached**
- **array_ref< … , array_property::Random >**
  - **Declares elements are accessed essentially randomly**
  - **Recall Kokkos' GPU + const + Random => use texture cache**
- **array_ref< … , array_property::CheckBounds >**
  - **Indexing operator performs bounds checking**
- **… alternative to [[*attribute-list*]] on array objects**
- **… boundless opportunities for bike-shedding names**

array_ref Property Pack Management

- **For ease of use, apply and remove meta functions**
- **array_property::apply< array_ref<…> , *property* >::type**
  - **Add *property* to the array_ref property pack**
- **array_property::remove< array_ref< … > , *property* >::type**
  - **Remove *property* from the array_ref property pack**

- **Assignability with non-identical properties**

template< typename UT , class … UP , typename VT , class … VP >

array_property::is_assignable< array_ref<UT,UP…> , array_ref<VT,VP…> >

  - **Conceptually analogous to cv-qualification rules**
  - **Compatibility of data type, rank, static dimensions, layout, …**

User Defined Layout::mapping

- **array_ref may be optimized for standard layouts**
- **User defined Layout::mapping is a common need**
  - **Tiling, symmetric tensor folding, space filling curve, …**
- ***Concept* of Layout::mapping for performant extensibility**
  - **indexing: constexpr size_type offset(… *indices*) const noexcept ;**
  - **construct: mapping(… dynamic_dimensions), mapping(layout)**
  - **domain properties: rank(), extent(i)**
  - **range properties: is_regular(), is_contiguous(), span(), stride(i)**
- **One catch: integration with subarray is challenging**
  - **Optimization is work-in-progress within Kokkos library**

Future Directions: Priorities and Plans

1. **Start with foundational capability**
   - Property pack limited to
     - dimension
     - Predefined standard layouts
2. **Relax array incomplete type declaration: T[ ][3][ ]**
3. **Shared ownership property with allocating constructors**
   - Also property pack management: apply, remove
4. **Memory space property with memory resource**
   - Requires memory space concept
5. **Performance hint properties**
6. **Extensible layout**
   - More experience needed with subarray integration