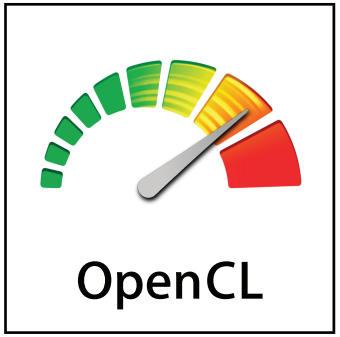
# OpenCL:
# A Hands-on Introduction

Tim Mattson
Intel Corp.




Alice Koniges
Berkeley Lab/NERSC

Simon McIntosh-Smith
University of Bristol

Acknowledgements: In addition to Tim, Alice and Simon … Tom Deakin (Bristol) and Ben Gaster (Qualcomm) contributed to this content.

# Agenda

| Lectures | Exercises |
|---|---|
| An Introduction to OpenCL | Logging in and running the Vadd program |
| Understanding Host programs | Chaining Vadd kernels together |
| Kernel programs | The D = A + B + C problem |
| Writing Kernel Programs | Matrix Multiplication |
| Lunch | |
| Working with the OpenCL memory model | Several ways to Optimize matrix multiplication |
| High Performance OpenCL | Matrix multiplication optimization contest |
| The OpenCL Zoo | Run your OpenCL programs on a variety of systems. |
| Closing Comments | |

# Course materials

**In addition to these slides, C++ API header files, a set of exercises, and solutions, we provide:**


OpenCL API 1.1 Quick Reference Card - Page 1

OpenCL (Open Computing Language) is a multi-vendor open standard for general-purpose parallel programming of heterogeneous systems that include CPUs, GPUs and other processors. OpenCL provides a uniform programming environment for software developers to write efficient, portable code for high-performance compute servers, desktop computer systems and handheld devices.

[n.n.n] refers to the section in the API Specification available at www.khronos.org/opencl.

The OpenCL Platform Layer

The OpenCL platform layer implements platform-specific features that allow applications to query OpenCL devices, device configuration information, and to create OpenCL contexts using one or more devices.

Contexts [4.4]

cl_context clCreateContext ( cl_context_properties *properties, cl_uint num_devices, const cl_device_id *devices, void (CL_CALLBACK *pfn_notify)( const char *errinfo, const void *private_info, size_t cb, void *user_data), void *user_data, cl_int *errcode_ret)

properties: CL_CONTEXT_PLATFORM, CL_GL_CONTEXT_KHR, CL_CGL_SHAREGROUP_KHR, CL_EGL_DISPLAY_KHR, CL_WGL_HDC_KHR

cl_context clCreateContextFromType ( const cl_context_properties *properties, cl_device_type device_type, void (CL_CALLBACK *pfn_notify)( const char *errinfo, const void *private_info, size_t cb, void *user_data), void *user_data, cl_int *errcode_ret)

properties: See clCreateContext

cl_int clRetainContext (cl_context context)

cl_int clReleaseContext (cl_context context)

cl_int clGetContextInfo (cl_context context, cl_context_info param_name, size_t param_value_size, void *param_value, size_t *param_value_size_ret)

param_name: CL_CONTEXT_REFERENCE_COUNT, CL_CONTEXT_DEVICES, CL_CONTEXT_PROPERTIES, CL_CONTEXT_NUM_DEVICES

Querying Platform Info and Devices [4.1, 4.2]

cl_int clGetPlatformIDs (cl_uint num_entries, cl_platform_id *platforms, cl_uint *num_platforms)

cl_int clGetPlatformInfo (cl_platform_id platform, cl_platform_info param_name, size_t param_value_size, void *param_value, size_t *param_value_size_ret)

param_name: CL_PLATFORM_PROFILE, CL_PLATFORM_VERSION, CL_PLATFORM_NAME, CL_PLATFORM_VENDOR, CL_PLATFORM_EXTENSIONS

cl_int clGetDeviceIDs (cl_platform_id platform, cl_device_type device_type, cl_uint num_entries, cl_device_id *devices, cl_uint *num_devices)

device_type: CL_DEVICE_TYPE_CPU, CL_DEVICE_TYPE_GPU, CL_DEVICE_TYPE_ACCELERATOR, CL_DEVICE_TYPE_DEFAULT, CL_DEVICE_TYPE_ALL

cl_int clGetDeviceInfo (cl_device_id device, cl_device_info param_name, size_t param_value_size, void *param_value, size_t *param_value_size_ret)

param_name: CL_DEVICE_TYPE, CL_DEVICE_VENDOR_ID, CL_DEVICE_MAX_COMPUTE_UNITS, CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS, CL_DEVICE_MAX_WORK_GROUP_SIZE, CL_DEVICE_MAX_WORK_ITEM_SIZES, CL_DEVICE_PREFERRED_VECTOR_WIDTH_CHAR, CL_DEVICE_PREFERRED_VECTOR_WIDTH_SHORT, CL_DEVICE_PREFERRED_VECTOR_WIDTH_INT, CL_DEVICE_PREFERRED_VECTOR_WIDTH_LONG, CL_DEVICE_PREFERRED_VECTOR_WIDTH_FLOAT, CL_DEVICE_PREFERRED_VECTOR_WIDTH_DOUBLE, CL_DEVICE_PREFERRED_VECTOR_WIDTH_HALF, CL_DEVICE_MAX_CLOCK_FREQUENCY, CL_DEVICE_ADDRESS_BITS, CL_DEVICE_MAX_MEM_ALLOC_SIZE, CL_DEVICE_IMAGE_SUPPORT, CL_DEVICE_MAX_READ_WRITE_IMAGE_ARGS, CL_DEVICE_IMAGE2D_MAX_WIDTH, HEIGHT, CL_DEVICE_IMAGE3D_MAX_WIDTH, HEIGHT, DEPTH, CL_DEVICE_MAX_SAMPLERS, CL_DEVICE_MAX_PARAMETER_SIZE, CL_DEVICE_MEM_BASE_ADDR_ALIGN, CL_DEVICE_MIN_DATA_TYPE_ALIGN_SIZE, CL_DEVICE_SINGLE_FP_CONFIG, CL_DEVICE_GLOBAL_MEM_CACHE_TYPE, SIZE, CL_DEVICE_GLOBAL_MEM_CACHELINE_SIZE, CL_DEVICE_GLOBAL_MEM_SIZE, CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE, ARGS, CL_DEVICE_LOCAL_MEM_TYPE, SIZE, CL_DEVICE_ERROR_CORRECTION_SUPPORT, CL_DEVICE_PROFILING_TIMER_RESOLUTION, CL_DEVICE_ENDIAN_LITTLE, CL_DEVICE_AVAILABLE, CL_DEVICE_COMPILER_AVAILABLE, CL_DEVICE_EXECUTION_CAPABILITIES, CL_DEVICE_QUEUE_PROPERTIES, CL_DEVICE_NAME, VENDOR, PROFILE, EXTENSIONS, CL_DEVICE_HOST_UNIFIED_MEMORY, CL_DEVICE_PLATFORM, CL_DEVICE_VERSION, CL_DRIVER_VERSION, CL_DEVICE_OPENCL_C_VERSION

The OpenCL Runtime

Command Queues [5.1]

cl_command_queue clCreateCommandQueue ( cl_context context, cl_device_id device, cl_command_queue_properties properties, cl_int *errcode_ret)

properties: CL_QUEUE_PROFILING_ENABLE, CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE

cl_int clRetainCommandQueue ( cl_command_queue command_queue)

cl_int clReleaseCommandQueue ( cl_command_queue command_queue)

cl_int clGetCommandQueueInfo ( cl_command_queue command_queue, cl_command_queue_info param_name, size_t param_value_size, void *param_value, size_t *param_value_size_ret)

param_name: CL_QUEUE_CONTEXT, CL_QUEUE_DEVICE, CL_QUEUE_REFERENCE_COUNT, CL_QUEUE_PROPERTIES

Buffer Objects

Elements of a buffer object can be a scalar or vector data type or a user-defined structure. Elements are stored sequentially and are accessed using a pointer by a kernel executing on a device. Data is stored in the same format as it is accessed by the kernel.

Create Buffer Objects [5.2.1]

cl_mem clCreateBuffer (cl_context context, cl_mem_flags flags, size_t size, void *host_ptr, cl_int *errcode_ret)

cl_mem clCreateSubBuffer (cl_mem buffer, cl_mem_flags flags, cl_buffer_create_type buffer_create_type, const void *buffer_create_info, cl_int *errcode_ret)

flags for clCreateBuffer and clCreateSubBuffer: CL_MEM_READ_WRITE, CL_MEM_WRITE_ONLY, CL_MEM_READ_ONLY, CL_MEM_USE, ALLOC, COPY_HOST_PTR

Read, Write, Copy Buffer Objects [5.2.2]

cl_int clEnqueueReadBuffer ( cl_command_queue command_queue, cl_mem buffer, cl_bool blocking_read, size_t offset, size_t cb, void *ptr, cl_uint num_events_in_wait_list, const cl_event *event_wait_list, cl_event *event)

cl_int clEnqueueWriteBuffer ( cl_command_queue command_queue, cl_mem buffer, cl_bool blocking_write, size_t offset, size_t cb, const void *ptr, cl_uint num_events_in_wait_list, const cl_event *event_wait_list, cl_event *event)

cl_int clEnqueueReadBufferRect ( cl_command_queue command_queue, cl_mem buffer, cl_bool blocking_read, const size_t buffer_origin[3], const size_t host_origin[3], const size_t region[3], size_t buffer_row_pitch, size_t buffer_slice_pitch, size_t host_row_pitch, size_t host_slice_pitch, void *ptr, cl_uint num_events_in_wait_list, const cl_event *event_wait_list, cl_event *event)

cl_int clEnqueueWriteBufferRect ( cl_command_queue command_queue, cl_mem buffer, cl_bool blocking_write, const size_t buffer_origin[3], const size_t host_origin[3], const size_t region[3], size_t buffer_row_pitch, size_t buffer_slice_pitch, size_t host_row_pitch, size_t host_slice_pitch, const void *ptr, cl_uint num_events_in_wait_list, const cl_event *event_wait_list, cl_event *event)

cl_int clEnqueueCopyBuffer ( cl_command_queue command_queue, cl_mem src_buffer, cl_mem dst_buffer, size_t src_offset, size_t dst_offset, size_t cb, cl_uint num_events_in_wait_list, const cl_event *event_wait_list, cl_event *event)

cl_int clEnqueueCopyBufferRect ( cl_command_queue command_queue, cl_mem src_buffer, cl_mem dst_buffer, const size_t src_origin[3], const size_t dst_origin[3], const size_t region[3], size_t src_row_pitch, size_t src_slice_pitch, size_t dst_row_pitch, size_t dst_slice_pitch, cl_uint num_events_in_wait_list, const cl_event *event_wait_list, cl_event *event)

Map Buffer Objects [5.2.3]

void * clEnqueueMapBuffer ( cl_command_queue command_queue, cl_mem buffer, cl_bool blocking_map, cl_map_flags map_flags, size_t offset, size_t cb, cl_uint num_events_in_wait_list, const cl_event *event_wait_list, cl_event *event, cl_int *errcode_ret)

Map Buffer Objects [5.2.3.1]

cl_int clRetainMemObject (cl_mem memobj)

cl_int clReleaseMemObject (cl_mem memobj)

cl_int clSetMemObjectDestructorCallback ( cl_mem memobj, void (CL_CALLBACK *pfn_notify)( cl_mem memobj, void *user_data), void *user_data)

cl_int clEnqueueUnmapMemObject ( cl_command_queue command_queue, cl_mem memobj, void *mapped_ptr, cl_uint num_events_in_wait_list, const cl_event *event_wait_list, cl_event *event)

Query Buffer Object [5.2.4]

cl_int clGetMemObjectInfo (cl_mem memobj, cl_mem_info param_name, size_t param_value_size, void *param_value, size_t *param_value_size_ret)

param_name: CL_MEM_TYPE, FLAGS, SIZE, HOST_PTR, CL_MEM_MAP_REFERENCE_COUNT, CL_MEM_CONTEXT, CL_MEM_ASSOCIATED_MEMOBJECT

Program Objects

Create Program Objects [5.4.1]

cl_program clCreateProgramWithSource ( cl_context context, cl_uint count, const char **strings, const size_t *lengths, cl_int *errcode_ret)

cl_program clCreateProgramWithBinary ( cl_context context, cl_uint num_devices, const cl_device_id *device_list, const size_t *lengths, const unsigned char **binaries, cl_int *binary_status, cl_int *errcode_ret)

cl_int clRetainProgram (cl_program program)

cl_int clReleaseProgram (cl_program program)

Build Program Executable [5.4.2]

cl_int clBuildProgram (cl_program program, cl_uint num_devices, const cl_device_id *device_list, const char *options, void (CL_CALLBACK *pfn_notify)( cl_program program, void *user_data), void *user_data)

Build Options [5.4.3]

Preprocessor options (processed in order listed in clBuildProgram): -D name, -D name=definition, -I dir

Optimization options: -cl-opt-disable, -cl-strict-aliasing, -cl-mad-enable, -cl-no-signed-zeros, -cl-finite-math-only, -cl-fast-relaxed-math, -cl-unsafe-math-optimizations

Math intrinsics: -cl-single-precision-constant, -cl-denorms-are-zero

Warning request/suppress: -w, -Werror

Control OpenCL C language version: -cl-std=CL1.1 // OpenCL 1.1 specification.

Query Program Objects [5.4.5]

cl_int clGetProgramInfo (cl_program program, cl_program_info param_name, size_t param_value_size, void *param_value, size_t *param_value_size_ret)

param_name: CL_PROGRAM_REFERENCE_COUNT, CONTEXT, NUM_DEVICES, DEVICES, SOURCE, BINARY_SIZES, BINARIES

(Program Objects Continue >)



www.khronos.org/opencl

**OpenCL C 1.2 Reference Card**
**OpenCL C++ 1.2 Reference Card**

These cards will help you keep track of the API as you do the exercises:

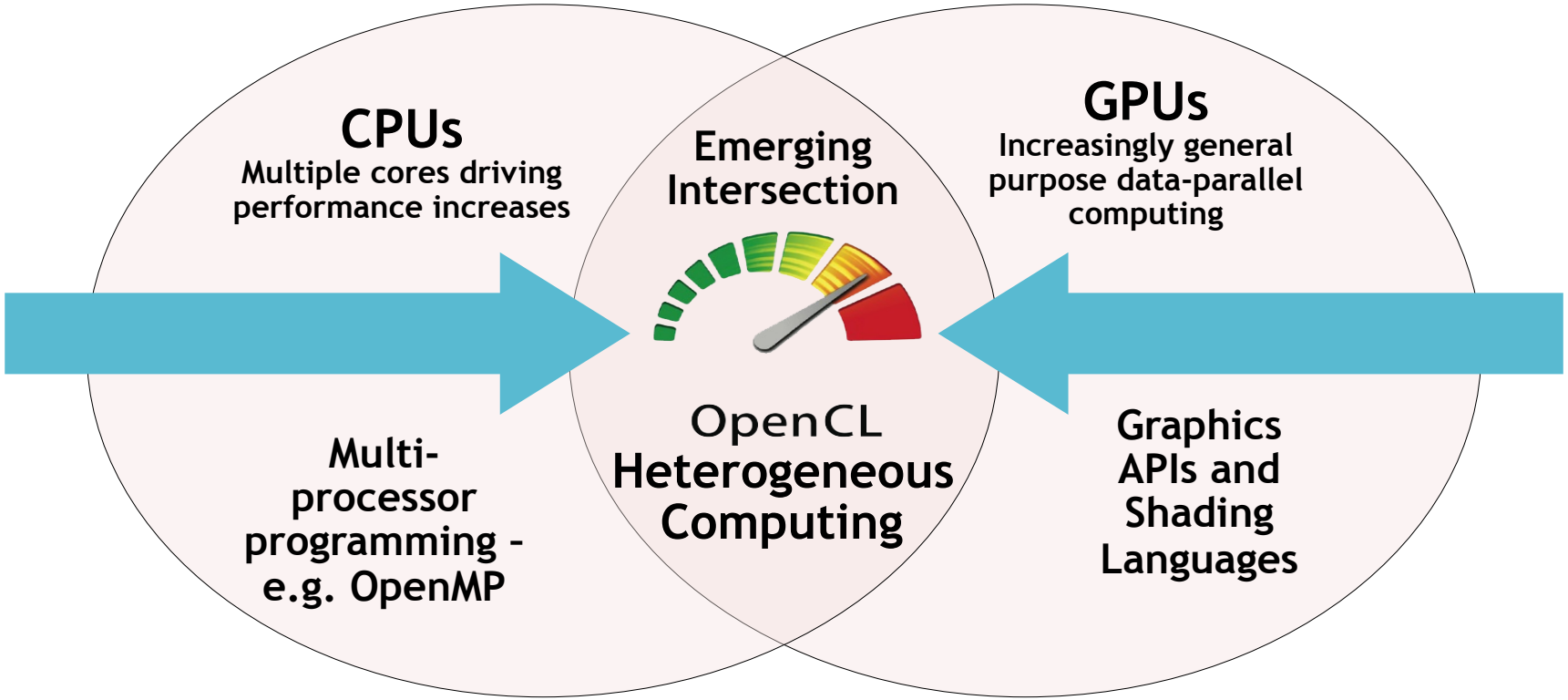
https://www.khronos.org/files/opencl-1-2-quick-reference-card.pdf

The v1.2 spec is also very readable and recommended to have on-hand:

https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

# AN INTRODUCTION TO OPENCL

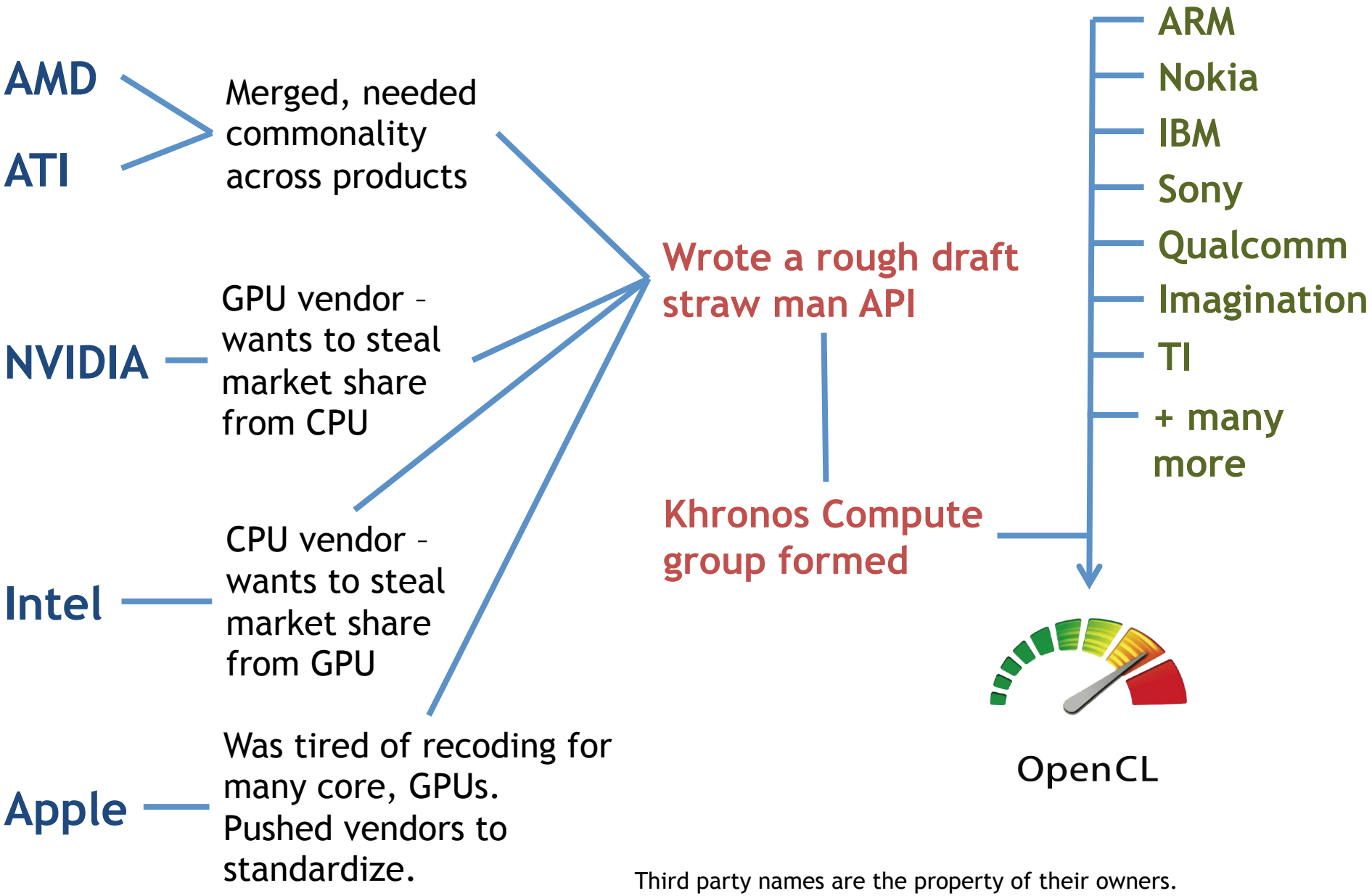
# Industry Standards for Programming Heterogeneous Platforms




## OpenCL – Open Computing Language

Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors

# The origins of OpenCL

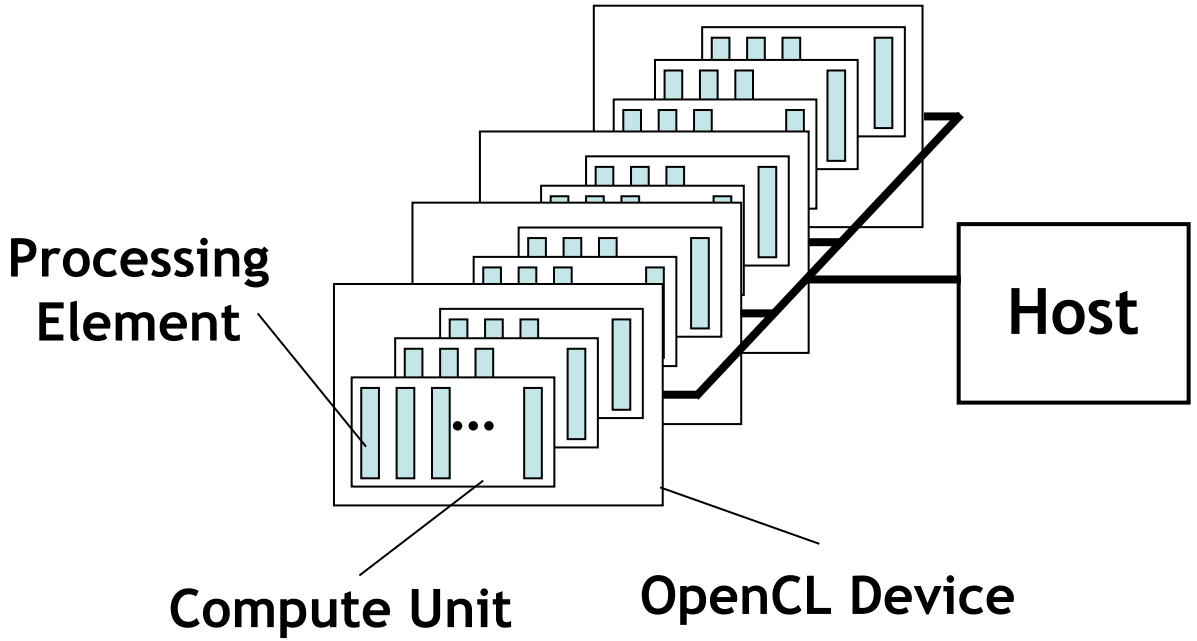


# OpenCL: From cell phone to supercomputer

- OpenCL Embedded profile for mobile and embedded silicon
  - Relaxes some data type and precision requirements
  - Avoids the need for a separate “ES” specification
- Khronos APIs provide computing support for imaging & graphics
  - Enabling advanced applications in, e.g., Augmented Reality
- OpenCL will enable parallel computing in new markets
  - Mobile phones, cars, avionics


A camera phone with GPS processes images to recognize buildings and landmarks and provides relevant data from internet

# OpenCL Platform Model




- One ***Host*** and one or more ***OpenCL Devices***
  - Each OpenCL Device is composed of one or more ***Compute Units***
    - Each Compute Unit is divided into one or more *Processing Elements*
- Memory divided into ***host memory*** and ***device memory***

# The BIG idea behind OpenCL

- Replace loops with functions (a kernel) executing at each point in a problem domain
  - E.g., process a 1024x1024 image with one kernel invocation per pixel or 1024x1024=1,048,576 kernel executions

## Traditional loops

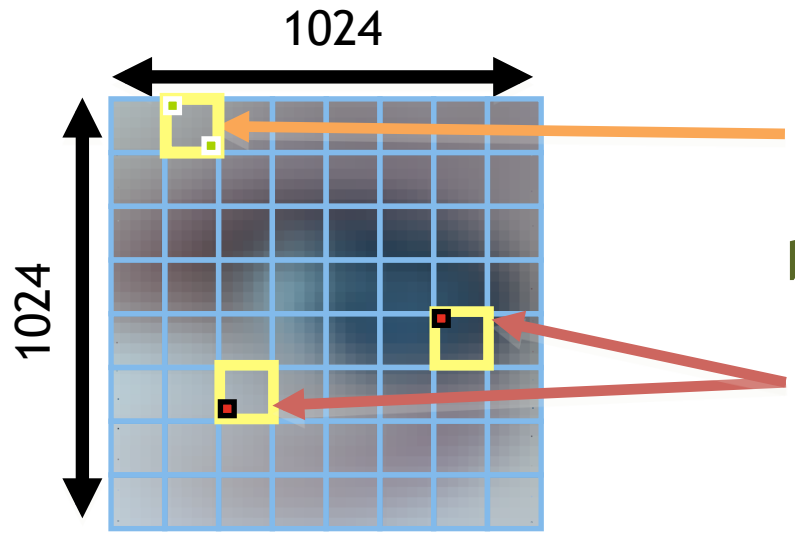
```
void
mul(const  int n,
    const  float *a,
    const  float *b,
           float *c)
{
  int i;
  for (i = 0; i < n; i++)
    c[i] = a[i] * b[i];
}
```

## OpenCL

```
__kernel void
mul(__global const float *a,
    __global const float *b,
    __global       float *c)
{
  int id = get_global_id(0);
  c[id]  = a[id] * b[id];
}
// execute over n work-items
```

# An N-dimensional domain of work-items

- **Global** Dimensions:
  - 1024x1024 (whole problem space)
- **Local** Dimensions:
  - 128x128 (**work-group**, executes together)



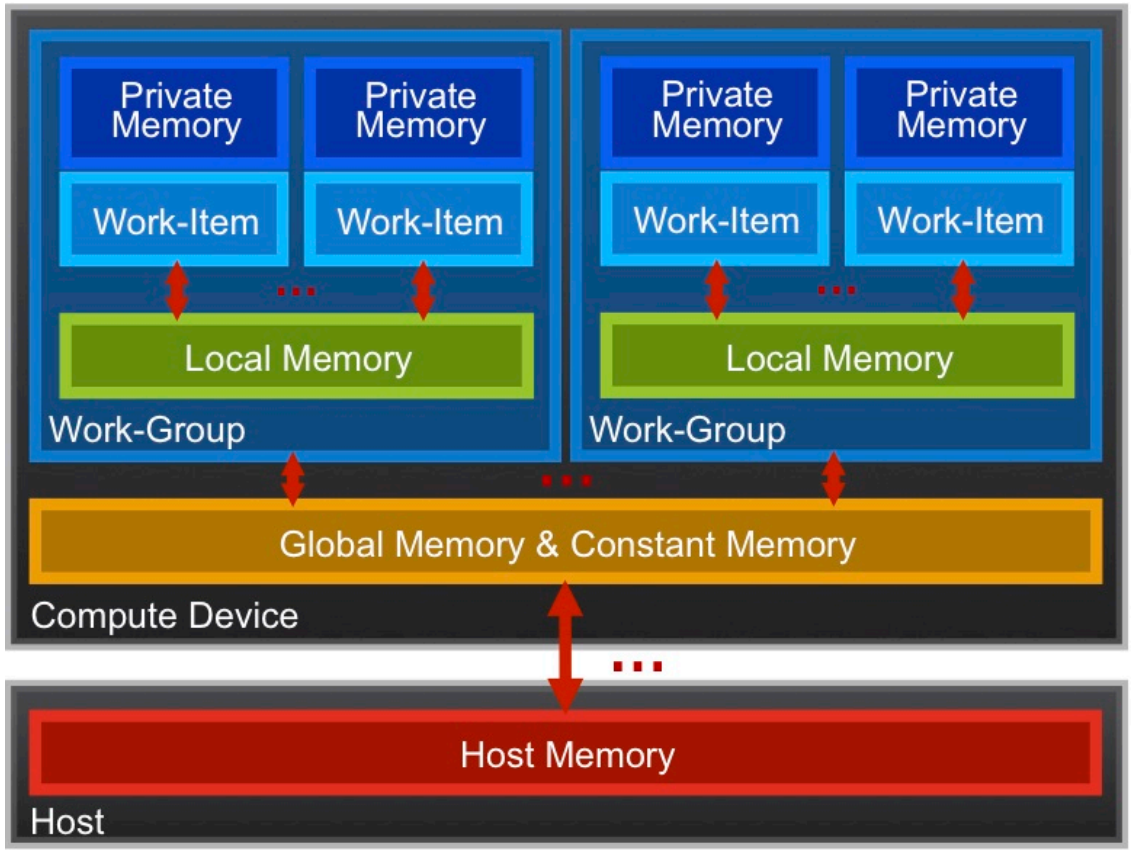

**Synchronization between work-items possible only within work-groups: barriers and memory fences**

**Cannot synchronize between work-groups within a kernel**

- Choose the dimensions (1, 2, or 3) that are "best" for your algorithm

# OpenCL Memory model

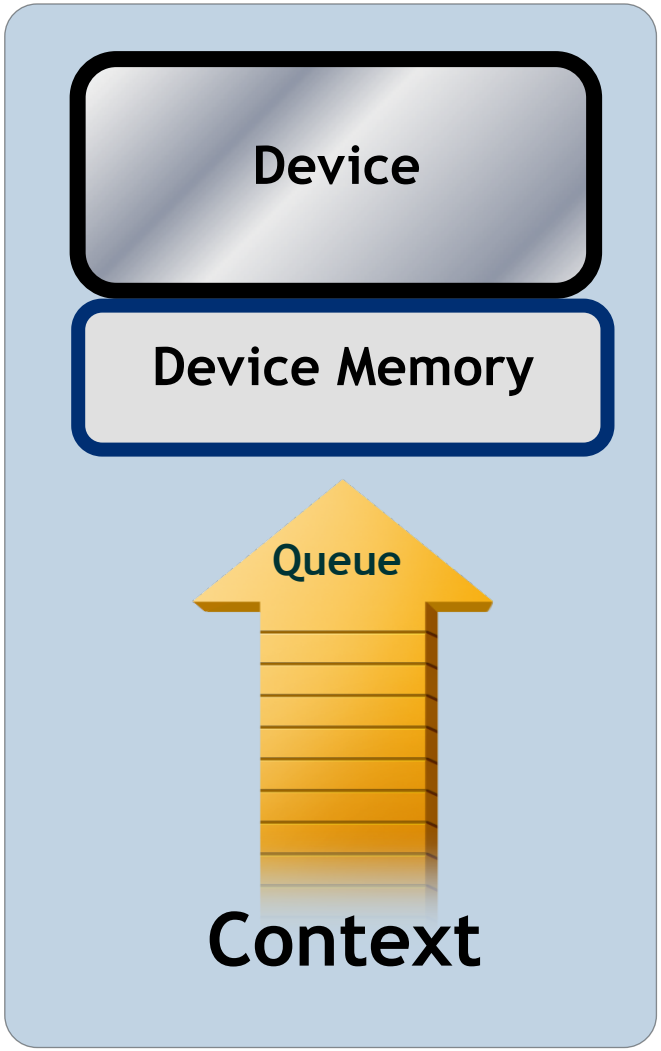
- ***Private Memory***
  - Per work-item
- ***Local Memory***
  - Shared within a work-group
- ***Global Memory***
  ***Constant Memory***
  - Visible to all work-groups
- ***Host memory***
  - On the CPU




Memory management is explicit:
You are responsible for moving data from
host → global → local *and* back

# Context and Command-Queues

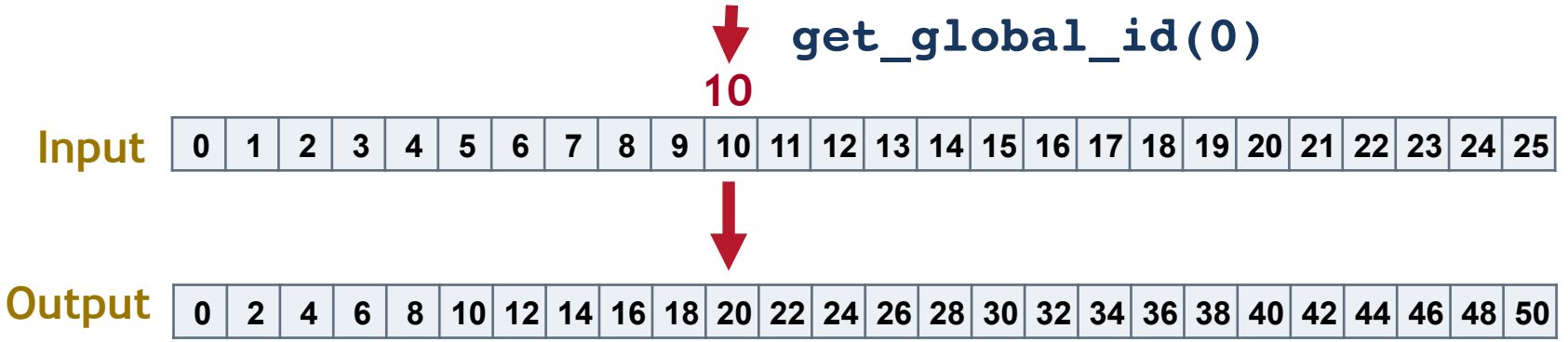
- ***Context***:
  - The environment within which kernels execute and in which synchronization and memory management is defined.
- The ***context*** includes:
  - One or more devices
  - Device memory
  - One or more command-queues
- All ***commands*** for a device (kernel execution, synchronization, and memory operations) are submitted through a ***command-queue***.
- Each ***command-queue*** points to a single device within a context.



# Execution model (kernels)

- OpenCL execution model … define a problem domain and execute an instance of a **kernel** for each point in the domain

```
__kernel void times_two(
    __global float* input,
    __global float* output)
{
   int i = get_global_id(0);
   output[i] = 2.0f * input[i];
}
```



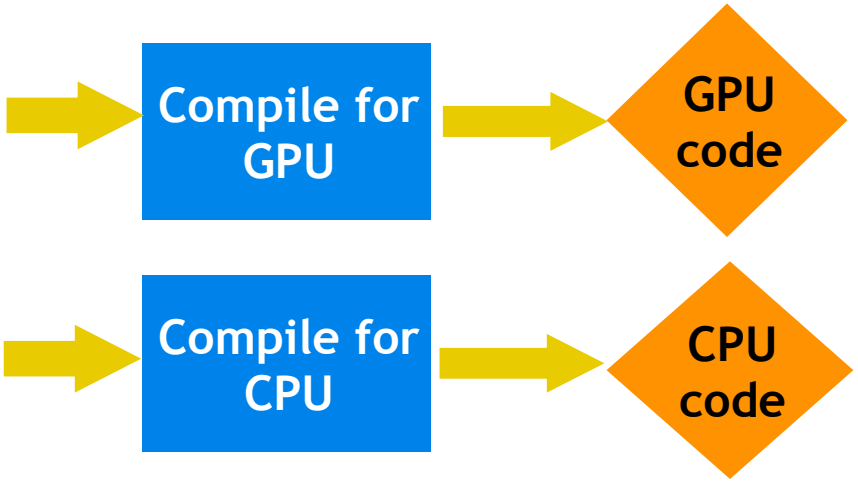
# Building Program Objects

- The program object encapsulates:
  - A context
  - The program source or binary, and
  - List of target devices and build options
- The build process to create a program object:

OpenCL uses **runtime compilation** … because in general you don't know the details of the target device when you ship the program

```
cl::Program program(context, KernelSource, true);
```

```
__kernel void
horizontal_reflect(read_only image2d_t src,
                   write_only image2d_t dst)
{
  int x = get_global_id(0);  // x-coord
  int y = get_global_id(1);  // y-coord
  int width = get_image_width(src);
  float4 src_val = read_imagef(src, sampler,
                               (int2)(width-1-x, y));
  write_imagef(dst, (int2)(x, y), src_val);
}
```



# Example: vector addition

- The “hello world” program of data parallel programming is a program to add two vectors

```
C[i] = A[i] + B[i] for i=0 to N-1
```

- For the OpenCL solution, there are two parts
  - Kernel code
  - Host code

# Vector Addition - Kernel

```
__kernel void vadd(
            __global const float *a,
            __global const float *b,
            __global       float *c)
{
    int gid = get_global_id(0);
    c[gid]  = a[gid] + b[gid];
}
```

# Exercise 1: Running the Vector Add kernel

- Goal:
  - To inspect and verify that you can run an OpenCL kernel
- Procedure:
  - Take the Vadd program we provide you. It will run a simple kernel to add two vectors together.
  - Look at the host code and identify the API calls in the host code. Compare them against the API descriptions on the OpenCL C++ reference card.
- Expected output:
  - A message verifying that the program completed successfully

**1. ssh -X train#@carver.nersc.gov (and enter supplied password)**
**2. ssh -X dirac# (and enter supplied password)**
**3. cp -r /projects/projectdirs/training/SC14/OpenCL_exercises/ .**
**4. module unload pgi openmpi cuda**
**5. module load gcc-sl6**
**6. module load openmpi-gcc-sl6**
**7. module load cuda**
**8. cd OpenCL_exercises**
**9. cp Make_def_files/dirac_linux_general.def make.def**
**8. cd /Exercises/Exercise01**
**9. make; ./vadd (etc)**
**More: https://www.nersc.gov/users/computational-systems/testbeds/dirac/opencl-tutorial-on-dirac/**

# UNDERSTANDING THE HOST PROGRAM

# Vector Addition – Host

- The host program is the code that runs on the host to:
  - Setup the environment for the OpenCL program
  - Create and manage kernels

- 5 simple steps in a basic host program:
  1. Define the ***platform*** … platform = devices+context+queues
  2. Create and Build the ***program*** (dynamic library for kernels)
  3. Setup ***memory*** objects
  4. Define the ***kernel*** (attach arguments to kernel function)
  5. Submit ***commands*** … transfer memory objects and execute kernels

Have a copy of the vadd host program on hand as we go over this set of slides.

# The C++ Interface

- Khronos has defined a common C++ header file containing a high level interface to OpenCL, **cl.hpp**
- This interface is dramatically easier to work with[1]
- Key features:
  - Uses common defaults for the platform and command-queue, saving the programmer from extra coding for the most common use cases
  - Simplifies the basic API by bundling key parameters with the objects rather than requiring verbose and repetitive argument lists
  - Ability to “call” a kernel from the host, like a regular function
  - Error checking can be performed with C++ exceptions

[1] especially for C++ programmers…

# C++ Interface: setting up the host program

- Enable OpenCL API Exceptions.

```
#define __CL_ENABLE_EXCEPTIONS
```

Do this before including the header files

- Include key header files … both standard and custom

```
#include <CL/cl.hpp>   // Khronos C++ Wrapper API
#include <cstdio>      // C style IO (e.g. printf)
#include <iostream>    // C++ style IO
#include <vector>      // C++ vector types
```

For information about C++, see the appendix "C++ for C programmers".

# 1. Create a context and queue

- Grab a context using a device type:

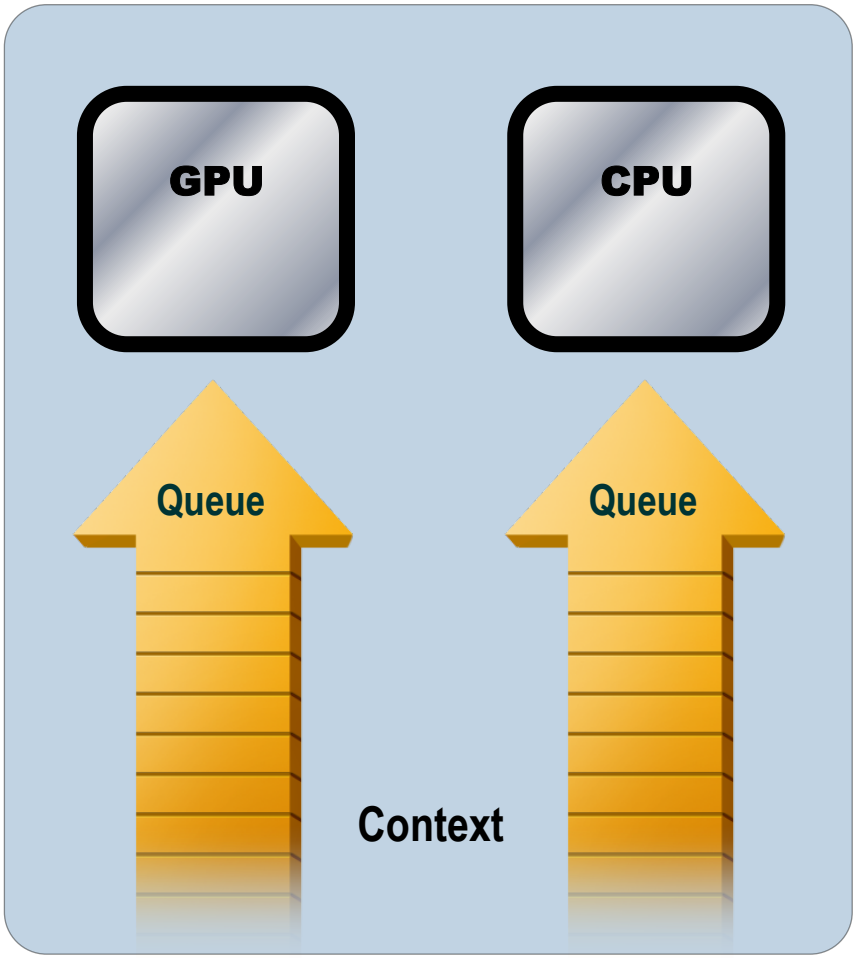
```
cl::Context context(CL_DEVICE_TYPE_DEFAULT);
```

- Create a command queue for the first device in the context:

```
cl::CommandQueue queue(context);
```

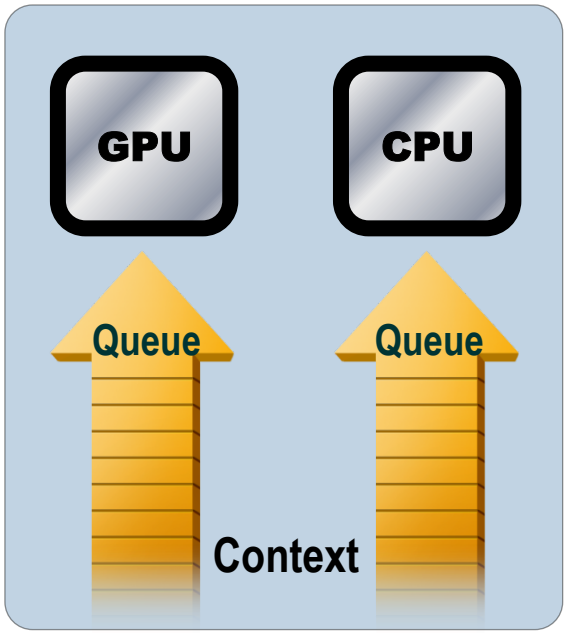
# Commands and Command-Queues

- Commands include:
  - Kernel executions
  - Memory object management
  - Synchronization
- The only way to submit commands to a device is through a command-queue.
- Each command-queue points to a single device within a context.
- Multiple command-queues can feed a single device.
  - Used to define independent streams of commands that don't require synchronization



# Command-Queue execution details

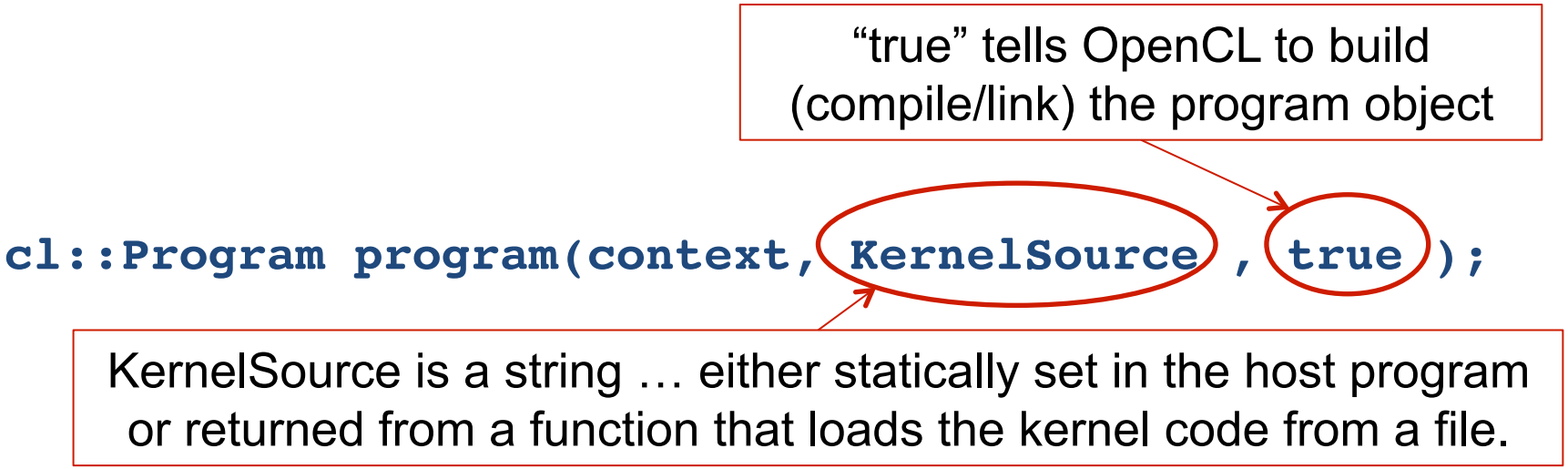
- *Command queues* can be configured in different ways to control how commands execute
- *In-order queues*:
  - Commands are enqueued and complete in the order they appear in the host program (program-order)
- *Out-of-order queues*:
  - Commands are enqueued in program-order but can execute (and hence complete) in any order.
- Execution of commands in the command-queue are guaranteed to be completed at synchronization points
  - Discussed later



# 2. Create and Build the program

- Define source code for the kernel-program either as a string literal (great for toy programs) or read it from a file (for real applications).

- Create the **program object and compile** to create a "dynamic library" from which specific kernels can be pulled:



# 3. Setup Memory Objects

- For vector addition we need 3 memory objects: one each for input vectors A and B, and one for the output vector C

- Create input vectors and assign values **on the host**:

```
std::vector<float> h_a(LENGTH), h_b(LENGTH), h_c(LENGTH);
for (i = 0; i < LENGTH; i++) {
    h_a[i] = rand() / (float)RAND_MAX;
    h_b[i] = rand() / (float)RAND_MAX;
}
```

- Define **OpenCL device buffers** and copy from **host buffers**:

```
cl::Buffer d_a(context, h_a.begin(), h_a.end(), true);
cl::Buffer d_b(context, h_b.begin(), h_b.end(), true);
cl::Buffer d_c(context, CL_MEM_WRITE_ONLY,
                    sizeof(float)*LENGTH);
```

or **CL_MEM_READ_ONLY** or **CL_MEM_READ_WRITE**

# What do we put in device memory?

- Memory Objects:
  - A handle to a reference-counted region of global memory.
- There are two kinds of memory object
  - ***Buffer*** object:
    - Defines a linear collection of bytes.
    - The contents of buffer objects are fully exposed within kernels and can be accessed using pointers
  - ***Image*** object:
    - Defines a two- or three-dimensional region of memory.
    - Image data can **only** be accessed with read and write functions, i.e. these are opaque data structures. The read functions use a sampler.

**Used when interfacing with a graphics API such as OpenGL. We won't use image objects in this tutorial.**

# Creating and manipulating buffers

- Buffers are declared on the host as object type:
  **`cl::Buffer`**

- Arrays in host memory hold your original host-side data:

```
std::vector<float> h_a, h_b;
```

- Create the device-side **buffer** (d_a), assign read-only memory to hold the host array (h_a) and copy it into device memory:

```
cl::Buffer  d_a(context, h_a.begin(), h_a.end(), true);
```

Start_iterator and end_iterator for the container holding host side object

Stipulates that this is a read-only buffer

# Creating and manipulating buffers

- The last argument sets the read/write access to the Buffer by the device. **true** means “read only” while **false** (the default) means “read/write”.
- Submit command to copy the device buffer back to host memory in array “h_c”:

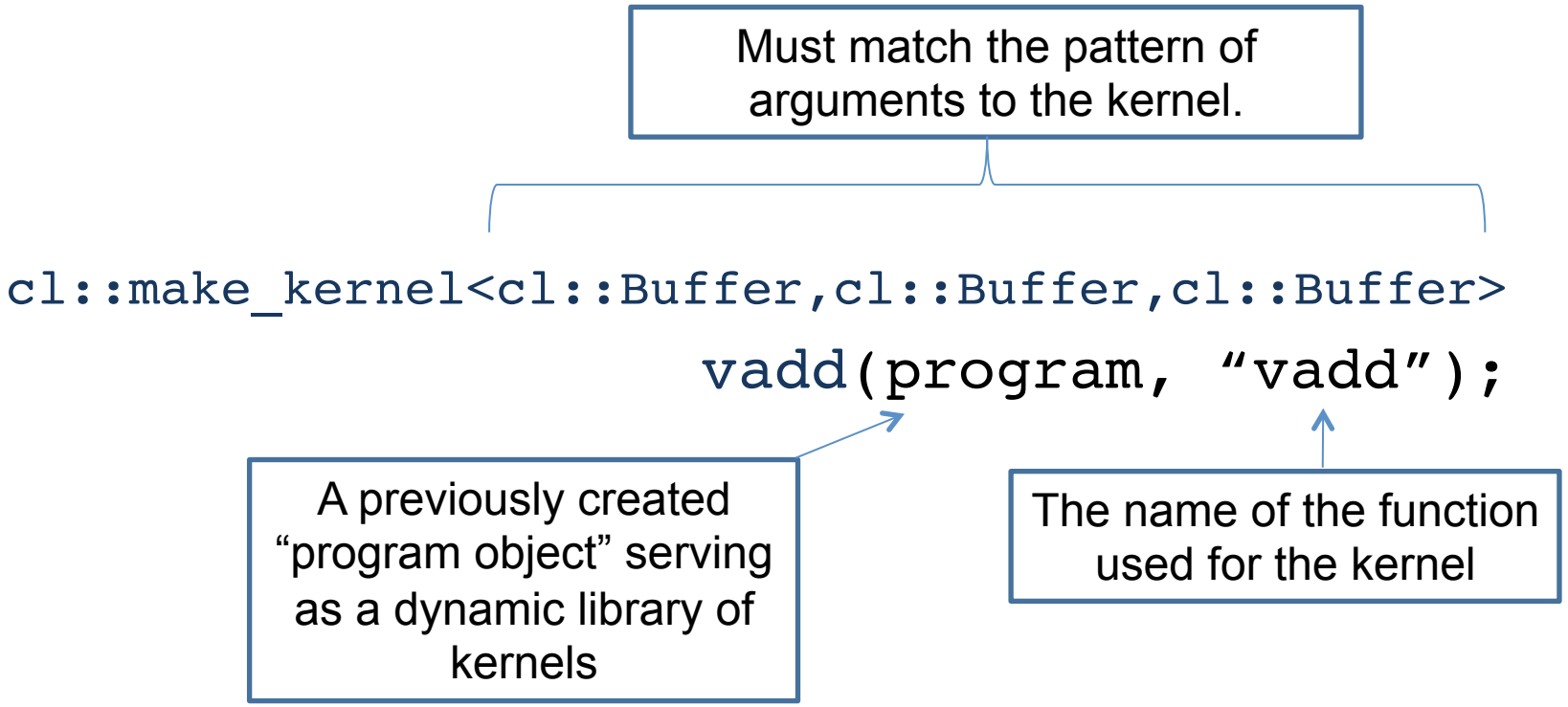
```
cl::copy(queue, d_c, h_c.begin(), h_c.end());
```

- Can also copy host memory to device buffers:

```
cl::copy(queue, h_c.begin(), h_c.end(), d_c);
```

# 4. Define the kernel

- Create a ***kernel functor*** for the kernels you want to be able to call in the **program**:




- This means you can 'call' the kernel as a 'function' in your host code to enqueue the kernel.

# 5. Enqueue commands

- For kernel launches, specify *global* and *local* dimensions
  - cl::NDRange global(1024)
  - cl::NDRange local(64)
  - If you don't specify a local dimension, it is assumed as cl::NullRange, and the runtime picks a size for you

- Enqueue the kernel for execution (note: returns immediately … i.e. this is a non-blocking command):

```
vadd(cl::EnqueueArgs(queue, global), d_a, d_b, d_c);
```

- Read back result (as a blocking operation). We use an in-order queue to assure the previous commands are completed before the read can begin

```
cl::copy(queue, d_c, h_c.begin(), h_c.end());
```

# C++ interface: The vadd host program

```
std::vector<float>
  h_a(N), h_b(N), h_c(N);
// initialize host vectors…

cl::Buffer d_a, d_b, d_c;

cl::Context  context(
   CL_DEVICE_TYPE_DEFAULT);

cl::CommandQueue
   queue(context);

cl::Program  program(
  context,
  loadprogram("vadd.cl"),
  true);

// Create the kernel functor
cl::make_kernel<cl::Buffer,
 cl::Buffer, cl::Buffer, int>
 vadd(program, "vadd");
```

```
// Create buffers
// True indicates CL_MEM_READ_ONLY
// False indicates CL_MEM_READ_WRITE

d_a = cl::Buffer(context,
        h_a.begin(), h_a.end(), true);

d_b = cl::Buffer(context,
        h_b.begin(), h_b.end(), true);

d_c = cl::Buffer(context,
        CL_MEM_READ_WRITE,
        sizeof(float) * LENGTH);

// Enqueue the kernel
vadd(cl::EnqueueArgs(
                queue,
                cl::NDRange(count)),
        d_a, d_b, d_c, count);

cl::copy(queue,
      d_c, h_c.begin(), h_c.end());
```

# Exercise 2: Chaining vector add kernels

- Goal:
  - To verify that you understand manipulating kernel invocations and buffers in OpenCL
- Procedure:
  - Start with your VADD program in C++
  - Add additional buffer objects and assign them to vectors defined on the host (see the provided vadd programs for examples of how to do this)
  - Chain vadds … e.g. C=A+B; D=C+E; F=D+G.
  - Read back the final result and verify that it is correct
- Expected output:
  - A message to standard output verifying that the chain of vector additions produced the correct result.

As you modify vadd to chain vadd kernels, you'll need to create additional buffers:

```
cl::Buffer(context, h_a.begin(), h_a.end(), true);
```

And enqueue additional kernels:

```
vadd(cl::EnqueueArgs(queue, cl::NDRange(count)),
     d_a, d_b, d_c, count);
```

# UNDERSTANDING THE KERNEL PROGRAM

# Working with Kernels (C++)

- The kernels are where all the action is in an OpenCL program.
- Steps to using kernels:
  1. Load kernel source code into a **program object** from a file
  2. Make a **kernel functor** from a function within the program
  3. Initialize **device memory**
  4. Call the **kernel functor**, specifying memory objects and global/local sizes
  5. Read **results** back from the device
- Note the kernel function argument list must match the kernel definition on the host.

# Create a kernel

- Kernel code:
  - A string in the host code (“toy codes”).
  - Loaded from a file as a string or binary.

- Compile for the default devices within the default Context

```
program.build();
```

**The build step can be carried out by specifying *true* in the program constructor. If you need to specify build flags you must specify *false* in the constructor and use this method instead.**

- Define the kernel functor from a function within the program – allows us to ‘call’ the kernel to enqueue it as if it were just another function

```
cl::make_kernel<cl::Buffer, cl::Buffer, cl::Buffer, int>
                vadd(program, “vadd”);
```

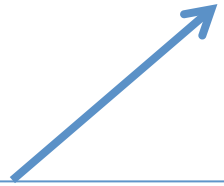
# Advanced: get info about the kernel

- Advanced: if you want to query information about a kernel, you will need to create a kernel object:

```
cl::Kernel ko_vadd(program, "vadd");
```

- For example, to query the default size of the local dimension (i.e. the size of a Work-Group)

```
::size_t work_group_size =
            ko_vadd.getWorkGroupInfo
            <CL_KERNEL_WORK_GROUP_SIZE>
            (cl::Device::getDefault());
```



**We can use any work-group-info parameter from table 5.15 in the OpenCL 1.2 specification. The function will return the appropriate type.**

# Call (enqueue) the kernel

- Enqueue the kernel for execution with buffer objects `d_a`, `d_b` and `d_c` and their length, `count`:

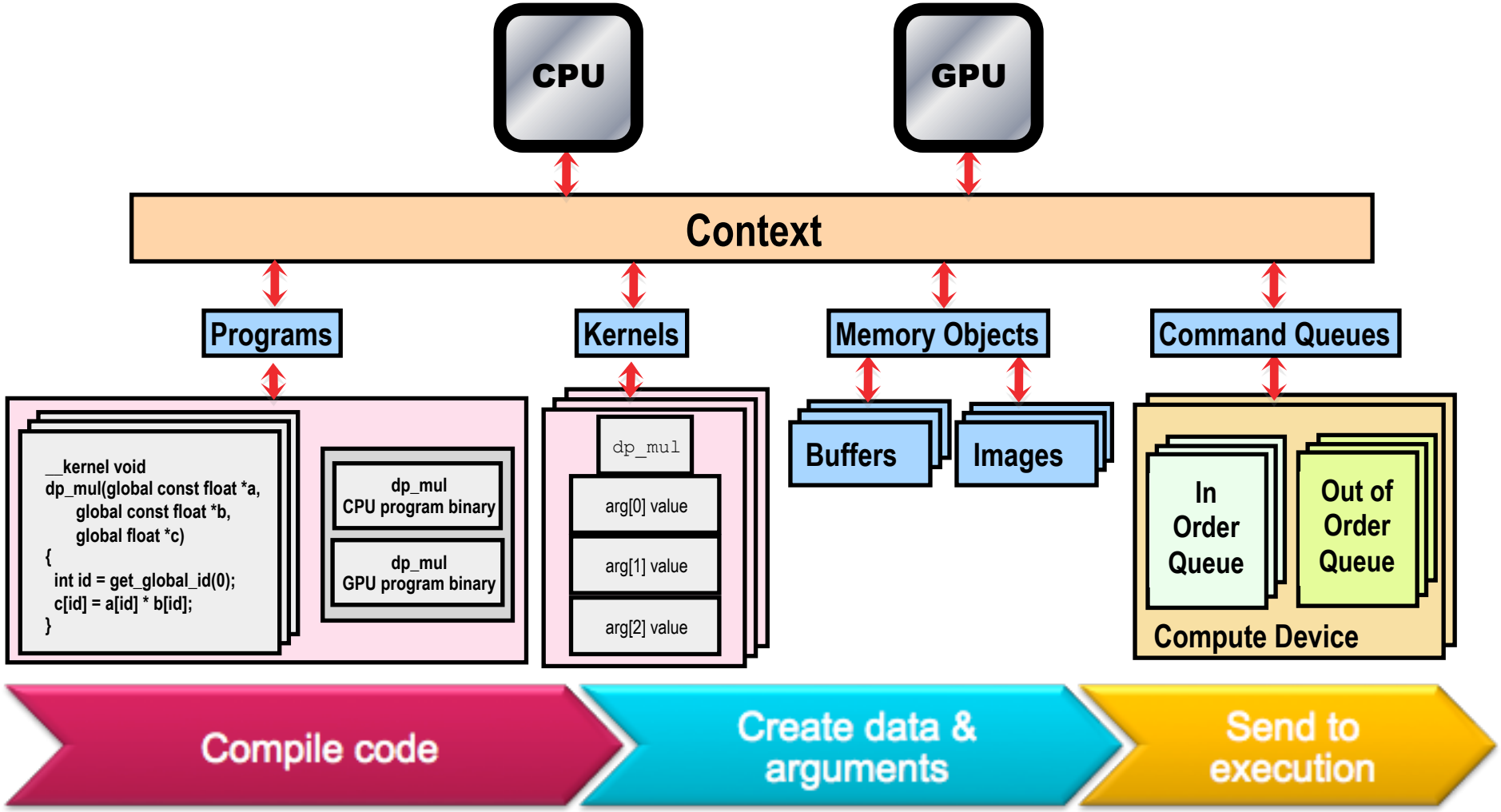
```
vadd( cl::EnqueueArgs(queue,
                      cl::NDRange(count),
                      cl::NDRange(local)),
      d_a, d_b, d_c, count );
```

**We can include any arguments from the clEnqueueNDRangeKernel function including Event wait lists and the command queue options**

# Exercise 3: The D = A + B + C problem

- Goal:
  - To verify that you understand how to control the argument definitions for a kernel.
  - To verify that you understand the host/kernel interface.
- Procedure:
  - Start with your VADD program.
  - Modify the kernel so it adds three vectors together.
  - Modify the host code to define three vectors and associate them with relevant kernel arguments.
  - Read back the final result and verify that it is correct.
- Expected output:
  - Test your result and verify that it is correct. Print a message to that effect on the screen.

# We have now covered the basic platform runtime APIs in OpenCL



# INTRODUCTION TO OPENCL KERNEL PROGRAMMING

# OpenCL C kernel language

- Derived from **ISO C99**
  - A few ***restrictions***: no recursion, function pointers, functions in C99 standard headers ...
  - Preprocessing directives defined by C99 are supported (#include etc.)

- Built-in data types
  - Scalar and vector data types, pointers
  - Data-type conversion functions:
    - convert_type<_sat><_roundingmode>
  - Image types: image2d_t, image3d_t and sampler_t

# Vector Types

- The OpenCL C kernel programming language provides a set of vector instructions:
  - These are portable between different vector instruction sets
- These instructions support vector lengths of 2, 4, 8, and 16 … for example:
  - **`char2, ushort4, int8, float16, …`**

- Vector literal

```
int4 vi0 = (int4) (2, 3, -7, -7);
```

| 2 | 3 | -7 | -7 |
|---|---|---|---|

```
int4 vi1 = (int4) (0, 1, 2, 3);
```

| 0 | 1 | 2 | 3 |
|---|---|---|---|

# Vector Types

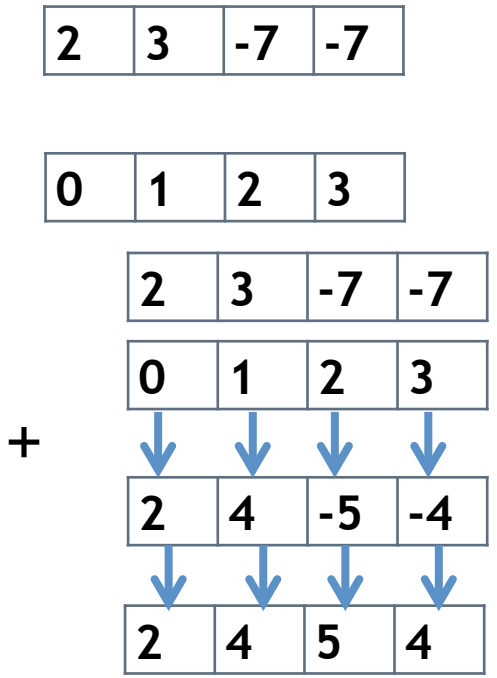
- The OpenCL C kernel programming language provides a set of vector instructions:
  - These are portable between different vector instruction sets
- These instructions support vector lengths of 2, 4, 8, and 16 … for example:
  - **`char2, ushort4, int8, float16, …`**

- Vector literal

```
int4 vi0 = (int4) (2, 3, -7, -7);

int4 vi1 = (int4) (0, 1, 2, 3);
```

- Vector ops

```
vi0 += vi1;

vi0 = abs(vi0);
```



# data conversions

- Data-type conversion functions:
  - convert_type[_sat][_roundingmode]

```
float4 f;   // a vector of 4 floats

//truncate (rtz) floats to generate ints.  Results
//implementation defined for f > INT_MAX, NaN etc
int4 i1 = convert_int4(f);

// Same as above (rtz) but for values > INT_MAX clamp
// to INT_MAX, values < INT_MIN clamp to INT_MIN.
// NaN → 0.
int4 i2 = convert_int4_sat( f );

// round the floats to the nearest integer
int4 i3 = convert_int4_rte( f );
```

# OpenCL C Language Highlights

- Function qualifiers
  - **__kernel** qualifier declares a function as a kernel
    - I.e. makes it visible to host code so it can be enqueued
  - Kernels can call other kernel-side functions
- Address space qualifiers
  - **__global, __local, __constant, __private**
  - Pointer kernel arguments must be declared with an address space qualifier
- Work-item functions
  - **uint get_work_dim()  … number of dimensions in use (1,2, or 3)**
  - **size_t get_global_id(uint n) … global work-item ID in dim "n"**
  - **size_t get_local_id(uint n)  …  work-item ID in dim "n" inside work-group**
  - **size_t get_group_id(uint n) … ID of work-group in dim "n"**
  - **size_t get_global_size(uint n) … num of work-items in dim "n"**
  - **size_t get_local_size(uint n) … num of work-items in work group in dim "n"**
- Synchronization functions
  - **Barriers** - all work-items within a work-group must execute the barrier function before any work-item can continue
  - **Memory fences** - provides ordering between memory operations

# OpenCL C Language Restrictions

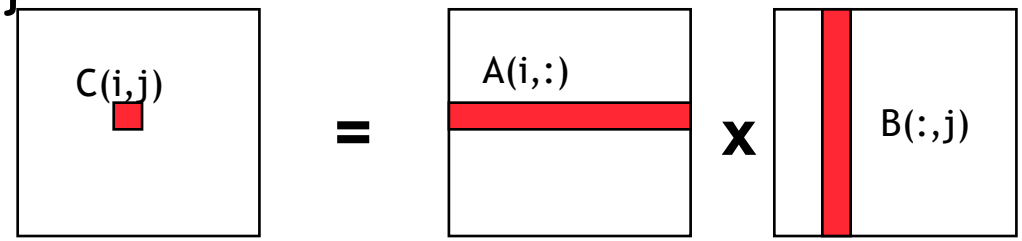
- Pointers to functions are ***not*** allowed
- Pointers to pointers allowed ***within*** a kernel, but not as an argument to a kernel invocation
- Bit-fields are not supported
- Variable length arrays and structures are not supported
- Recursion is not supported (yet!)
- Double types are optional in OpenCL v1.2, but the key word is reserved

  (note: most implementations support double)

# Matrix multiplication: sequential code

We calculate C=AB, dimA = (N x P), dimB=(P x M), dimC=(N x M)

```
void mat_mul(int Mdim, int Ndim, int Pdim,
                        float *A, float *B, float *C)
{
    int i, j, k;
    for (i = 0; i < Ndim; i++) {
        for (j = 0; j < Mdim; j++) {
            C[i*N+j] = 0.0f;
            for (k = 0; k < Pdim; k++) {
                // C(i, j) = sum(over k) A(i,k) * B(k,j)
                C[i*Ndim+j] += A[i*Ndim+k] * B[k*Pdim+j];
            }
        }
    }
}
```



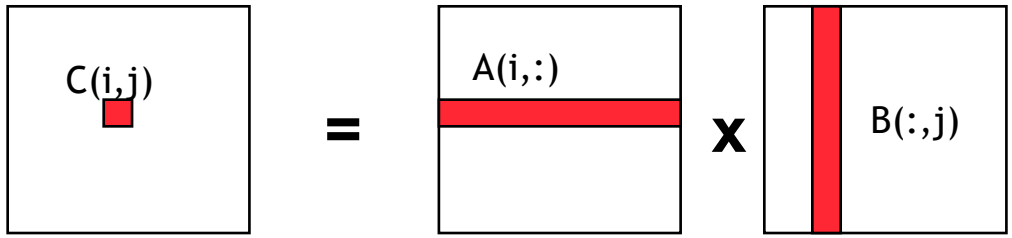

Dot product of a row of A and a column of B for each element of C

# Matrix multiplication: sequential code

We calculate C=AB, where all three matrices are NxN

```
void mat_mul(int N, float *A, float *B, float *C)
{
    int i, j, k;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            C[i*N+j] = 0.0f;
            for (k = 0; k < N; k++) {
                // C(i, j) = sum(over k) A(i,k) * B(k,j)
                C[i*N+j] += A[i*N+k] * B[k*N+j];
            }
        }
    }
}
```

Let's make it easier and specialize to square matrices




**Dot product of a row of A and a column of B for each element of C**

# Matrix multiplication performance

- Serial C code on CPU (single core).

| Case | MFLOPS | |
|---|---|---|
| | CPU | GPU |
| Sequential C (not OpenCL) | 887.2 | N/A |

Device is Intel® Xeon® CPU, E5649 @ 2.53GHz
using the gcc compiler.

**These are not official benchmark results. You may observe completely different results should you run these tests on your own system.**

# Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
    int i, j, k;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            C[i*N+j] = 0.0f;
            for (k = 0; k < N; k++) {
              // C(i, j) = sum(over k) A(i,k) * B(k,j)
              C[i*N+j] += A[i*N+k] * B[k*N+j];
            }
        }
    }
}
```

# Exercise 4: Write a matrix multiply kernel

- Goal:
  - To verify that you understand how to convert an array based serial code into an OpenCL kernel.
- Procedure:
  - Start with the provided serial matrix multiply code.
  - Copy it into a file (mmul.cl) and convert it into a kernel callable from OpenCL where each work-item computes an element of the result matrix
- Expected output:
  - None … we just want you to write the code and we'll discuss the results as a group.

Hints:
__kernel to mark your kernel which much be a void function
__global to mark global memory objects

```
int gid = get_global_id(DIM);  //DIM is 0, 1 or 2 for
                               //dimension in NDRange
```

# Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)

{
    int i, j, k;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            C[i*N+j] = 0.0f;
            for (k = 0; k < N; k++) {
              // C(i, j) = sum(over k) A(i,k) * B(k,j)
              C[i*N+j] += A[i*N+k] * B[k*N+j];
            }
        }
    }
}
```

# Matrix multiplication: OpenCL kernel (1/2)

```
__kernel void mat_mul(const int N, __global float *A,
                      __global float *B, __global float
{
    int i, j, k;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            C[i*N+j] = 0.0f;
            for (k = 0; k < N; k++) {
              // C(i, j) = sum(over k) A(i,k) * B(k,j)
              C[i*N+j] += A[i*N+k] * B[k*N+j];
            }
        }
    }
}
```

Mark as a kernel function and specify memory qualifiers

# Matrix multiplication: OpenCL kernel (2/2)

```
__kernel void mat_mul(const int N, __global float *A,
                      __global float *B, __global float
{
    int i, j, k;
    i = get_global_id(0);
        j = get_global_id(1);
            C[i*N+j] = 0.0f;
            for (k = 0; k < N; k++) {
              // C(i, j) = sum(over k) A(i,k) * B(k,j)
              C[i*N+j] += A[i*N+k] * B[k*N+j];
            }
        }
    }
}
```

Replace loops with the work item's global id

# Matrix multiplication: kernel cleaned-up

Rearrange and use a local scalar for intermediate C element values (a common optimization in Matrix Multiplication functions)

```
__kernel void mmul(
    const int N,
    __global float *A,
    __global float *B,
    __global float *C)
{
  int k;
  int i = get_global_id(0);
  int j = get_global_id(1);
  float tmp = 0.0f;
  for (k = 0; k < N; k++)
   tmp += A[i*N+k]*B[k*N+j];

  C[i*N+j] = tmp;
}
```

# Building programs

- To use a kernel, you must:
  1. build the program object containing the kernel
     - Specify the kernel source in the program object constructor.
     - We provide a utility function to load the source from a file (or you can specify the source as a string)

```
cl::Program program(context, util::loadProgram("matmul1.cl"));
```

  2. Compile the program object
     - You can compile the program with the constructor by specifying the last argument as “true”:

```
cl::Program program(context, util::loadProgram("matmul1.cl"),true);
```

How do you recover compiler messages should there be compile-time errors in your kernel?

# Compiler error messages (1/2)

- You need to use an explicit build step so you can recover the error and query the system to fetch the compiler messages.
  1. Do NOT build the program with the constructor (i.e. do not set the last argument in the program constructor to true).

     **cl::Program program(context, util::loadProgram("matmul1.cl"));**
  2. Explicitly build the program within a try block to catch the error exception:

```
try
    {
       program.build();
    }
    catch (cl::Error error)
    {
      // If it was a build error then show the error
      if (error.err() == CL_BUILD_PROGRAM_FAILURE)
       {
          ///   recover compiler message (see the next slide)
       }
       throw error;
    }
```

# Compiler error messages (2/2)

- Compiled code is connected to the device … so we need a handle to reference the device

```
std::vector<cl::Device> devices;
devices = context.getInfo<CL_CONTEXT_DEVICES>();
```

- In our programming, we've been using the default device which is the first one (devices[0]).
- We need to query the program object to get the "BuildInfo"

```
std::string built =  program.getBuildInfo
        <CL_PROGRAM_BUILD_LOG>(devices[0]);
```

- Now we can output the error compiler message

```
std::cerr << built << "\n";
```

# Compiler error messages: Complete Example

```
cl::Program program(context, util::loadProgram("matmul1.cl"));
 try
 {
    program.build();
 }
 catch (cl::Error error)
 {
   // If it was a build error then show the error
   if (error.err() == CL_BUILD_PROGRAM_FAILURE)
    {
       std::vector<cl::Device> devices;
       devices = context.getInfo<CL_CONTEXT_DEVICES>();
       std::string built =  program.getBuildInfo
                            <CL_PROGRAM_BUILD_LOG>(devices[0]);
       std::cerr << built << "\n";
    }
    throw error;
 }
```

# Exercise 5: The C = A * B problem

- Goal:
  - To verify that you understand how to write a host program.
- Procedure:
  - Using the VADD host program and serial matrix multiply programs as your guide, write a host program to call your matrix multiplication kernel.
  - Copy the result matrix back to the host and verify output (using the functions we provided in the serial program).
- Expected output:
  - Test your result and verify that it is correct. Output the runtime and the MFLOPS.

# Matrix multiplication host program

```
#define DEVICE CL_DEVICE_TYPE_DEFAULT

    // declarations (not shown)
    sz = N * N;
    std::vector<float> h_A(sz);
    std::vector<float> h_B(sz);
    std::vector<float> h_C(sz);

    cl::Buffer d_A, d_B, d_C;

    // initialize matrices  and setup
    // the problem (not shown)

    cl::Context context(DEVICE);
    cl::Program program(context,
        util::loadProgram("matmul1.cl"),
        true);

    cl::CommandQueue queue(context);

    cl::make_kernel
            <int, cl::Buffer, cl::Buffer, cl::Buffer>
            mmul(program, "mmul");

    d_A = cl::Buffer(context,
                        h_A.begin(), h_A.end(), true);
    d_B = cl::Buffer(context,
                        h_B.begin(), h_B.end(), true);
    d_C = cl::Buffer(context,
                        CL_MEM_WRITE_ONLY,
                        sizeof(float) * sz);

    mmul(
      cl::EnqueueArgs(queue, cl::NDRange(N,N)),
      N, d_A,  d_B,  d_C );

    cl::copy(queue, d_C, h_C.begin(), h_C.end());

    // Timing and check results (not shown)
```

# Matrix multiplication performance

- Matrices are stored in global memory.

| Case | MFLOPS | |
|---|---|---|
| | CPU | GPU |
| Sequential C (not OpenCL) | 887.2 | N/A |
| C(i,j) per work-item, all global | 3,926.1 | 3,720.9 |

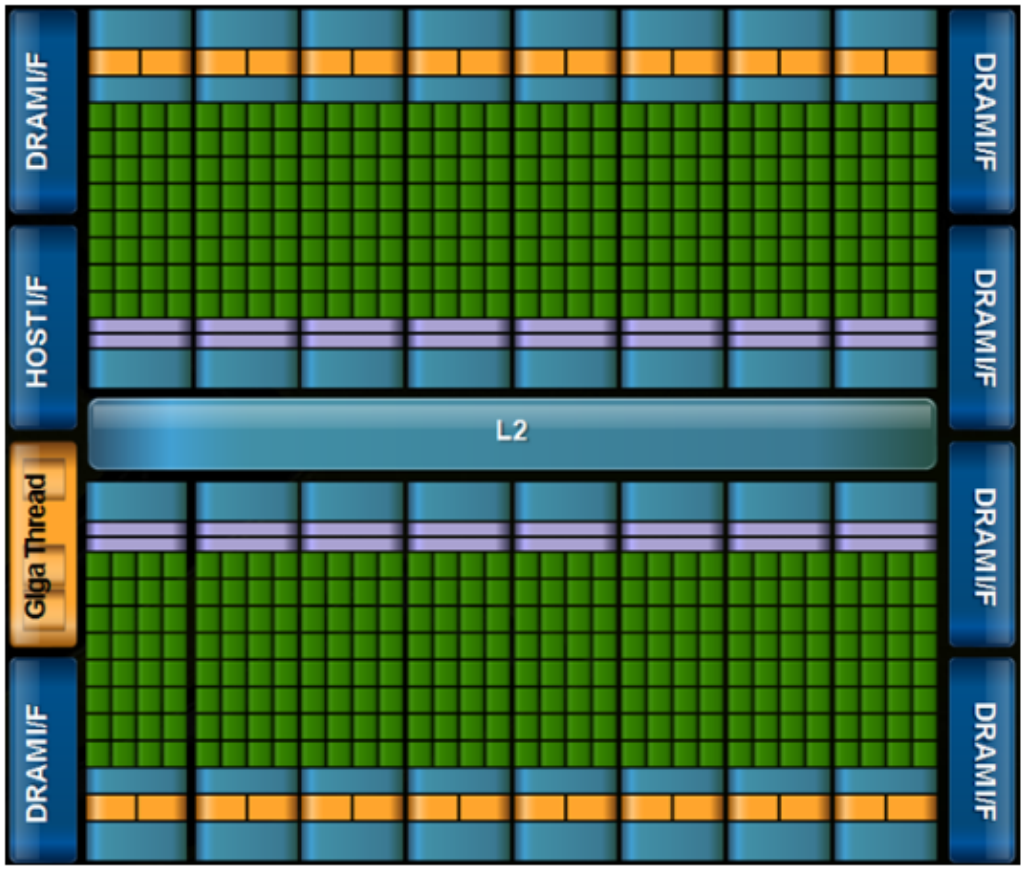
Device is Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs
Device is Intel® Xeon® CPU, E5649 @ 2.53GHz

**These are not official benchmark results. You may observe completely different results should you run these tests on your own system.**

# Dirac Node Configuration

Dirac is a 50 GPU node cluster connected with QDR IB. Each GPU node also contains 2 Intel 5530 2.4 GHz, 8MB cache, 5.86GT/sec QPI Quad core Nehalem processors (8 cores per node) and 24GB DDR3-1066 Reg ECC memory.

- 44 nodes: 1 NVIDIA Tesla C2050 (Fermi) GPU with 3GB of memory and 448 parallel CUDA processor cores.
- 4 nodes: 1 C1060 NVIDIA Tesla GPU with 4GB of memory and 240 parallel CUDA processor cores.
- 1 node: 4 NVIDIA Tesla C2050 (Fermi) GPU's, each with 3GB of memory and 448 parallel CUDA processor cores.
- 1 node: 4 C1060 Nvidia Tesla GPU's, each with 4GB of memory and 240 parallel CUDA processor cores.

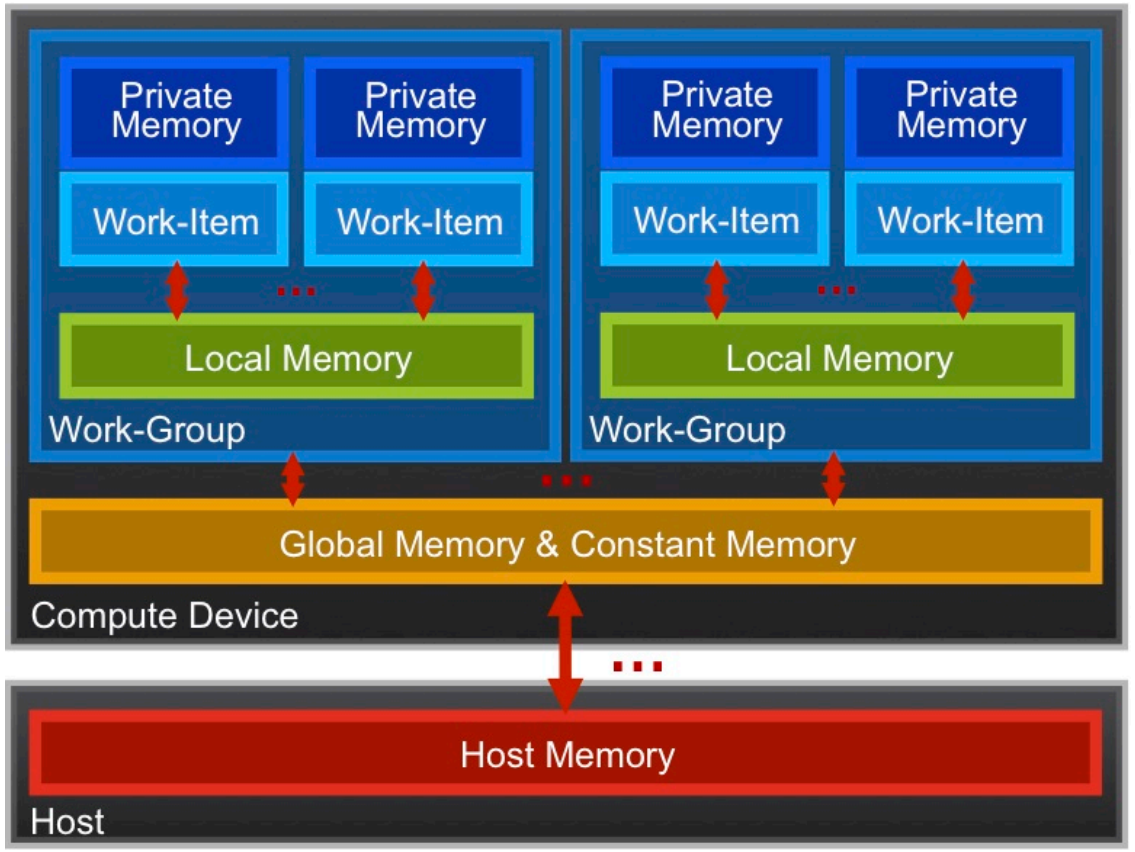



Source: http://www.nersc.gov/users/computational-systems/dirac/node-and-gpu-configuration/

# UNDERSTANDING THE OPENCL MEMORY HIERARCHY

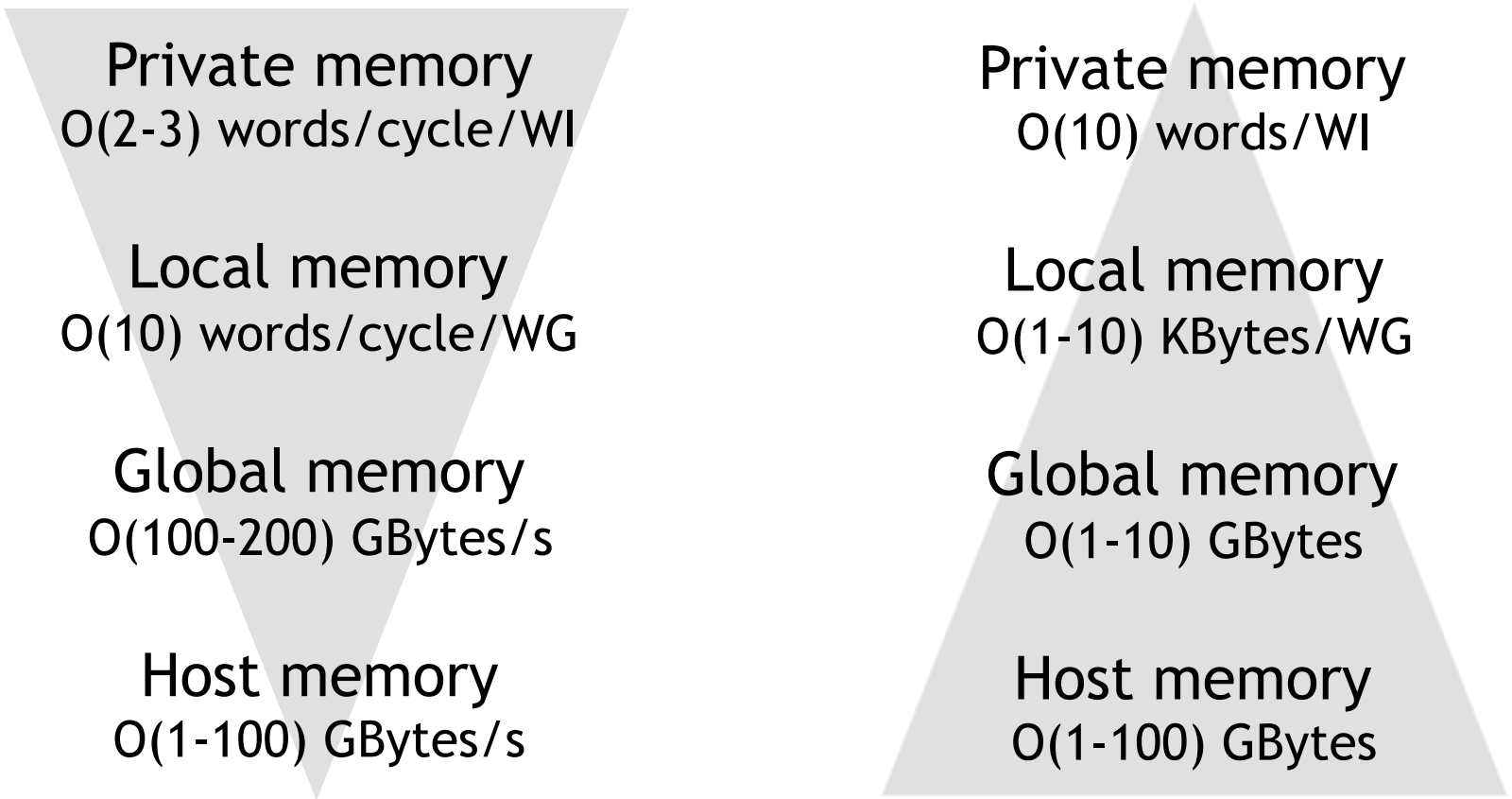
# OpenCL Memory model

- Private Memory
  - Per work-item
- Local Memory
  - Shared within a work-group
- Global/Constant Memory
  - Visible to all work-groups
- Host memory
  - On the CPU




Memory management is **explicit**:
You are responsible for moving data from
host → global → local *and* back

# The Memory Hierarchy




Managing the memory hierarchy is one of ***the*** most important things to get right to achieve good performance

*Size and performance numbers are approximate and for a high-end discrete GPU, circa 2011

# Optimizing matrix multiplication

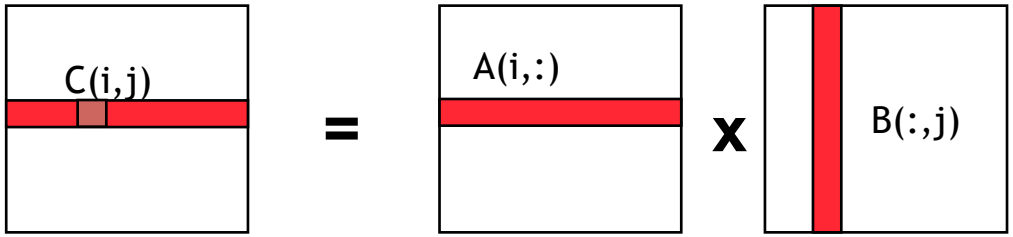
- MM cost determined by FLOPS and memory movement:
  - $2*n^3 = O(n^3)$ FLOPS
  - Operates on $3*n^2 = O(n^2)$ numbers
- To optimize matrix multiplication, we must ensure that for every memory access we execute as many FLOPS as possible.
- Outer product algorithms are faster, but for pedagogical reasons, let's stick to the simple dot-product algorithm.


Dot product of a row of A and a column of B for each element of C

- We will work with work-item/work-group sizes and the memory model to optimize matrix multiplication

# Optimizing matrix multiplication

- There may be significant overhead to manage work-items and work-groups.
- So let's have each work-item compute a full row of C




**Dot product of a row of A and a column of B for each element of C**

- And with an eye towards future optimizations, let's collect work-items into work-groups with 64 work-items per work-group

# Exercise 6: C = A * B (1 row per work-item)

- Goal:
  - To give you experience managing the number of work-items per work-group.
- Procedure:
  - Start from you last matrix multiplication program. Modify it so each work-item handles an entire row of the matrix.
- Expected output:
  - Test your result and verify that it is correct. Output the runtime and the MFLOPS.

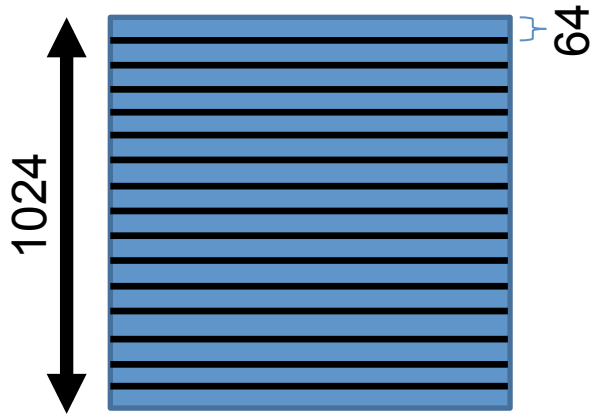
cl::EnqueueArgs() is used with the kernel functor to control how a kernel is enqueued. There are many overloaded forms … the one you'll need is:

cl::EnqueueArgs(NDRange Global, NDRange Local)

Where "global" and "local" are (N), (N,N), or (N,N,N) depending on the dimensionality of the NDRange index space.

# An N-dimension domain of work-items

- Global Dimensions: 1024 (1D)
  Whole problem space (index space)
- Local Dimensions:  64 (work-items per work-group)
  Only 1024/64 = 16 work-groups in total




- Important implication: we will have a lot fewer work-items per work-group (64) and work-groups (16). Why might this matter?

# Matrix multiplication: One work item per row of C

```
__kernel void mmul(
    const int N,
    __global float *A,
    __global float *B,
    __global float *C)
{
   int j, k;
   int i = get_global_id(0);
   float tmp;
   for (j = 0; j < N; j++) {
    tmp = 0.0f;
    for (k = 0; k < N; k++)
      tmp += A[i*N+k]*B[k*N+j];
    C[i*N+j] = tmp;
   }
}
```

# Mat. Mul. host program (1 row per work-item)

```
#define DEVICE CL_DEVICE_TYPE_DEFAULT

  // declarations (not shown)
  sz = N * N;
  std::vector<float> h_A(sz);
  std::vector<float> h_B(sz);
  std::vector<float> h_C(sz);

  cl::Buffer d_A, d_B, d_C;

  // initialize matrices  and setup
  // the problem (not shown)

  cl::Context context(DEVICE);
  cl::Program program(context,
    util::loadProgram("mmulCrow.cl"),
        true);
```

```
cl::CommandQueue queue(context);

cl::make_kernel
        <int, cl::Buffer, cl::Buffer, cl::Buffer>
        mmul(program, "mmul");

d_A = cl::Buffer(context,
                    h_A.begin(), h_A.end(), true);
d_B = cl::Buffer(context,
                    h_B.begin(), h_B.end(), true);
d_C = cl::Buffer(context,
                    CL_MEM_WRITE_ONLY,
                    sizeof(float) * sz);

mmul( cl::EnqueueArgs(
          queue, cl::NDRange(N), cl::NdRange(64) ),
        N, d_A,  d_B,  d_C );

cl::copy(queue, d_C, h_C.begin(), h_C.end());

// Timing and check results (not shown)
```

# Mat. Mul. host program (1 row per work-item)

```
#define DEVICE CL_DEVICE_TYPE_DEFAULT

// declarations (not shown)
sz = N * N;
std::vector<float> h_A(sz);
std::vector<float> h_B(sz);
std::vector<float> h_C(sz);
```

Changes to host program:

1. 1D ND Range set to number of rows in the C matrix
2. Local Dimension set to 64 (which gives us 16 work-groups which matches the GPU's number of compute units).

```
cl::CommandQueue queue(context);

cl::make_kernel
        <int, cl::Buffer, cl::Buffer, cl::Buffer>
        mmul(program, "mmul");

d_A = cl::Buffer(context,
                 h_A.begin(), h_A.end(), true);
d_B = cl::Buffer(context,
                 h_B.begin(), h_B.end(), true);
d_C = cl::Buffer(context,
                 CL_MEM_WRITE_ONLY,
                 sizeof(float) * sz);

mmul( cl::EnqueueArgs(
          queue, cl::NDRange(N), cl::NdRange(64) ),
        N, d_A,  d_B,  d_C );

cl::copy(queue, d_C, h_C.begin(), h_C.end());

// Timing and check results (not shown)
```

# Matrix multiplication performance

- Matrices are stored in global memory.

| Case | MFLOPS | |
|---|---|---|
| | CPU | GPU |
| Sequential C (not OpenCL) | 887.2 | N/A |
| C(i,j) per work-item, all global | 3,926.1 | 3,720.9 |
| C row per work-item, all global | 3,379.5 | 4,195.8 |

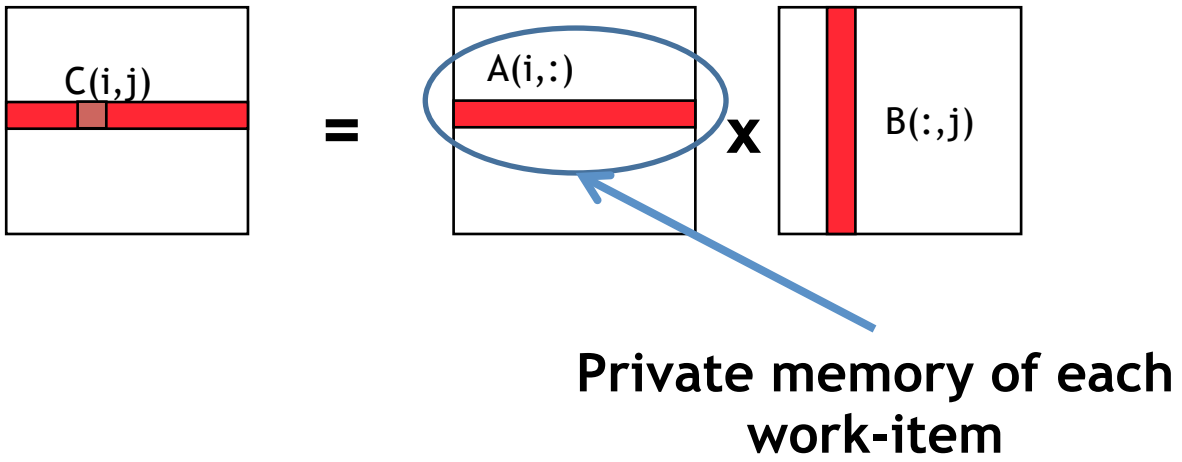
This has started to help.

Device is Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs
Device is Intel® Xeon® CPU, E5649 @ 2.53GHz

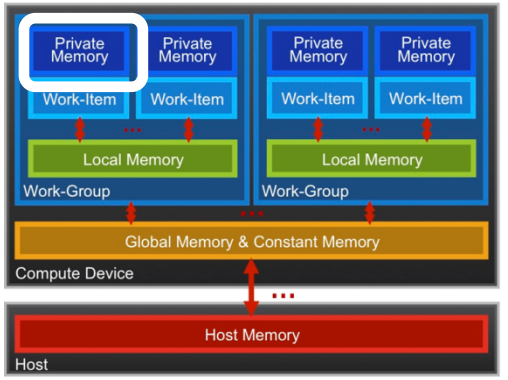
These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

# Optimizing matrix multiplication

- Notice that, in one row of C, each element reuses the same row of A.
- Let's copy that row of A into private memory of the work-item that's (exclusively) using it to avoid the overhead of loading it from global memory for each C(i,j) computation.



# Private Memory




- A work-items private memory:
  - A **very scarce** resource, only a few tens of 32-bit words per Work-Item at most (on a GPU)
  - If you use too much it spills to global memory or reduces the number of Work-Items that can be run at the same time, potentially harming performance*
  - Think of these like registers on the CPU
- How do you create and manage private memory?
  - Declare statically inside your kernel

* Occupancy on a GPU

# Exercise 7: C = A * B (Row of A in private memory)

- Goal:
  - To give you experience working with private memory.
- Procedure:
  - Start from you last matrix multiplication program (the row-based method). Modify it so each work item copies A from global to private memory to reduce traffic into global memory.
- Expected output:
  - Test your result and verify that it is correct. Output the runtime and the MFLOPS.

Private memory can be allocated as an automatic (i.e. not with malloc) inside a kernel … so just declare any arrays you need. You can use normal loads and stores inside a kernel to move data between private and global address spaces.

# Matrix multiplication: (Row of A in private memory)

```
__kernel void mmul(
    const int N,
    __global float *A,
    __global float *B,
    __global float *C)
{
  int j, k;
  int i =
    get_global_id(0);
  float tmp;
  float Awrk[1024];

  for (k = 0; k < N; k++)
    Awrk[k] = A[i*N+k];

  for (j = 0; j < N; j++) {
    tmp = 0.0f;
    for (k = 0; k < N; k++)
      tmp += Awrk[k]*B[k*N+j];

    C[i*N+j] += tmp;
  }
}
```

# Matrix multiplication: (Row of A in private memory)

Copy a row of A into private memory from global memory before we start with the matrix multiplications.

```
__kernel void mmul(
    const int N,
    __global float *A,
    __global float *B,
    __global float *C)
{
  int j, k;
  int i =
    get_global_id(0);
  float tmp;
  float Awrk[1024];

  for (k = 0; k < N; k++)
    Awrk[k] = A[i*N+k];

  for (j = 0; j < N; j++) {
    tmp = 0.0f;
    for (k = 0; k < N; k++)
      tmp += Awrk[k]*B[k*N+j];

    C[i*N+j] += tmp;
  }
}
```

Setup a work array for A in private memory*

(*Actually, this is using ***far*** more private memory than we'll have and so Awrk[] will be spilled to global memory)

# Mat. Mul. host program (Row of A in private memory)

```
#define DEVICE CL_DEVICE_TYPE_DEFAULT

// declarations (not shown)
sz = N * N;
std::vector<float> h_A(sz);
std::vector<float> h_B(sz);
std::vector<float> h_C(sz);

cl::Buffer d_A, d_B, d_C;

// initialize matrices  and setup
// the problem (not shown)

cl::Context context(DEVICE);
cl::Program program(context,
  util::loadProgram("mmulCrow.cl"),
     true);
```

```
cl::CommandQueue queue(context);

cl::make_kernel
        <int, cl::Buffer, cl::Buffer, cl::Buffer>
        mmul(program, "mmul");

d_A = cl::Buffer(context,
                 h_A.begin(), h_A.end(), true);
d_B = cl::Buffer(context,
                 h_B.begin(), h_B.end(), true);
d_C = cl::Buffer(context,
                 CL_MEM_WRITE_ONLY,
                 sizeof(float) * sz);

mmul( cl::EnqueueArgs(
          queue, cl::NDRange(N), cl::NdRange(64) ),
        N, d_A,  d_B,  d_C );

cl::copy(queue, d_C, h_C.begin(), h_C.end());
```

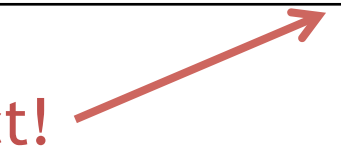
**Host program unchanged from last exercise**

# Matrix multiplication performance

- Matrices are stored in global memory.

| Case | MFLOPS | |
|---|---|---|
| | CPU | GPU |
| Sequential C (not OpenCL) | 887.2 | N/A |
| C(i,j) per work-item, all global | 3,926.1 | 3,720.9 |
| C row per work-item, all global | 3,379.5 | 4,195.8 |
| C row per work-item, A row private | 3,385.8 | 8,584.3 |

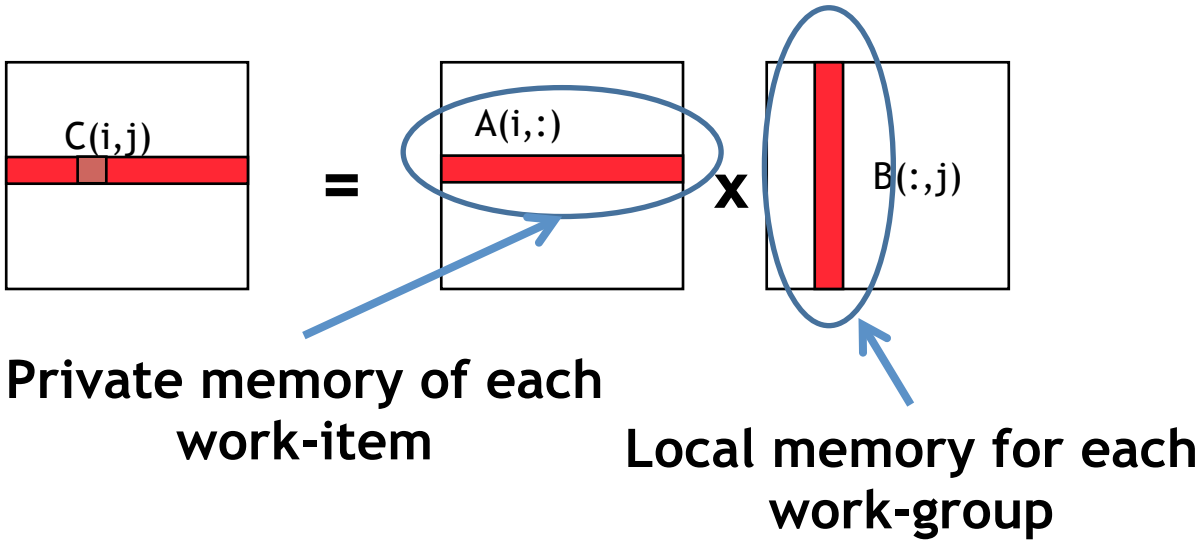
Device is Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs
Device is Intel® Xeon® CPU, E5649 @ 2.53GHz



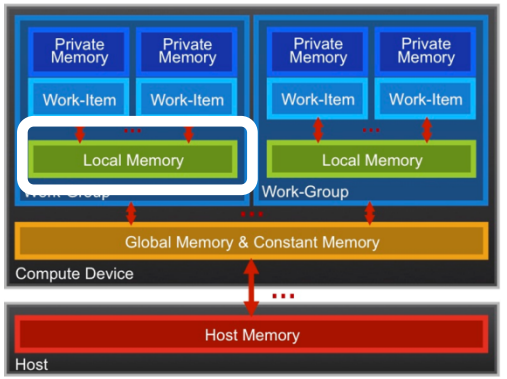

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

# Optimizing matrix multiplication

- We already noticed that, in one row of C, each element uses the same row of A
- Each work-item in a work-group also uses the same columns of B
- So let's store the B columns in local memory (which is shared by the work-items in the work-group)



# Local Memory

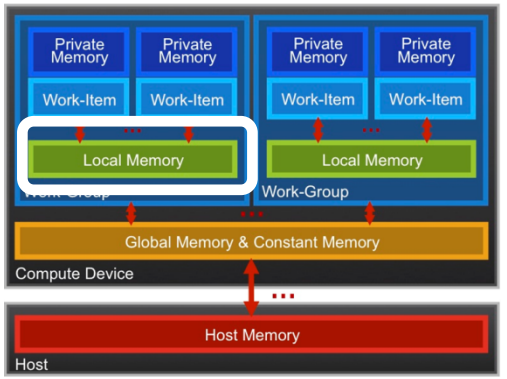



- A work-group's shared memory
  - Typically 10's of KBytes per Compute Unit*
  - Use Local Memory to hold data that can be **reused by all the work-items** in a work-group
  - As multiple Work-Groups may be running on each Compute Unit (CU), only a fraction of the total Local Memory size may be available to each Work-Group

- How do you create and manage local memory?
  - Create and Allocate local memory on the host
    - `cl::LocalSpaceArg localmem = cl::Local(sizeof(float)*N);`
  - Setup the kernel to receive local memory blocks
    - `cl::make_kernel<int, cl::Buffer, cl::LocalSpaceArg> foo(program, "bar");`
  - Mark kernel arguments that are from local memory as __local
  - Your kernels are responsible for transferring data between Local and Global/Constant memories … there are built-in functions to help (async_work_group_copy(), async_workgroup_strided_copy(), etc)

*Size and performance numbers are approximate and for a high-end discrete GPU, circa 2011

# Local Memory performance hints




- Local Memory doesn't always help…
  - CPUs don't have special hardware for it
  - This can mean excessive use of Local Memory might slow down kernels on CPUs
  - GPUs now have effective on-chip caches which can provide much of the benefit of Local Memory but without programmer intervention
  - Access patterns to Local Memory affect performance in a similar way to accessing Global Memory
    - Have to think about things like coalescence & bank conflicts
  - So, your mileage may vary!

# Memory Consistency

- OpenCL uses a **relaxed consistency** memory model; i.e.
  - The state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times.
- Within a work-item:
  - Memory has load/store consistency to the work-item's private view of memory, i.e. it sees its own reads and writes correctly
- Within a work-group:
  - Local memory is consistent between work-items at a barrier.
- Global memory is consistent within a work-group at a barrier, **but *not* guaranteed across different work-groups!!**
  - This is a common source of bugs!
- Consistency of memory shared between commands (e.g. kernel invocations) is enforced by synchronization (barriers, events, in-order queue)

# Work-Item Synchronization

Ensure correct order of memory operations to local or global memory (with flushes or queuing a memory fence)

- Within a work-group

  `void barrier()`

  – Takes optional flags

    CLK_LOCAL_MEM_FENCE and/or CLK_GLOBAL_MEM_FENCE

  – A work-item that encounters a barrier() will wait until ALL work-items in its work-group reach the barrier()

  – Corollary: If a barrier() is inside a branch, then the branch must be taken by either:

    - ALL work-items in the work-group, OR
    - NO work-item in the work-group

- Across work-groups

  – No guarantees as to where and when a particular work-group will be executed relative to another work-group

  – Cannot exchange data, or have barrier-like synchronization between two different work-groups! (Critical issue!)

  – Only solution: finish the kernel and start another

# Exercise 8: C=A*B, share B column between work-items

- Goal:
  - To give you experience working with local memory.
- Procedure:
  - Start from your last matrix multiplication program (the row-based method using private memory). Modify it so each work group copies a column of B into local memory and shares it between work-items in a work-group.
- Expected output:
  - Verify that your result is correct. Output the runtime and the MFLOPS.

```
Tell the kernel an argument is local
       __local

Find a work-item's ID in a work-group and size of a work-group:
        int iloc = get_local_id(0);
        int nloc = get_local_size(0);

Work items copy global into local data, so you need to
synchronize them
       barrier(CLK_LOCAL_MEM_FENCE);

Allocate local memory on the host and pass it into the kernel
       cl::LocalSpaceArg localmem = cl::Local(sizeof(float) * N);
       cl::make_kernel<int, cl::Buffer, cl::LocalSpaceArg>
         rowcol(program, "mmul");
```

# Matrix multiplication: B column shared between work-items

```
__kernel void mmul(
      const int N,
    __global float *A,
    __global float *B,
    __global float *C,
    __local  float *Bwrk)
{
 int j, k;
 int i =
    get_global_id(0);

 int iloc  =
      get_local_id(0);

 int nloc =
      get_local_size(0);

 float tmp;
 float Awrk[1024];

 for (k = 0; k < N; k++)
   Awrk[k] = A[i*N+k];

 for (j = 0; j < N; j++) {

   for (k=iloc; k<N; k+=nloc)
     Bwrk[k] = B[k*N+j];

   barrier(CLK_LOCAL_MEM_FENCE);

   tmp = 0.0f;
   for (k = 0; k < N; k++)
     tmp += Awrk[k]*Bwrk[k];

   C[i*N+j] = tmp;

   barrier(CLK_LOCAL_MEM_FENCE);
 }
}
```

Pass a work array in local memory to hold a column of B. All the work-items do the copy "in parallel" using a cyclic loop distribution (hence why we need iloc and nloc)

# Mat. Mul. host program (Share a column of B within a work-group)

```
#define DEVICE CL_DEVICE_TYPE_DEFAULT

  // declarations (not shown)
  sz = N * N;
  std::vector<float> h_A(sz);
  std::vector<float> h_B(sz);
  std::vector<float> h_C(sz);

  cl::Buffer d_A, d_B, d_C;

  // initialize matrices  and setup
  // the problem (not shown)

  cl::Context context(DEVICE);
  cl::Program program(context,
    util::loadProgram("mmulCrow.cl"),
        true);
```

```
cl::CommandQueue queue(context);

cl::make_kernel<int, cl::Buffer, cl::Buffer,
     cl::Buffer, cl::LocalSpaceArg>
   mmul(program, "mmul");

d_A = cl::Buffer(context,
                  h_A.begin(), h_A.end(), true);
d_B = cl::Buffer(context,
                  h_B.begin(), h_B.end(), true);
d_C = cl::Buffer(context,
                  CL_MEM_WRITE_ONLY,
                  sizeof(float) * sz);
cl::LocalSpaceArg Bwrk =
              cl::Local(sizeof(float) * sz);

mmul( cl::EnqueueArgs(
          queue, cl::NDRange(N), cl::NDRange(64) ),
        N, d_A,  d_B,  d_C, Bwrk );

cl::copy(queue, d_C, h_C.begin(), h_C.end());
```

# Mat. Mul. host program (Share a column of B within a work-group)

```
#define DEVICE CL_DEVICE_TYPE_DEFAULT

// declarations (not shown)
sz = N * N;
std::vector<float> h_A(sz);
```

Change host program to pass local memory to kernels.

- Add an arg of type LocalSpaceArg is needed.
- Allocate the size of local memory
- Update argument list in kernel functor

```
cl::Context context(DEVICE);
cl::Program program(context,
  util::loadProgram("mmulCrow.cl“),
     true);
cl::CommandQueue queue(context);

cl::make_kernel<int, cl::Buffer, cl::Buffer,
     cl::Buffer, cl::LocalSpaceArg>
   mmul(program, "mmul");

d_A = cl::Buffer(context,
                 h_A.begin(), h_A.end(), true);
d_B = cl::Buffer(context,
                 h_B.begin(), h_B.end(), true);
d_C = cl::Buffer(context,
                 CL_MEM_WRITE_ONLY,
                 sizeof(float) * sz);
cl::LocalSpaceArg Bwrk =
                 cl::Local(sizeof(float) * sz);

mmul( cl::EnqueueArgs(
          queue, cl::NDRange(N), cl::NDRange(64) ),
        N, d_A,  d_B,  d_C, Bwrk );

cl::copy(queue, d_C, h_C.begin(), h_C.end());
```

# Matrix multiplication performance

- Matrices are stored in global memory.

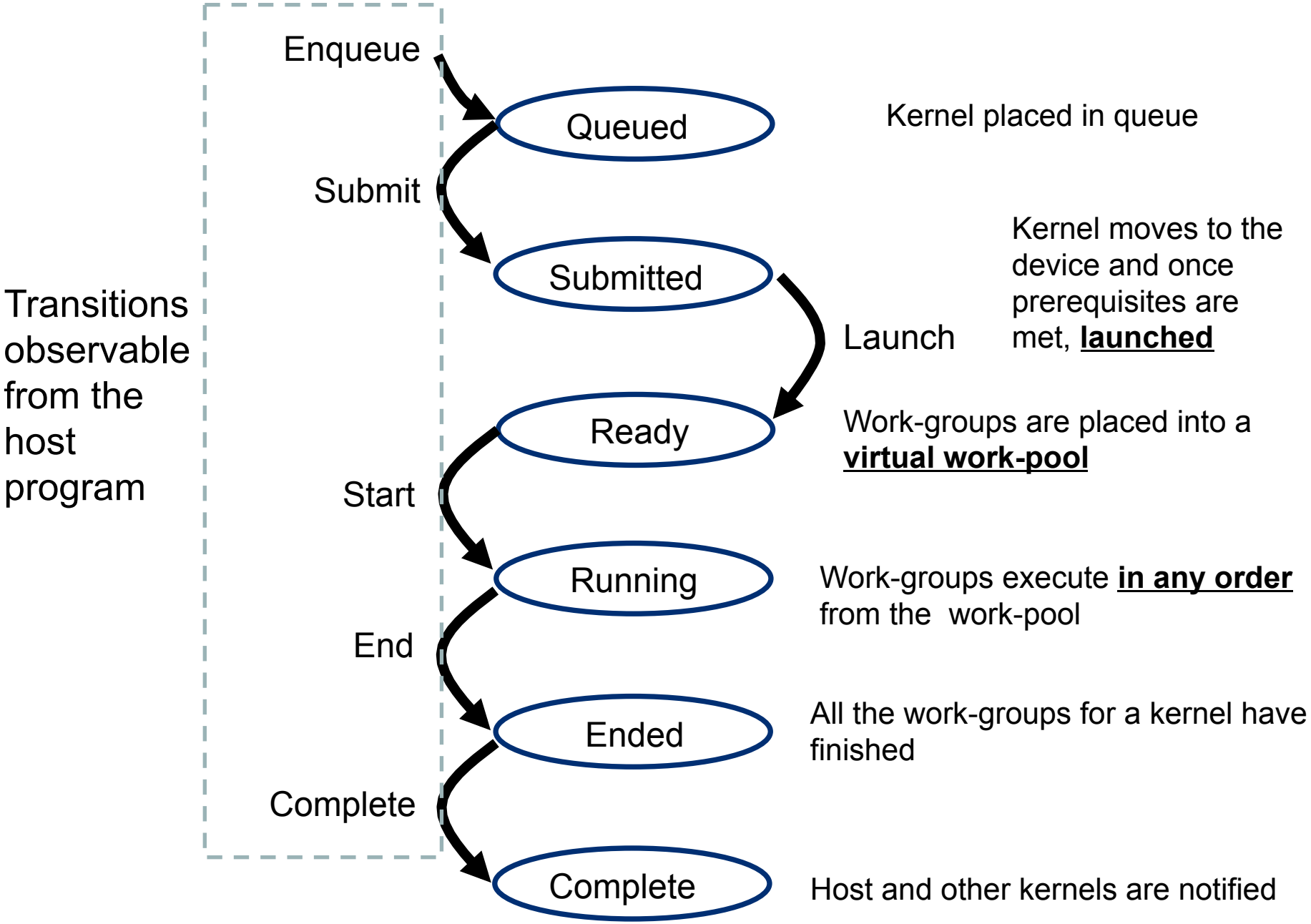
| Case | MFLOPS | |
|---|---|---|
| | CPU | GPU |
| Sequential C (not OpenCL) | 887.2 | N/A |
| C(i,j) per work-item, all global | 3,926.1 | 3,720.9 |
| C row per work-item, all global | 3,379.5 | 4,195.8 |
| C row per work-item, A row private | 3,385.8 | 8,584.3 |
| C row per work-item, A private, B local | 10,047.5 | 8,181.9 |

Device is Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs
Device is Intel® Xeon® CPU, E5649 @ 2.53GHz

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

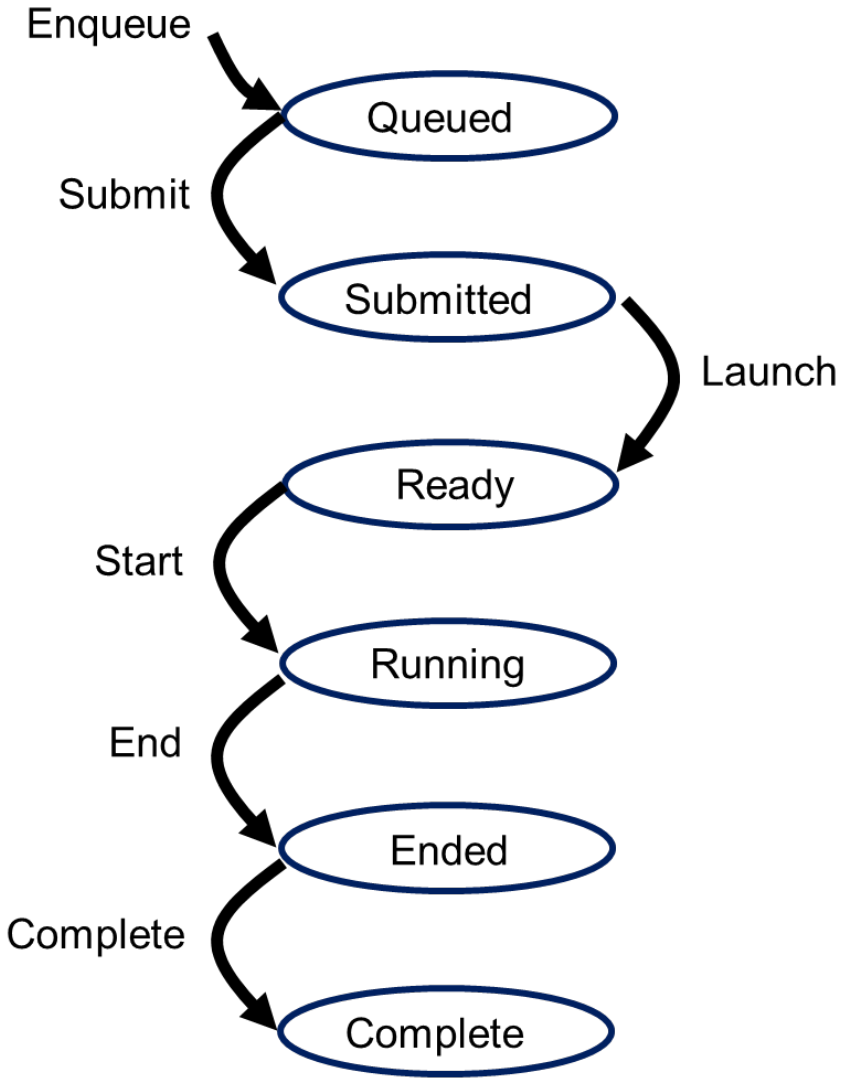
# HIGH PERFORMANCE OPENCL

# Execution Model: How do kernels execute?



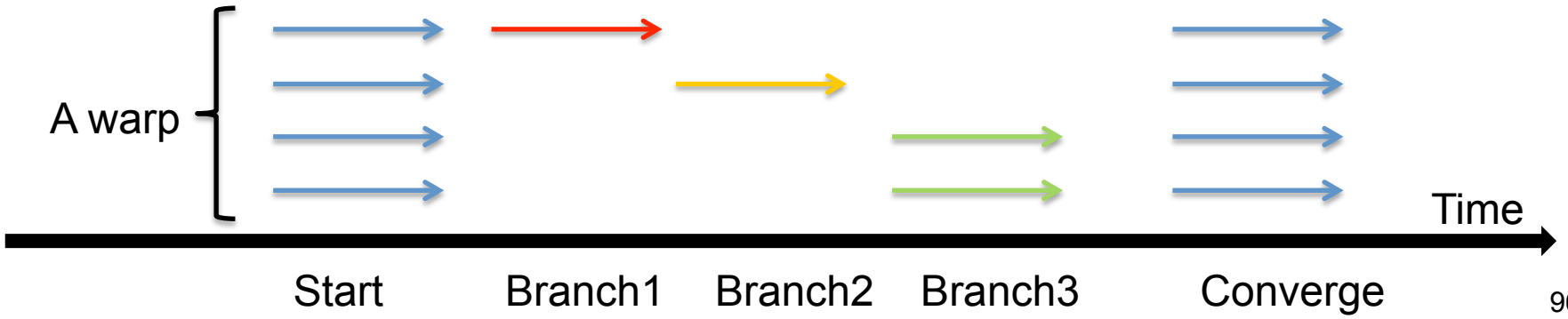
# High Performance OpenCL

- A good OpenCL program is optimized for high throughput … work-groups are scheduled from the work-pool and stream through the device, hopefully without observable stalls.
- By having more work than processing elements, you can hide memory latencies and keep all the hardware busy.
- Instruction overhead minimized … work-groups broken down into collections that execute together in “SIMD mode” from a single stream of instructions. (Warp for NVIDIA, Wavefront for AMD)



# Work-item divergence

- What happens when work-items branch?
- Work-items are gathered into collections that run together on the hardware (This is the concept of a “Warp” from CUDA).
- The hardware runs this collection in a SIMD data parallel model with all the work-items starting together from the same program address.
- Each work-item has its own instruction address counter and register state
  - Each work-item is free to branch and execute independently (diverge)
  - Provide the MIMD abstraction
- Branch behavior
  - Each branch will be executed serially
  - Work-items not following the current branch will be disabled (masked)

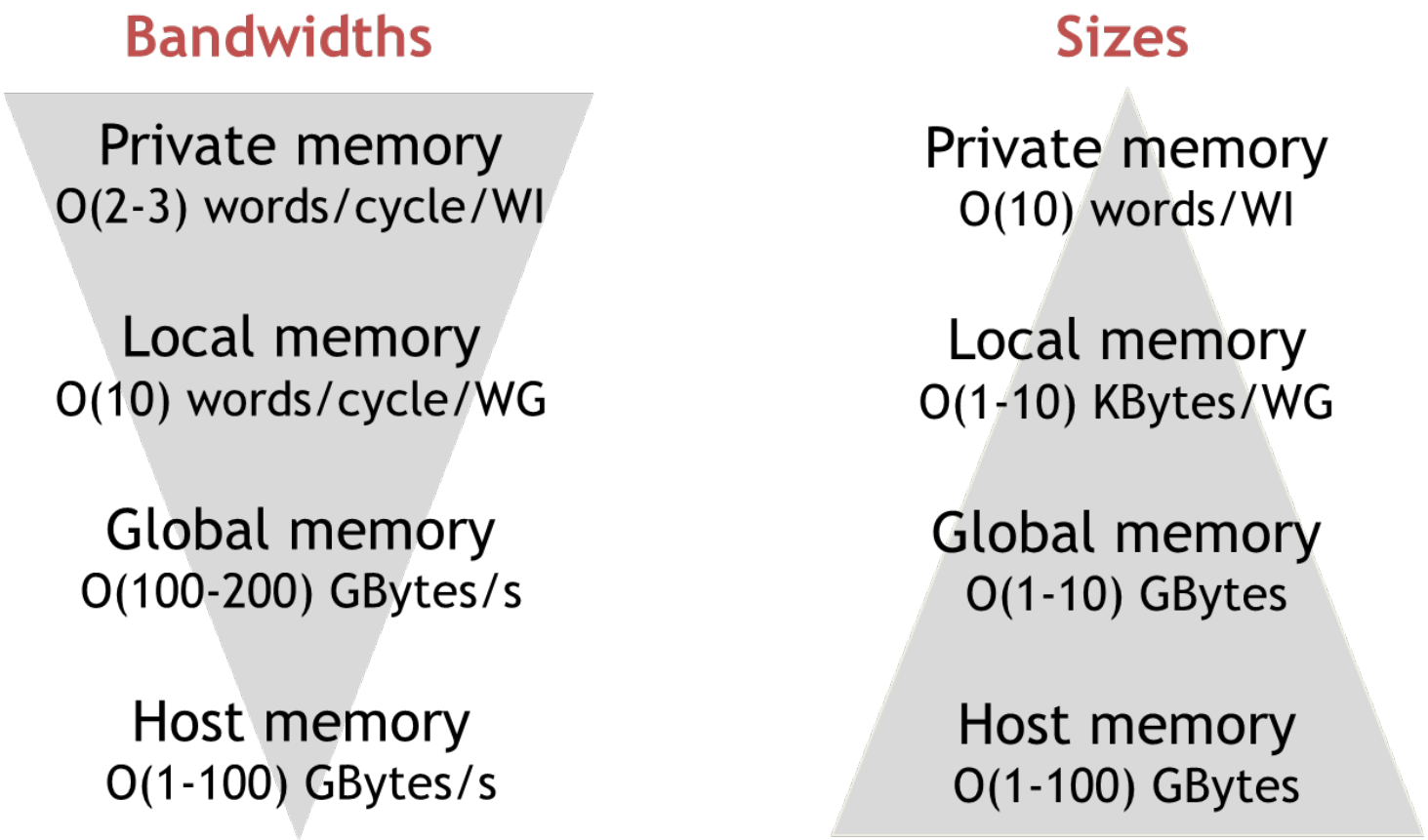


# Keep the processing elements (PE) busy

- You need **uniform** execution of the work-items scheduled to execute together. Avoid **divergent control flows**.
- **Occupancy**: a measure of the fraction of time during a computation when the PE's are busy. Goal is to keep this number high (well over 50%).
- Pay attention to the number of work-items and work-group sizes
  - Rule of thumb: On a modern GPU you want at least 4 work-items per PE in a Compute Unit
  - More work-items are better, but diminishing returns, and there is an upper limit
    - Each work item consumes PE finite resources (registers etc)

# Use the Memory Hierarchy effectively

- Organize your computation so it puts the most frequently used data in faster memory … optimized of course around the available size.




*Size and performance numbers are approximate and for a high-end discrete GPU, circa 2011

# Optimization issues

- Efficient access to memory
  - **Memory coalescing**
    - Ideally get work-item i to access data[i] and work-item j to access data[j] at the same time etc.
  - **Memory alignment**
    - Pad arrays to keep everything aligned to multiples of 16, 32 or 64 bytes
- Registers per Work-Item– ideally low and a nice divisor of the number of hardware registers per Compute Unit
  - E.g. 32,768 on M2050 GPUs
  - These are statically allocated and shared between all Work-Items and Work-Groups assigned to each Compute Unit
  - Four Work-Groups of 1,024 Work-Items each would result in just 8 registers per Work-Item! Typically aim for 16-32 registers per Work-Item

# Memory layout is critical to performance

- "Structure of Arrays vs. Array of Structures" problem:

```
struct { float x, y, z, a; } Point;
```

- Structure of Arrays (SoA) suits memory coalescence on GPUs


Adjacent work-items like to access adjacent memory

- Array of Structures (AoS) may suit cache hierarchies on CPUs


Individual work-items like to access adjacent memory

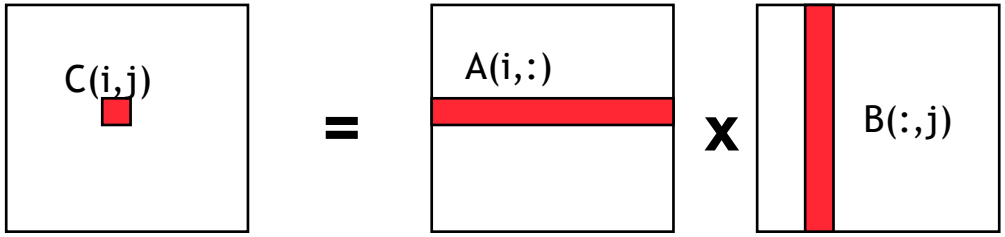
# Portable performance in OpenCL

- **Don't optimize too much for any one platform**, e.g.
  - Don't write specifically for certain warp/wavefront sizes etc
  - Be careful not to max out specific sizes of local/global memory
  - OpenCL's vector data types have varying degrees of support – faster on some devices, slower on others
  - Some devices have caches in their memory hierarchies, some don't, and it can make a big difference to your performance without you realizing
  - Choosing the allocation of Work-Items to Work-Groups and dimensions on your kernel launches
  - Performance differences between unified vs. disjoint host/global memories
  - Double precision performance varies considerably from device to device

- Recommend trying your code on several different platforms to see what happens (profiling is good!)
  - At least two different GPUs (ideally different vendors!) and at least one CPU

# Consider our matrix multiplication example

- So far, we've used matrix multiplication to explore the memory hierarchy, but we haven't really thought about what the algorithm needs to run REALLY fast?




Dot product of a row of A and a column of B for each element of C

- To make this fast, you need to break the problem down into chunks that do lots of work for sub problems that fit in fast memory (OpenCL local memory).

# Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
    int i, j, k;
    for (i = 0; i < N; i++) {
      for (j = 0; j < N; j++) {
        for (k = 0; k < N; k++) {
          C[i*N+j] += A[i*N+k] * B[k*N+j];
        }
      }
    }
}
```

# Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
    int i, j, k;
    for (i = 0; i < N; i++)
      for (j = 0; j < N; j++)
        for (k = 0; k < N; k++)
          C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

Let's get rid of all those ugly brackets

# Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 float tmp;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (i = ib*NB; i < (ib+1)*NB; i++)
     for (jb = 0; jb < NB; jb++)
       for (j = jb*NB; j < (jb+1)*NB; j++)
         for (kb = 0; kb < NB; kb++)
           for (k = kb*NB; k < (kb+1)*NB; k++)
             C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

Break each loop into chunks with a size chosen to match the size of your fast memory

# Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 float tmp;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (jb = 0; jb < NB; jb++)
     for (kb = 0; kb < NB; kb++)

 for (i = ib*NB; i < (ib+1)*NB; i++)
   for (j = jb*NB; j < (jb+1)*NB; j++)
     for (k = kb*NB; k < (kb+1)*NB; k++)
       C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

Rearrange loop nest to move loops over blocks “out” and leave loops over a single block together

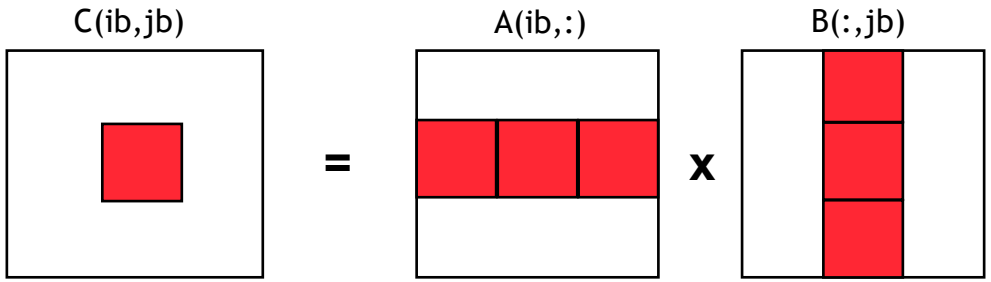
# Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 float tmp;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (jb = 0; jb < NB; jb++)
     for (kb = 0; kb < NB; kb++)

 for (i = ib*NB; i < (ib+1)*NB; i++)
   for (j = jb*NB; j < (jb+1)*NB; j++)
     for (k = kb*NB; k < (kb+1)*NB; k++)
       C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

This is just a local matrix multiplication of a single block

# Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (jb = 0; jb < NB; jb++)
     for (kb = 0; kb < NB; kb++)
       sgemm(C, A, B, …)    // C_ib,jb = A_ib,kb * B_kb,jb
```




```
}
```

Note: sgemm is the name of the level three BLAS routine to multiply two matrices

# Exercise 9: The C = A * B Competition

- Goal:
  - To see who can get the best performance from their matrix multiplication program.
- Procedure:
  - Start from which ever matrix multiplication program you choose.
  - Make it fast.  But TEST OUTPUT …. You must get correct results.
  - Remember … block your algorithms to exploit the natural execution widths of your hardware, make good use of local and private memory.
- Expected output:
  - Test your result and verify that it is correct.  Output the runtime and the MFLOPS.

# Blocked matrix multiply: kernel

```
#define blksz 16
__kernel void mmul(
             const unsigned int N,
             __global float* A,
             __global float* B,
             __global float* C,
             __local  float* Awrk,
             __local  float* Bwrk)
{
  int kloc, Kblk;
  float Ctmp=0.0f;

  //  compute element C(i,j)
  int i = get_global_id(0);
  int j = get_global_id(1);

  // Element C(i,j) is in block C(Iblk,Jblk)
  int Iblk = get_group_id(0);
  int Jblk = get_group_id(1);

  // C(i,j) is element C(iloc, jloc)
  //  of block C(Iblk, Jblk)
  int iloc = get_local_id(0);
  int jloc = get_local_id(1);
  int Num_BLK = N/blksz;

  // upper-left-corner and inc for A and B
  int Abase = Iblk*N*blksz;   int Ainc  = blksz;
  int Bbase = Jblk*blksz;     int Binc  = blksz*N;

// C(Iblk,Jblk) = (sum over Kblk) A(Iblk,Kblk)*B(Kblk,Jblk)
  for (Kblk = 0;  Kblk<Num_BLK;  Kblk++)
  {
     //Load A(Iblk,Kblk) and B(Kblk,Jblk).
     //Each work-item loads a single element of the two
     //blocks which are shared with the entire work-group

     Awrk[jloc*blksz+iloc] = A[Abase+jloc*N+iloc];
     Bwrk[jloc*blksz+iloc] = B[Bbase+jloc*N+iloc];

     barrier(CLK_LOCAL_MEM_FENCE);

     #pragma unroll
     for(kloc=0; kloc<blksz; kloc++)
        Ctmp+=Awrk[jloc*blksz+kloc]*Bwrk[kloc*blksz+iloc];

     barrier(CLK_LOCAL_MEM_FENCE);
     Abase += Ainc;       Bbase += Binc;
   }
   C[j*N+i] = Ctmp;
}
```

# Blocked matrix multiply: kernel

It's getting the indices right that makes this hard

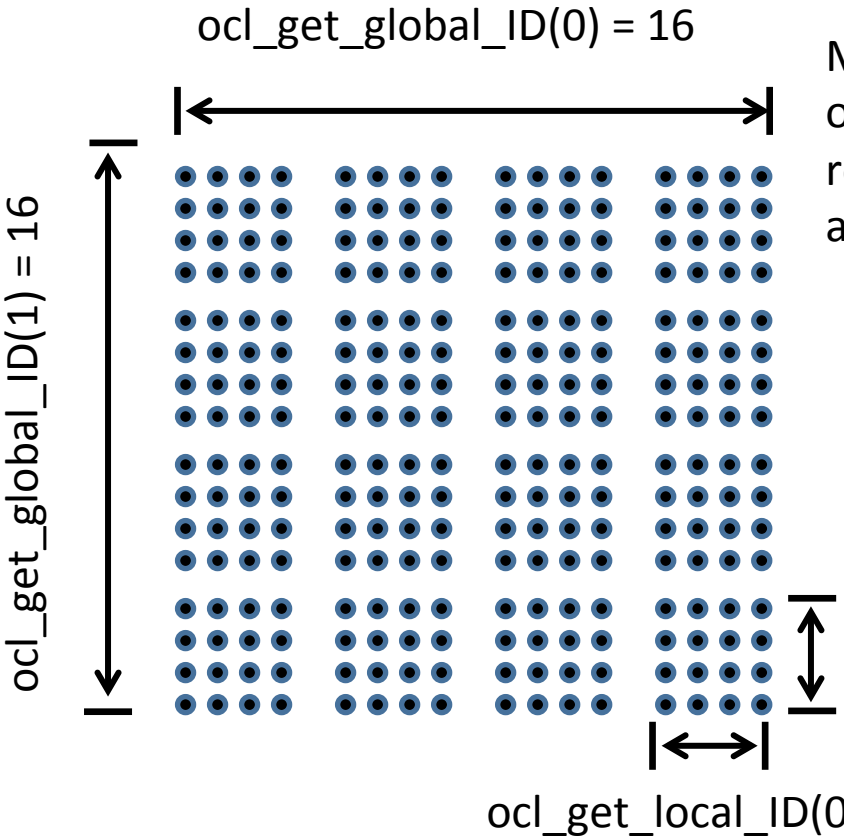
```
#define blksz 16
__kernel void mmul(
            const unsigned int N,
            __global float* A,
            __global float* B,
            __global float* C,
            __local  float* Awrk,
            __local  float* Bwrk)
{
  int kloc, Kblk;
  float Ctmp=0.0f;

  //  compute element C(i,j)
  int i = get_global_id(0);
  int j = get_global_id(1);

  // Element C(i,j) is in block C(Iblk,Jblk)
  int Iblk = get_group_id(0);
  int Jblk = get_group_id(1);

  // C(i,j) is element C(iloc, jloc)
  //  of block C(Iblk, Jblk)
  int iloc = get_local_id(0);
  int jloc = get_local_id(1);
  int Num_BLK = N/blksz;

  // upper-left-corner and inc for A and B
  int Abase = Iblk*N*blksz;   int Ainc  = blksz;
  int Bbase = Jblk*blksz;     int Binc  = blksz*N;

// C(Iblk,Jblk) = (sum over Kblk) A(Iblk,Kblk)*B(Kblk,Jblk)
  for (Kblk = 0;  Kblk<Num_BLK;  Kblk++)
  {
    //Load A(Iblk,Kblk) and B(Kblk,Jblk).
    //Each work-item loads a single element of the two
    //blocks which are shared with the entire work-group

    Awrk[jloc*blksz+iloc] = A[Abase+jloc*N+iloc];
    Bwrk[jloc*blksz+iloc] = B[Bbase+jloc*N+iloc];

    barrier(CLK_LOCAL_MEM_FENCE);

    #pragma unroll
    for(kloc=0; kloc<blksz; kloc++)
      Ctmp+=Awrk[jloc*blksz+kloc]*Bwrk[kloc*blksz+iloc];

    barrier(CLK_LOCAL_MEM_FENCE);
    Abase += Ainc;      Bbase += Binc;
  }
  C[j*N+i] = Ctmp;
}
```

Load A and B blocks, wait for all work-items to finish

Wait for everyone to finish before going to next iteration of Kblk loop.

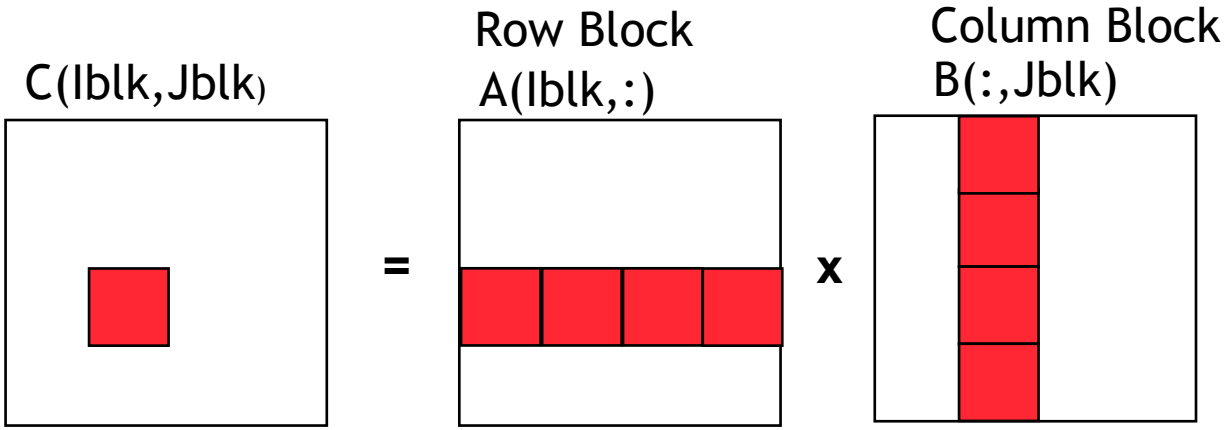
# Mapping into A, B, and C from each work item

16 x 16 NDRange with workgroups of size 4x4




Map Matrices A, B and C onto this NDRange in a row major order (N = 16 and Blksz = 4).

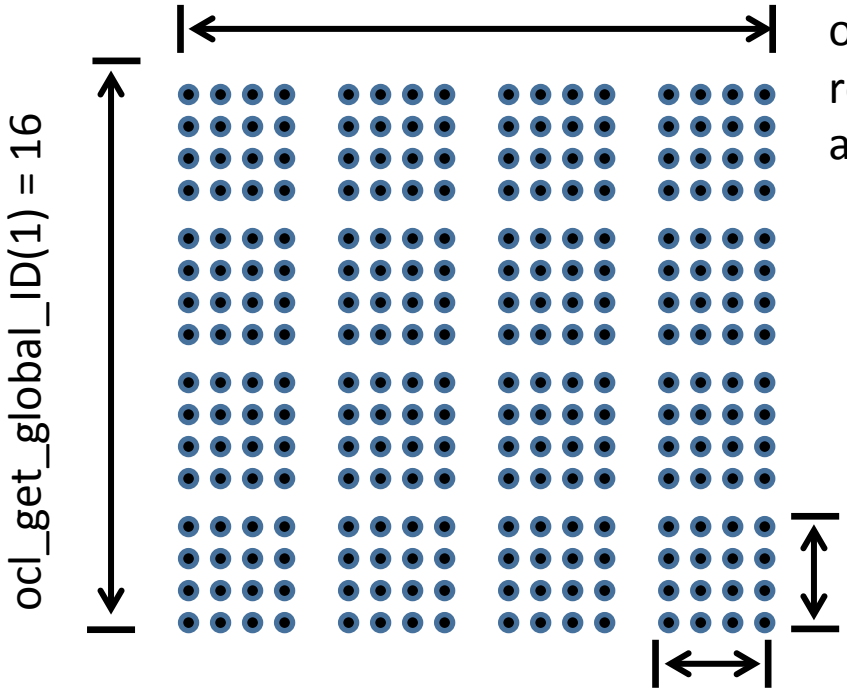
# Mapping into A, B, and C from each work item



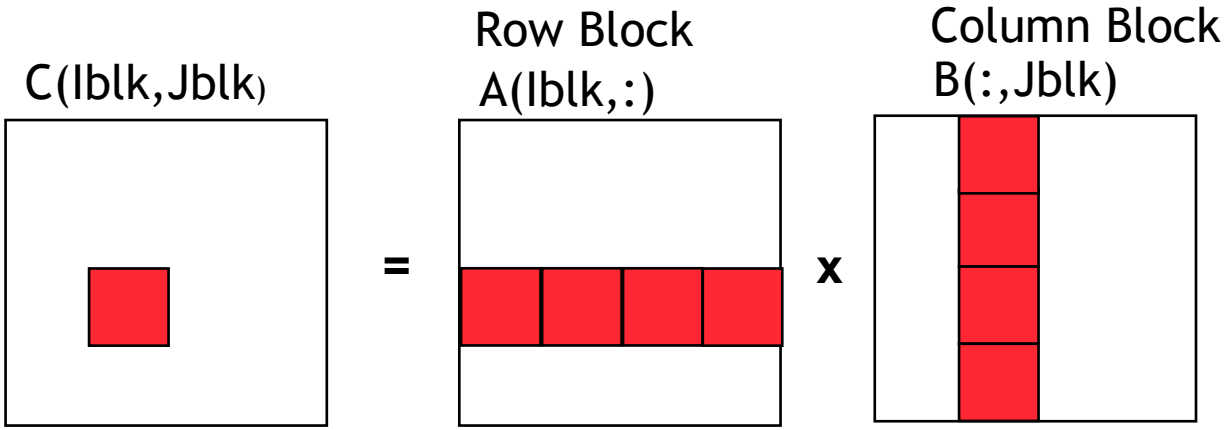

16 x 16 NDRange with workgroups of size 4x4

ocl_get_global_ID(0) = 16

Map Matrices A, B and C onto this NDRange in a row major order (N = 16 and Blksz = 4).



# Mapping into A, B, and C from each work item




16 x 16 NDRange with workgroups of size 4x4

Consider indices for computation of the block C(Iblk=2, Jblk=1)

Bbase = Jblk*blksz = 1*4

Abase = Iblk*N*blksz
= 1 * 16 * 4

Subsequent A blocks by shifting index by Ainc = blksz = 4

Subsequent B blocks by shifting index by Binc = blksz * N
= 4 * 16 = 64

Map Matrices A, B and C onto this NDRange in a row major order (N = 16 and Blksz = 4).

# Blocked matrix multiply: Host

```
#define DEVICE CL_DEVICE_TYPE_DEFAULT

// declarations (not shown)
sz = N * N;
std::vector<float> h_A(sz);
std::vector<float> h_B(sz);
std::vector<float> h_C(sz);

cl::Buffer d_A, d_B, d_C;

// initialize matrices  and setup
// the problem (not shown)

cl::Context context(DEVICE);
cl::Program program(context,
   util::loadProgram("mmul.cl“),
      true);

cl::CommandQueue queue(context);
```

```
cl::make_kernel
      <int, cl::Buffer, cl::Buffer, cl::Buffer,
            cl::LocalSpaceArg, cl::LocalSpaceArg>
       mmul(program, "mmul");

d_A = cl::Buffer(context,
                    h_A.begin(), h_A.end(), true);
d_B = cl::Buffer(context,
                    h_B.begin(), h_B.end(), true);
d_C = cl::Buffer(context,
          CL_MEM_WRITE_ONLY, sizeof(float) * sz);

cl::LocalSpaceArg Awrk =
               cl::Local(sizeof(float) * N);
cl::LocalSpaceArg Bwrk =
               cl::Local(sizeof(float) * N);
mmul( cl::EnqueueArgs( queue,
             cl::NDRange(N,N),  cl::NDRange(16,16) ),
         N, d_A,  d_B,  d_C, Awrk, Bwrk );

cl::copy(queue, d_C, h_C.begin(), h_C.end());

// Timing and check results (not shown)
```

# Blocked matrix multiply: Host

```
#define DEVICE CL_DEVICE_TYPE_DEFAULT

// declarations (not shown)
sz = N * N;
std::vector<float> h_A(sz);
std::vector<float> h_B(sz);
std::vector<float> h_C(sz);

cl::Buffer d_A, d_B, d_C;

// initializ
// the pro

cl::Context context(DEVICE);
cl::Program program(context,
   util::loadProgram("mmul.cl"),
      true);

cl::CommandQueue queue(context);
```

Setup local memory with blocks of A and B (16 by 16) that should fit in local memory.

```
cl::make_kernel
      <int, cl::Buffer, cl::Buffer, cl::Buffer,
            cl::LocalSpaceArg, cl::LocalSpaceArg>
       mmul(program, "mmul");

d_A = cl::Buffer(context,
                  h_A.begin(), h_A.end(), true);
d_B = cl::Buffer(context,
                  h_B.begin(), h_B.end(), true);
d_C = cl::Buffer(context,
          CL_MEM_WRITE_ONLY, sizeof(float) * sz);

cl::LocalSpaceArg Awrk =
              cl::Local(sizeof(float) * 16*16);
cl::LocalSpaceArg Bwrk =
              cl::Local(sizeof(float) * 16*16);

mmul( cl::EnqueueArgs( queue,
          cl::NDRange(N,N),  cl::NDRange(16,16) ),
        N, d_A,  d_B,  d_C, Awrk, Bwrk );

cl::copy(queue, d_C, h_C.begin(), h_C.end());

// Timing and check results (not shown)
```

One work-item per element of the C matrix organized into 16 by 16 blocks.

# Matrix multiplication performance

- Matrices are stored in global memory.

| Case | MFLOPS | |
|---|---|---|
| | CPU | GPU |
| Sequential C (not OpenCL) | 887.2 | N/A |
| C(i,j) per work-item, all global | 3,926.1 | 3,720.9 |
| C row per work-item, all global | 3,379.5 | 4,195.8 |
| C row per work-item, A row private | 3,385.8 | 8,584.3 |
| C row per work-item, A private, B local | 10,047.5 | 8,181.9 |
| Block oriented approach using local | | 119,304.6 |

Device is Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs
Device is Intel® Xeon® CPU, E5649 @ 2.53GHz

CuBLAS performance 283,366.4 MFLOPS



These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

# Matrix multiplication performance (CPU)

- Matrices are stored in global memory.

| Case | MFLOPS CPU |
|---|---|
| Sequential C (not OpenCL, compiled /O3) | 224.4 |
| C(i,j) per work-item, all global | 841.5 |
| C row per work-item, all global | 869.1 |
| C row per work-item, A row private | 1,038.4 |
| C row per work-item, A private, B local | 3,984.2 |
| Block oriented approach using local (blksz=8) | 7,482.5 |
| Block oriented approach using local (blksz=16) | 12,271.3 |
| Block oriented approach using local (blksz=32) | 16,268.8 |
| Intel MKL SGEMM | 63,780.6 |

Device is Intel® Core™ i5-2520M CPU @2.5 GHz (dual core) Windows 7 64 bit OS, Intel compiler 64 bit version 13.1.1.171, OpenCL SDK 2013, MKL 11.0 update 3.



These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

# THE OPENCL ZOO

# The OpenCL Zoo

- We have a range of systems available at the University of Bristol and NERSC
- Time will be made available so you can take the code you wrote over the course of the day and run on these different systems.  The goal is to explore the concept of performance portability.

# The OpenCL Zoo at Bristol

- Full instructions at http://uob-hpc.github.io/zoo
- ssh to hpc.cs.bris.ac.uk
- Use your username from Dirac
- Your Zoo password is: sc14trainX where X is the number you had + 50

# The Zoo at the University of Bristol

# SOME CONCLUDING REMARKS

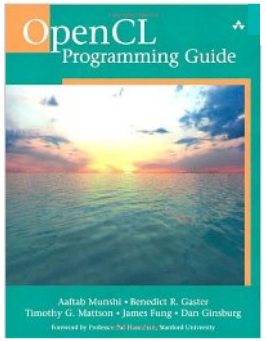
# Conclusion

- OpenCL has ***widespread*** industrial support
- OpenCL defines a platform-API/framework for ***heterogeneous computing***, not just GPGPU or CPU-offload programming
- OpenCL has the potential to deliver ***portably performant code***; but it has to be used correctly
- The latest ***C++ and Python APIs*** makes developing OpenCL programs much simpler than before
- The future is clear:
  - OpenCL is the ***only*** parallel programming standard that enables mixing task parallel and data parallel code in a single program while load balancing across **ALL** of the platform's available resources.

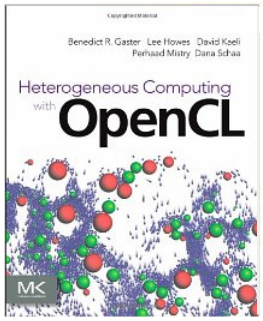
# Other important related trends

- OpenCL's Standard Portable Intermediate Representation (SPIR)
  - Based on LLVM's IR
  - Makes interchangeable front- and back-ends straightforward
- OpenCL 2.0
  - Adding High Level Model (HLM)
  - Lots of other improvements
- For the latest news on SPIR and new OpenCL versions see:
  - http://www.khronos.org/opencl/

# Resources: https://www.khronos.org/opencl/




**OpenCL Programming Guide**:
Aaftab Munshi, Benedict Gaster, Timothy G. Mattson and James Fung, 2011




**Heterogeneous Computing with OpenCL**
Benedict Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry and Dana Schaa, 2011

**HandsOnOpenCL** online resource:
http://handsonopencl.github.io

# Become part of the OpenCL community!

- New annual OpenCL conference
  - http://www.iwocl.org/
  - Held in May each year
  - CFP to be announced at SC


- OpenCL BOF at SuperComputing
- ACTION: become an active member of the OpenCL community!!

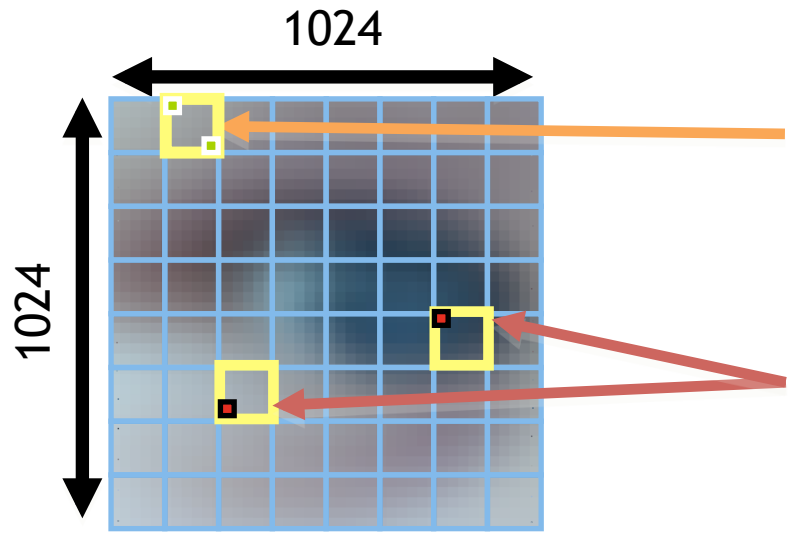
Thank you for coming!

Appendix A

# SYNCHRONIZATION IN OPENCL

# Consider N-dimensional domain of work-items

- Global Dimensions:
  - 1024x1024 (whole problem space)
- Local Dimensions:
  - 128x128 (work-group, executes together)



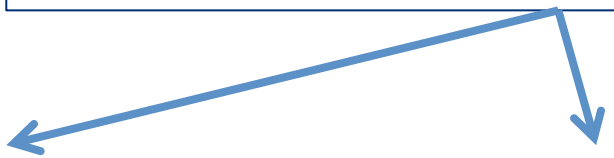

**Synchronization between work-items possible only within work-groups: barriers and memory fences**

**Cannot synchronize between work-groups within a kernel**

Synchronization: when multiple units of execution (e.g. work-items) are brought to a known point in their execution. Most common example is a barrier … i.e. all units of execution "in scope" arrive at the barrier before any proceed.

# Work-Item Synchronization

Ensure correct order of memory operations to local or global memory (with flushes or queuing a memory fence)



- Within a work-group

  `void barrier()`

  – Takes optional flags

    CLK_LOCAL_MEM_FENCE and/or CLK_GLOBAL_MEM_FENCE

  – A work-item that encounters a **barrier**() will wait until ALL work-items in its work-group reach the **barrier**()

  – **Corollary**: If a **barrier**() is inside a branch, then the branch must be taken by either:

    - **ALL** work-items in the work-group, OR
    - **NO** work-item in the work-group

- Across work-groups

  – No guarantees as to where and when a particular work-group will be executed relative to another work-group

  – **Cannot exchange data, or have barrier-like synchronization between two different work-groups! (Critical issue!)**

  – **Only solution: finish the kernel and start another**

# Where might we need synchronization?

- Consider a reduction … reduce a set of numbers to a single value
  - E.g. find sum of all elements in an array
- Sequential code

```
int reduce(int Ndim, int *A)
{
  int sum = 0;
  for(int i = 0; i < Ndim; i++)
    sum += A[i];
}
```

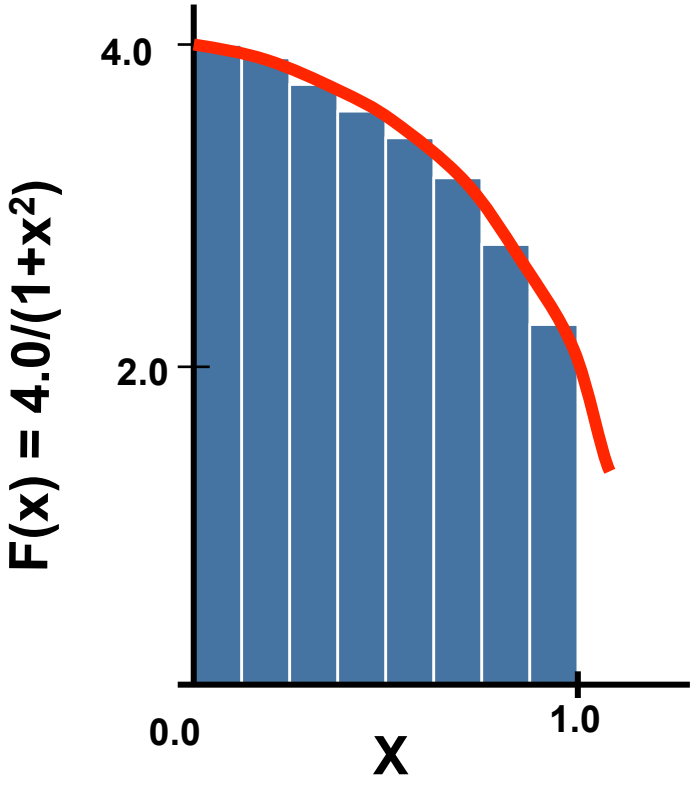
# Simple parallel reduction

- A reduction can be carried out in three steps:
  1. Each work-item sums its private values into a local array indexed by the work-item's local id
  2. When all the work-items have finished, one work-item sums the local array into an element of a global array (indexed by work-group id).
  3. When all work-groups have finished the kernel execution, the global array is summed on the host.

- Note: this is a simple reduction that is straightforward to implement. More efficient reductions do the work-group sums in parallel on the device rather than on the host. These more scalable reductions are considerably more complicated to implement.

# A simple program that uses a reduction

**Numerical Integration**




Mathematically, we know that we can approximate the integral as a sum of rectangles.

Each rectangle has width and height at the middle of interval.

# Numerical integration source code

## The serial Pi program

```
static long num_steps = 100000;
double step;
void main()
{
  int i; double x, pi, sum = 0.0;

  step = 1.0/(double) num_steps;

  for (i = 0; i < num_steps; i++) {
    x = (i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
  }
  pi = step * sum;
}
```

# Numerical integration source code

## The serial Pi program

```
static long num_steps = 100000;
float step;
void main()
{
  int i; float x, pi, sum = 0.0;

  step = 1.0f/(float) num_steps;

  for (i = 0; i < num_steps; i++) {
    x = (i+0.5f)*step;
    sum = sum + 4.0/(1.0+x*x);
  }
  pi = step * sum;
}
```

Let's do this with **float** just to keep GPUs happy.

# Exercise 10: The Pi program

- Goal:
  - To understand synchronization between work-items in the OpenCL C kernel programming language.
  - To get more practice writing kernel and host code.
- Procedure:
  - Start with the provided serial program to estimate Pi through numerical integration
  - Write a kernel and host program to compute the numerical integral using OpenCL
  - Note: You will need to implement a reduction
- Expected output:
  - Output result plus an estimate of the error in the result
  - Report the runtime

Hint: you will want each work-item to do many iterations of the loop, i.e. don't create one work-item per loop iteration. To do so would make the reduction so costly that performance would be terrible.

# The Pi program: kernel

```
void reduce(   __local  float*,
               __global float*);

__kernel void pi(
   const int          niters,
   const float        step_size,
   __local  float*    local_sums,
   __global float*    partial_sums)
{
   int num_wrk_items  = get_local_size(0);
   int local_id       = get_local_id(0);
   int group_id       = get_group_id(0);
   float x, accum = 0.0f;
   int i,istart,iend;

   istart =   (group_id * num_wrk_items
                        + local_id) * niters;
   iend   = istart+niters;

   for(i= istart; i<iend; i++){
      x = (i+0.5f)*step_size;
      accum += 4.0f/(1.0f+x*x);
   }
   local_sums[local_id] = accum;
   barrier(CLK_LOCAL_MEM_FENCE);

   reduce(local_sums, partial_sums);
}

void reduce(      __local  float*    local_sums,
                  __global float*    partial_sums)
{
   int num_wrk_items  = get_local_size(0);
   int local_id       = get_local_id(0);
   int group_id       = get_group_id(0);

   float sum;      int i;

   if (local_id == 0) {
      sum = 0.0f;
         for (i=0; i<num_wrk_items; i++) {
          sum += local_sums[i];
      }
        partial_sums[group_id] = sum;
   }
}
```

# The Pi program: Host (1/2)

```
// various include files (not shown)
#define INSTEPS (512*512*512)
#define ITERS (262144)

int main(void)
{
   float *h_psum;
   int in_nsteps = INSTEPS;
   int niters = ITERS;
   int nsteps;
   float step_size;
   ::size_t nwork_groups;
   ::size_t max_size, work_group_size = 8;
   float pi_res;
   cl::Buffer d_partial_sums;

  cl::Context context(DEVICE);
  cl::Program program(context, util::loadProgram("pi_ocl.cl"), true);
  cl::CommandQueue queue(context);
  cl::Kernel ko_pi(program, "pi");

  std::vector<cl::Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();
  cl::Device device = devices[0];
```

This host program is more complicated than the others since we query the system to find the best match between the total number of integration steps and the preferred work-group size.

# The Pi program: Host (2/2)

```
 // Get the devices preferred work group size
work_group_size = ko_pi.getWorkGroupInfo<CL_KERNEL_WORK_GROUP_SIZE>(device);

  auto pi = cl::make_kernel<int, float, cl::LocalSpaceArg, cl::Buffer>(program, "pi");

 // Set num. of work groups, num. of steps, and the step size based on the work group size
 nwork_groups = in_nsteps/(work_group_size*niters);
 if ( nwork_groups < 1) {
  nwork_groups =  device.getInfo<CL_DEVICE_MAX_COMPUTE_UNITS>();
  work_group_size=in_nsteps / (nwork_groups*niters);
 }
 nsteps = work_group_size * niters * nwork_groups;  step_size = 1.0f/static_cast<float>(nsteps);
 std::vector<float> h_psum(nwork_groups);

 d_partial_sums = cl::Buffer(context, CL_MEM_WRITE_ONLY, sizeof(float) * nwork_groups);

 pi(cl::EnqueueArgs(queue, cl::NDRange(nwork_groups * work_group_size),
                           cl::NDRange(work_group_size)), niters,step_size,
                           cl::Local(sizeof(float) * work_group_size),d_partial_sums);

 cl::copy(queue, d_partial_sums, begin(h_psum), end(h_psum));

  // complete the sum and compute final integral value
  for (unsigned int i = 0, pi_res=0.0f; i< nwork_groups; i++) pi_res += h_psum[i] * step_size;
}
```

Appendix B

# VECTOR OPERATIONS WITHIN KERNELS

# Before we continue...

- The OpenCL device compilers are good at auto-vectorizing your code
  - Adjacent work-items may be packed to produce vectorized code
- By using vector operations the compiler may not optimize as successfully
- So think twice before you explicitly vectorize your OpenCL kernels, you might end up hurting performance!

# Vector operations

- Modern microprocessors include vector units:
  Functional units that carry out operations on blocks of numbers
- For example, x86 CPUs have over the years introduced MMX, SSE, and AVX instruction sets …
  characterized in part by their widths (e.g. SSE operates on 128 bits at a time, AVX 256 bits etc)
- To gain full performance from these processors it is important to exploit these vector units
- Compilers can sometimes automatically exploit vector units.
  Experience over the years has shown, however, that you all too often have to code vector operations by hand.
- Example using 128 bit wide SSE:

```
#include "xmmintrin.h "     // vector intrinsics from gcc for SSE (128 bit wide)

__m128 ramp = _mm_setr_ps(0.5, 1.5, 2.5, 3.5);   // pack 4 floats into vector register
__m128 vstep = _mm_load1_ps(&step);      // pack step into each of r 32 bit slots in a vector register
__m128 xvec; = _mm_mul_ps(ramp,vstep);   // multiple corresponding 32 bit floats and assign to xvec
```

# Vector intrinsics challenges

- Requires an assembly code style of programming:
  - Load into registers
  - Operate with register operands to produce values in another vector register
- Non portable
  - Change vector instruction set (even from the same vendor) and code must be re-written. Compilers might treat them differently too
- Consequences:
  - Very few programmers are willing to code with intrinsics
  - Most programs only exploit vector instructions that the compiler can automatically generate – which can be hit or miss
  - Most programs grossly under exploit available performance.

Solution: a high level portable vector instruction set …
which is precisely what OpenCL provides.

# Vector Types

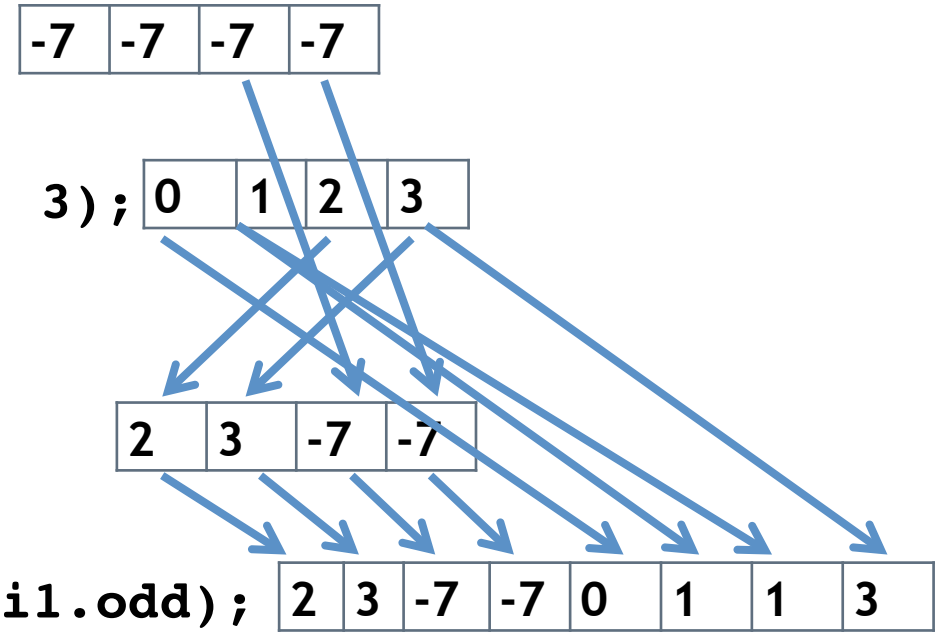
- The OpenCL C kernel programming language provides a set of vector instructions:
  - These are portable between different vector instruction sets
- These instructions support vector lengths of 2, 4, 8, and 16 … for example:
  - **`char2, ushort4, int8, float16, double2`**, …
- Properties of these types include:
  - Endian safe
  - Aligned at vector length
  - Vector operations (elementwise) and built-in functions

Remember, double (and hence vectors of double) are optional in OpenCL

# Vector Operations

- Vector literal

```
int4 vi0 = (int4) -7;

int4 vi1 = (int4) (0, 1, 2, 3);
```

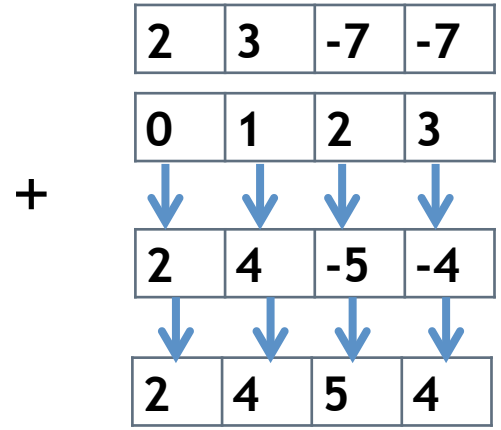



  - Vector components

```
vi0.lo = vi1.hi;

int8 v8=(int8)(vi0,vi1.s01,vi1.odd);
```

- Vector ops

```
vi0 += vi1;

vi0 = abs(vi0);
```



# Using vector operations

- You can convert a scalar loop into a vector loop using the following steps:
  - Based on the width of your vector instruction set and your problem, choose the number of values you can pack into a vector register (the width):
    - E.g. for a 128 bit wide SSE instruction set and float data (32 bit), you can pack four values (128 bits =4*32 bits) into a vector register
  - Unroll the loop to match your width (in our example, 4)
  - Set up the loop preamble and postscript. For example, if the number of loop iterations doesn't evenly divide the width, you'll need to cover the extra iterations in a loop postscript or pad your vectors in a preamble
  - Replace instructions in the body of the loop with their vector instruction counter parts

# Vector instructions example

- Scalar loop:

```
for (i = 0; i < 34; i++) x[i] = y[i] * y[i];
```

- Width for a 128-bit SSE is 128/32=4
- Unroll the loop, then add postscript and premable as needed:

```
NLP = 34+2; x[34]=x[35]=y[34]=y[35]=0.0f // preamble to zero pad
arrays
for (i = 0; i < NLP; i = i + 4) {
  x[i] = y[i] * y[i];  x[i+1] = y[i+1] * y[i*1];
  x[i+2] = y[i+2] * y[i*2];  x[i+3] = y[i+3] * y[i*3];
}
```

- Replace unrolled loop with associated vector instructions:

```
float4 x4[DIM], y4[DIM];
// DIM set to hold 34 values extended to multiple of 4 (36)
float4 zero = {0.0f, 0.0f, 0.0f, 0.0f};
NLP = 34 % 4 + 1 // 9 values … to cover the fact 34 isn't a
multiple of 4
x4[NLP-1] = 0.0f; y4[NLP-1] = 0.0f; // zero pad arrays

for (i = 0; i < NLP; i++) x4[i] = y4[i] * y4[i]; // actual
vector operations
```

# Exercise A: The vectorized Pi program

- Goal:
  - To understand the vector instructions in the kernel programming language
- Procedure:
  - Start with your best Pi program
  - Unroll the loops 4 times. Verify that the program still works
  - Use vector instructions in the body of the loop
- Expected output:
  - Output result plus an estimate of the error in the result
  - Report the runtime and compare vectorized and scalar versions of the program
  - You could try running this on the CPU as well as the GPU…

Appendix C

# THE OPENCL EVENT MODEL

# OpenCL Events

- An event is an object that communicates the status of commands in OpenCL … legal values for an event:
  - CL_QUEUED: command has been enqueued.
  - CL_SUBMITTED: command has been submitted to the compute device
  - CL_RUNNING: compute device is executing the command
  - CL_COMPLETE: command has completed
  - ERROR_CODE: a negative value indicates an error condition occurred.
- Can query the value of an event from the host … for example to track the progress of a command.

Examples:
- CL_EVENT_CONTEXT
- CL_EVENT_COMMAND_EXECUTION_STATUS
- CL_EVENT_COMMAND_TYPE



```
cl_int clGetEventInfo (
          cl_event event,    cl_event_info param_name,
          size_t param_value_size, void *param_value,
          size_t *param_value_size_ret)
```

# Generating and consuming events

- Consider the command to enqueue a kernel. The last three arguments optionally expose events (NULL otherwise).

```
cl_int clEnqueueNDRangeKernel (
  cl_command_queue command_queue,
  cl_kernel kernel,
  cl_uint work_dim,
  const size_t *global_work_offset,
  const size_t *global_work_size,
  const size_t *local_work_size,
  cl_uint num_events_in_wait_list,
  const cl_event *event_wait_list,
  cl_event *event)
```

Number of events this command is waiting to complete before executing

Array of pointers to the events being waited upon … Command queue and events must share a context.

Pointer to an event object generated by this command

# Event: basic event usage

- Events can be used to impose order constraints on kernel execution.
- Very useful with out-of-order queues.

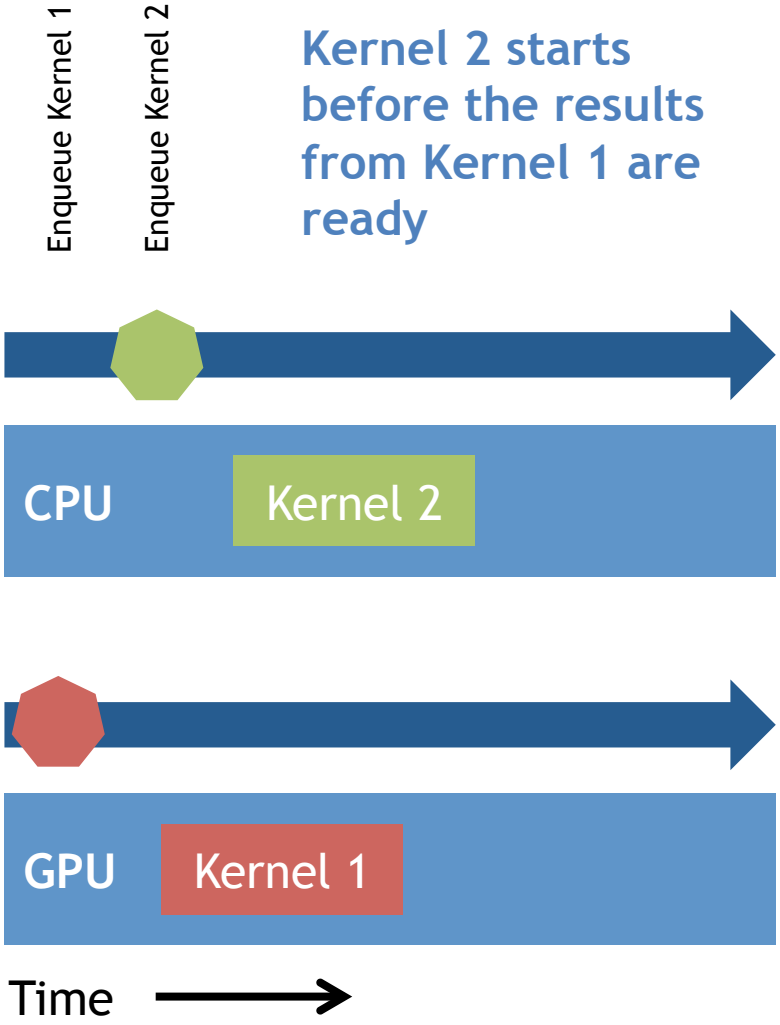
```
cl_event    k_events[2];

err = clEnqueueNDRangeKernel(commands, kernel1, 1,
        NULL, &global, &local, 0, NULL, &k_events[0]);

err = clEnqueueNDRangeKernel(commands, kernel2, 1,
        NULL, &global, &local, 0, NULL, &k_events[1]);

err = clEnqueueNDRangeKernel(commands, kernel3, 1,
        NULL, &global, &local, 2, k_events, NULL);
```

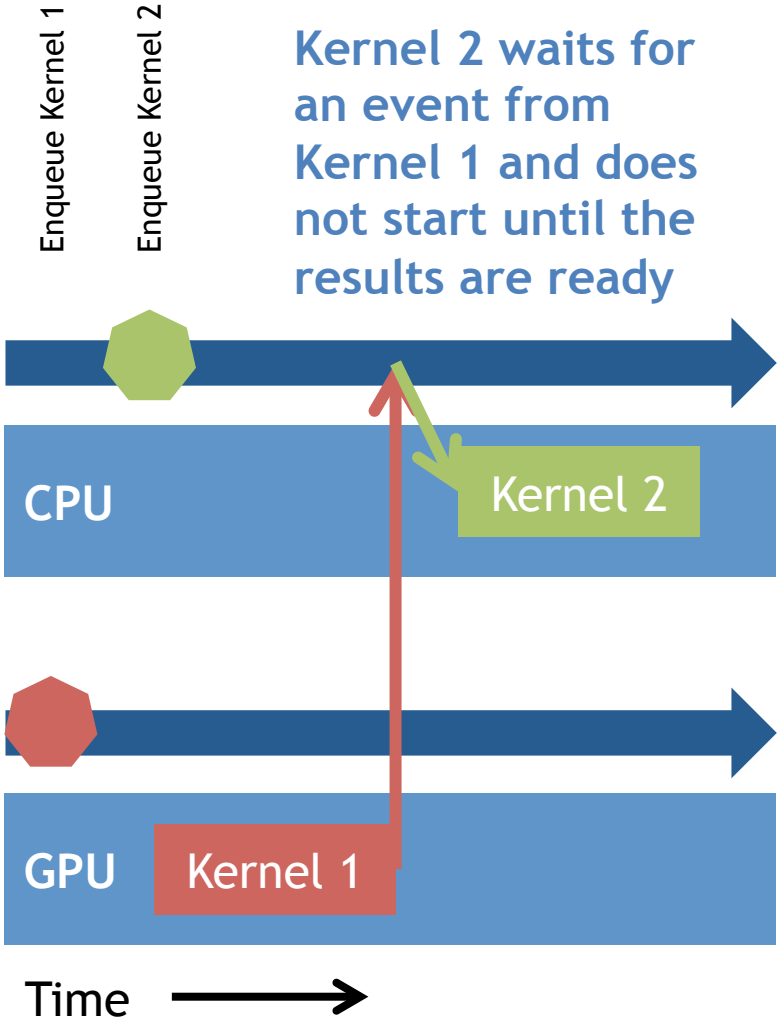
Enqueue two kernels that expose events

Wait to execute until two previous events complete

# OpenCL synchronization: queues & events

- Events connect command invocations. Can be used to synchronize executions inside out-of-order queues or between queues
- Example: 2 queues with 2 devices



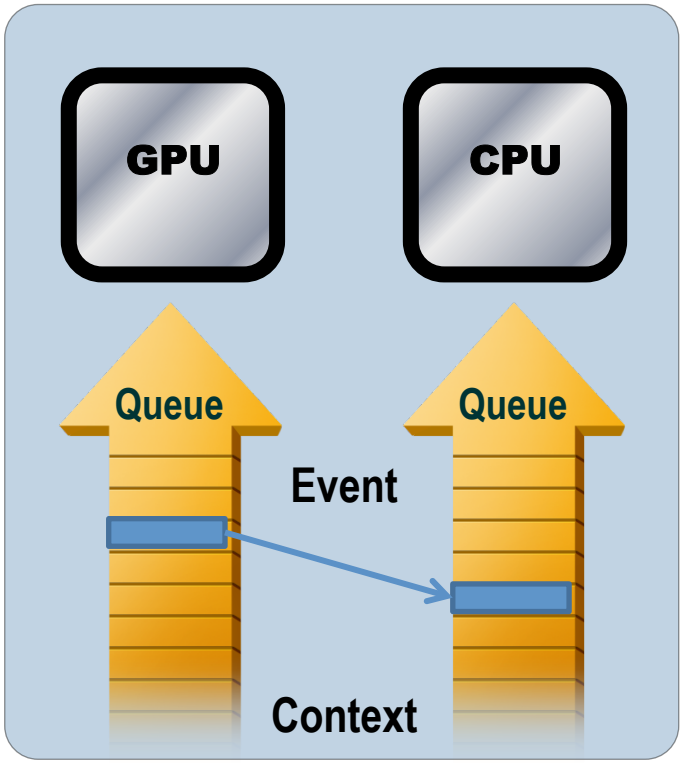



# Why Events? Won't a barrier do?

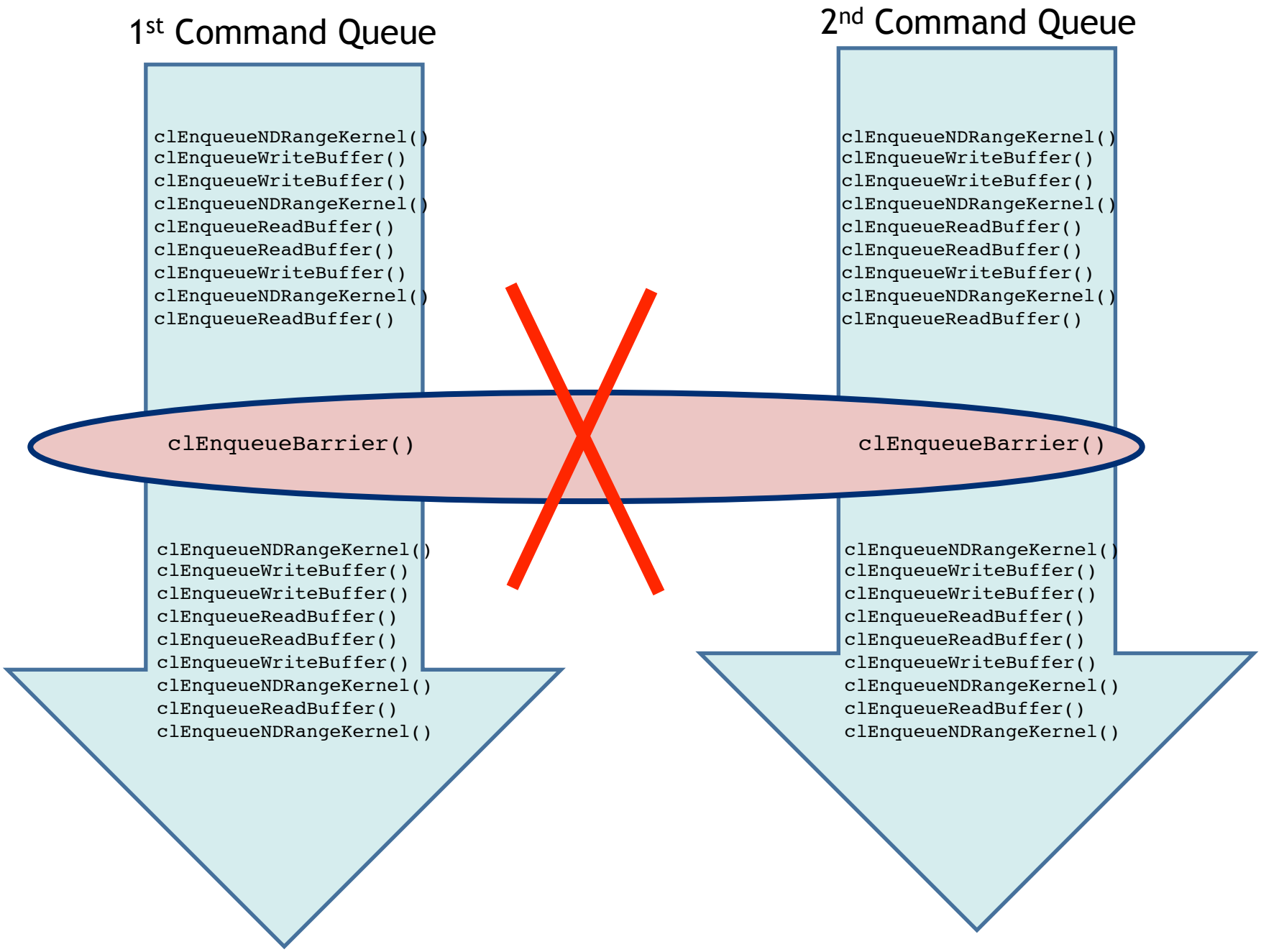
- A barrier defines a synchronization point … commands following a barrier wait to execute until all prior enqueued commands complete

```
cl_int
clEnqueueBarrier(cl_command_queue
queue)
```

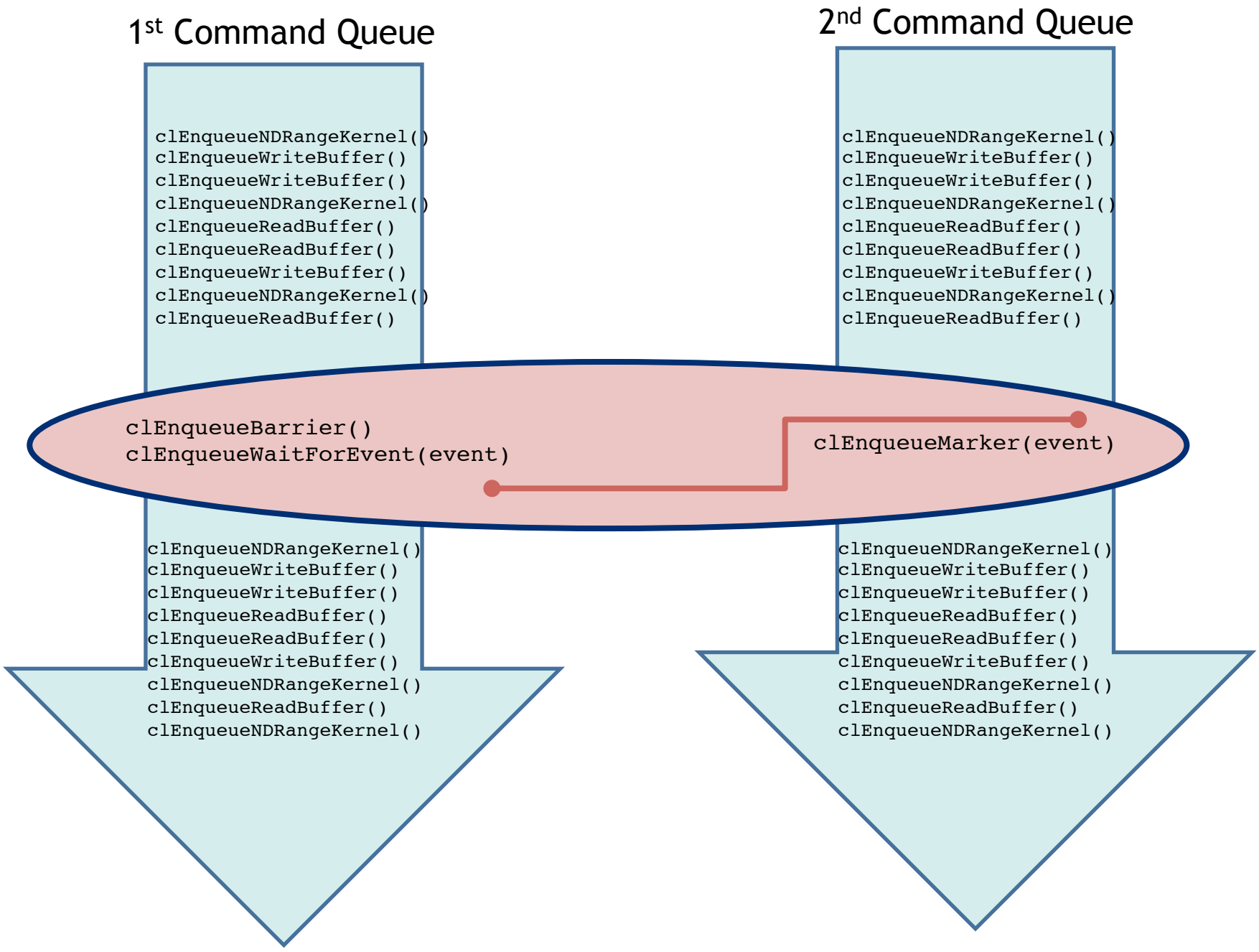
- Events provide fine grained control … this can really matter with an out-of-order queue.
- Events work between commands in the different queues … as long as they share a context
- Events convey more information than a barrier … provide info on state of a command, not just whether it's complete or not.



# Barriers between queues: clEnqueueBarrier doesn't work



# Barriers between queues: this works!



# Host generated events influencing execution of commands: User events

- “user code” running on a host thread can generate event objects

```
cl_event clCreateUserEvent(cl_context context, cl_int *errcode_ret)
```

- Created with value CL_SUBMITTED.
- It’s just another event to enqueued commands.
- Can set the event to one of the legal event values

```
cl_int clSetUserEventStatus(cl_event event, cl_int execution_status)
```

- Example use case: Queue up block of commands that wait on user input to finalize state of memory objects before proceeding.

# Command generated events influencing execution of host code

- A thread running on the host can pause waiting on a list of events to complete. This can be done with the function:

```
cl_int clWaitForEvents(
   cl_uint num_events,
   const cl_event *event_list)
```

Number of events to wait on

An array of pointers to event object

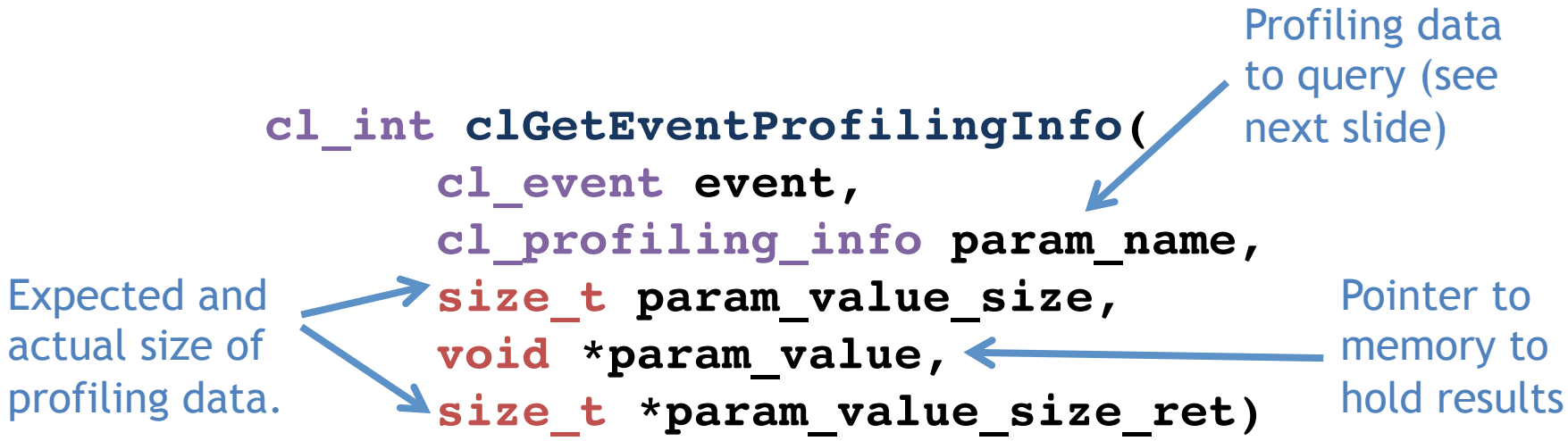
- Example use case: Host code waiting for an event to complete before extracting information from the event.

# Profiling with Events

- OpenCL is a performance oriented language … Hence performance analysis is an essential part of OpenCL programming.
- The OpenCL specification defines a portable way to collect profiling data.
- Can be used with most commands placed on the command queue … includes:
  - Commands to read, write, map or copy memory objects
  - Commands to enqueue kernels, tasks, and native kernels
  - Commands to Acquire or Release OpenGL objects
- Profiling works by turning an event into an opaque object to hold timing data.

# Using the Profiling interface

- Profiling is enabled when a queue is created with the CL_QUEUE_PROFILING_ENABLE flag set.
- When profiling is enabled, the following function is used to extract the timing data



# cl_profiling_info values

- CL_PROFILING_COMMAND_QUEUED
  - the device time in nanoseconds when the command is enqueued in a command-queue by the host. (cl_ulong)
- CL_PROFILING_COMMAND_SUBMIT
  - the device time in nanoseconds when the command is submitted to compute device. (cl_ulong)
- CL_PROFILING_COMMAND_START
  - the device time in nanoseconds when the command starts execution on the device. (cl_ulong)
- CL_PROFILING_COMMAND_END
  - the device time in nanoseconds when the command has finished execution on the device. (cl_ulong)

# Profiling Examples (C)

```
cl_event prof_event;
cl_command_queue comm;

comm = clCreateCommandQueue(
       context, device_id,
      CL_QUEUE_PROFILING_ENABLE,
       &err);

err = clEnqueueNDRangeKernel(
          comm, kernel,
          nd, NULL, global, NULL,
          0, NULL, prof_event);

clFinish(comm);
err = clWaitForEvents(1,
&prof_event );
```

```
cl_ulong start_time, end_time;
size_t return_bytes;

err = clGetEventProfilingInfo(
         prof_event,
        CL_PROFILING_COMMAND_QUEUED,
          sizeof(cl_ulong),
          &start_time,
          &return_bytes);

err = clGetEventProfilingInfo(
          prof_event,
          CL_PROFILING_COMMAND_END,
          sizeof(cl_ulong),
          &end_time,
          &return_bytes);

run_time =(double)(end_time -
start_time);
```

# Events inside Kernels … Async. copy

```
// A, B, C kernel args … global  buffers.
// Bwrk is a local buffer

for(k=0;k<Pdim;k++)
    Awrk[k] = A[i*Ndim+k];

for(j=0;j<Mdim;j++){
    event_t ev_cp  = async_work_group_copy(
        (__local float*) Bwrk, (__global float*) B,
        (size_t) Pdim, (event_t) 0);

    wait_group_events(1, &ev_cp);

    for(k=0, tmp= 0.0;k<Pdim;k++)
            tmp  += Awrk[k] *  Bwrk[k];
    C[i*Ndim+j] = tmp;
}
```

- Compute a row of C = A * B
  - 1 A column per work-item
  - Work group shares rows of B

Start an async. copy for row of B returning an event to track progress.

Wait for async. copy to complete before proceeding.

Compute element of C using A from private memory and B from local memory.

# Events and the C++ interface (for profiling)

- Enqueue the kernel with a returned event

```
Event event = vadd( EnqueueArgs(commands,
        NDRange(count), NDRange(local)), a_in,
b_in, c_out, count);
```

- What for the command attached to the event to complete

```
event.wait();
```

- Extract timing data from the event:

```
cl_ulong ev_start_time =
 event.getProfilingInfo<CL_PROFILING_COMMAND_START>();

cl_ulong ev_end_time =
 event.getProfilingInfo<CL_PROFILING_COMMAND_END>();
```

Appendix D

# C++ FOR C PROGRAMMERS

# C++ for C programmers

- This Appendix shows and highlights some of the basic features and principles of C++.
- It is intended for the working C programmer.
- The C++ standards:
  - ISO/ANSI Standard 1998 (revision 2003)
  - ISO/ANSI Standard 2011 (aka C++0x or C++11)

# Comments, includes, and variable definitions

- Single line comments:

```
// this is a C++ comment
```

- C includes are prefixed with “c”:

```
#include <cstdio>
```

- I/O from keyboard and to console

```
#include <iosteam>
int a;  // variables can be declared inline
std::cin >> a;  // input integer to a
std::cout << a; // outputs ‘a’ to console
```

# Namespaces

- Definitions and variables can be scoped with namespaces.
  :: is used to dereference.
- Using namespace opens names space into current scope.
- Default namespace is std.

```
#include <iostream> // definitions in std namespace

namespace foo {
            int id(int x) { return x; }
          };

int x = foo::id(10);
using namespace std;
cout << x; // no need to prefix with std::
```

# References in C++ … a safer way to do pointers

- References are non-null pointers. Since they can't be NULL, you don't have to check for NULL value all the time (as you do with C)
- For example, in C we need to write:

```
int foo(int * x)  {
     if (x != NULL) return *x;
     else return 0;
}
```

- In C++ we could write:

```
int foo(int & x)  {
          return x;
}
```

- Note that in both cases the memory address of x is passed (i.e. by reference) and not the value!

# New/Delete Memory allocation

- C++ provides safe(r) memory allocation
- new and delete operator are defined for each type, including user defined types. No need to multiple by sizeof(type) as in C.

```
int * x = new int;
delete x;
```

- For multi element allocation (i.e. arrays) we must use delete[].

```
int * array = new int[100];
delete[] array;
```

# Overloading

- C++ allows functions to have the same name but with different argument types.

```
int add(int x, int y) {
   return x+y;
}
float add(float x, float y) {
   return x+y;
}
float  f = add(10.4f, 5.0f);
// calls the float version of add
int i = add(100,20);
// calls the int version of add
```

# Classes (and structs)

- C++ classes are an extension of C structs (and unions) that can functions (called member functions) as well as data.

```
class Vector {
  private:
    int x_, y_, z_ ;
  public:
  Vector (int x,int y,int z): x_(x),y_(y),z_(z){}// constructor

  ~Vector // destructor
  {
       cout << "vector destructor";
  }
  int getX() const { return x_; } // access member function
        …
};
```

The keyword "const" can be applied to member functions such as getX() to state that the particular member function will not modify the internal state of the object, i.e it will not cause any visual effects to someone owning a pointer to the said object. This allows for the compiler to report errors if this is not the case, better static analysis, and to optimize uses of the object , i.e. promote it to a register or set of registers.

# More information about constructors

- Consider the constructor from the previous slide …

```
Vector (int x, int y, int z): x_(x), y_(y), z_(z) {}
```

- C++ member data local to a class (or struct) can be initialized using the noation

```
: data_name(initializer_name), ...
```

- Consider the following two semantically equivalent structs in which the constructor sets the data member x_ to the input value x:

A

```
struct Foo
{
  int x_;
  Foo(int x) : x_(x) {}
}
```

B

```
struct Foo
{
  int x_;
  Foo(int x) { x_ = x; }
}
```

- Case B must use a temporary to read the value of x, while this is not so for Case A. This is due to C's definition of local stack allocation.
- This turns out to be very import in C++11 with its memory model which states that an object is said to exist once inside the body of the constructor and hence thread safety becomes an issue, this is not the case for the constructor initalization list (case A). This means that safe double locking and similar idioms can be implemented using this approach.

# Classes (and structs) continued

- Consider the following block where we construct an object (the vector “v”), use it and then reach the end of the block

```
{
   Vector v(10,20,30);
   // vector {x_ = 10, y_ = 20 , z_ = 30}
   // use v
} // at this point v's destructor would be called!
```

- Note that at the end of the block, v is no longer accessible and hence can be destroyed. At this point, the destructor for v is called.

# Classes (and structs) continued

- There is a lot more to classes, e.g. inheritance but it is all based on this basic notion.

- The previous examples adds no additional data or overhead to a traditional C struct, it has just improved software composibility.

# Function objects

- Function application operator can be overloaded to define functor classes

```
struct Functor
  {
       int operator() (int x) { return x*x; }
  };

Functor f(); // create an object of type Functor
int value = f(10); // call the operator()
```

# Template functions

- Don’t want to write the same function many times for different types?
- Templates allow functions to be parameterized with a type(s).

```
template<typename T>
    T add(T x, T y) { return x+y; }

    float  f = add<float>(10.4f, 5.0f); // float version
    int i = add<int>(100,20);           // int version
```

- You can use the templatized type, T, inside the template function

# Template classes

- Don’t want to write the same class many times for different types?
- Templates allow class to be parameterized with a type(s) too.

```
template <typename T>
    class Square
    {
      T operator() (T x) { return x*x; }
    };
    Square<int> f_int();
    int value = f_int(10);
```

# C++11 defines a function template

- C++ function objects can be stored in the templated class std::function. The following header defines the class std::function

```
#include <functional>
```

- We can define a C++ function object (e.g. functor) and then store it in the tempated class std::function

```
struct Functor
    {
        int operator() (int x) { return x*x; }
    };
    std::function<int (int)> square(Functor());
```

# C++ function template: example 1

The header <functional> just defines the template std::function. This can be used to warp standard functions or function objects, e.g.:

```
int foo(int x)  { return x; }  // standard function
std::function<int (int)> foo_wrapper(foo);

struct Foo // function object
{
  void operator()(int x)
    { return x;}
};
 std::function<int (int)> foo_functor(Foo());
```

foo_functor and foo_wrapper are basically the same but one is using a standard C like function, while the other is using a function object

# C++ function template: example 2

What is the point of function objects? Well they can of course contain local state, which functions cannot, they can also contain member functions and so on. A silly example might be:

```
struct Foo // function object
{          int y_;
           Foo() : y_(100) {}

          void operator()(int x)
            { return x+100; }
};

std::function<int (int)> add100(Foo());
// function that adds 100 to its argument
```

Appendix E

# MEMORY COALESCENCE:
# MATRIX MULTIPLICATION CASE STUDY

# Performance issues with matrix multiplication

- Consider the following version of the blocked matrix multiplication kernel from exercise 9.

# Blocked matrix multiply: kernel

```
#define blksz 16
__kernel void mmul(
              const unsigned int N,
              __global float* A,
              __global float* B,
              __global float* C,
              __local  float* Awrk,
              __local  float* Bwrk)
{
  int kloc, Kblk;
  float Ctmp=0.0f;

  //  compute element C(i,j)
  int i = get_global_id(0);
  int j = get_global_id(1);

  // Element C(i,j) is in block C(Iblk,Jblk)
  int Iblk = get_group_id(0);
  int Jblk = get_group_id(1);

  // C(i,j) is element C(iloc, jloc)
  //  of block C(Iblk, Jblk)
  int iloc = get_local_id(0);
  int jloc = get_local_id(1);
  int Num_BLK = N/blksz;

  // upper-left-corner and inc for A and B
  int Abase = Iblk*N*blksz;       int Ainc  = blksz;
  int Bbase = Jblk*blksz;    int Binc  = blksz*N;

// C(Iblk,Jblk) = (sum over Kblk) A(Iblk,Kblk)*B(Kblk,Jblk)
  for (Kblk = 0;  Kblk<Num_BLK;  Kblk++)
  {
     //Load A(Iblk,Kblk) and B(Kblk,Jblk).
     //Each work-item loads a single element of the two
     //blocks which are shared with the entire work-group

     Awrk[iloc*blksz+jloc] = A[Abase+iloc*N+jloc];
     Bwrk[iloc*blksz+jloc] = B[Bbase+iloc*N+jloc];

     barrier(CLK_LOCAL_MEM_FENCE);

     #pragma unroll
     for(kloc=0; kloc<blksz; kloc++)
        Ctmp+=Awrk[jloc*blksz+kloc]*Bwrk[kloc*blksz+iloc];

     barrier(CLK_LOCAL_MEM_FENCE);
     Abase += Ainc;       Bbase += Binc;
   }
   C[j*N+i] = Ctmp;
}
```

# Blocked matrix multiply: kernel (performance bug)

```
#define blksz 16
__kernel void mmul(
              const unsigned int N,
              __global float* A,
              __global float* B,
              __global float* C,
              __local  float* Awrk,
              __local  float* Bwrk)
{
  int kloc, Kblk;
  float Ctmp=0.0f;

  int jloc = get_local_id(1);
  int Num_BLK = N/blksz;

  // upper-left-corner and inc for A and B
  int Abase = Iblk*N*blksz;       int Ainc  = blksz;
  int Bbase = Jblk*blksz;    int Binc  = blksz*N;

// C(Iblk,Jblk) = (sum over Kblk) A(Iblk,Kblk)*B(Kblk,Jblk)
  for (Kblk = 0;  Kblk<Num_BLK;  Kblk++)
  {
     //Load A(Iblk,Kblk) and B(Kblk,Jblk).
     //Each work-item loads a single element of the two
     //blocks which are shared with the entire work-group

     Awrk[iloc*blksz+jloc] = A[Abase+iloc*N+jloc];
     Bwrk[iloc*blksz+jloc] = B[Bbase+iloc*N+jloc];

     barrier(CLK_LOCAL_MEM_FENCE);

     #pragma unroll
     for(kloc=0; kloc<blksz; kloc++)
       Ctmp+=Awrk[jloc*blksz+kloc]*Bwrk[kloc*blksz+iloc];

     barrier(CLK_LOCAL_MEM_FENCE);
     Abase += Ainc;       Bbase += Binc;
   }
   C[j*N+i] = Ctmp;
}
```

Note that the pattern of indices loaded differ from the patterned used.

This mistake means that the memory is not coalesced. Performance was around 76695.8 MFLOPS on an NVIDIA M2090 GPU. We were expecting almost twice that many FLOPS.

# Blocked matrix multiply: kernel (fixed)

```
#define blksz 16
__kernel void mmul(
              const unsigned int N,
              __global float* A,
              __global float* B,
              __global float* C,
              __local  float* Awrk,
              __local  float* Bwrk)
{
   int kloc, Kblk;
   float Ctmp=0.0f;

   //
   in
   in

   //                                                    Jblk)
   in
   in

   //
   //
   in
   in
   int Num_BLK = N/blksz;
```

We fixed this by making sure the pattern of indices on the load matched the later block of code where we used these arrays.

With that small change, the performance on an NVIDIA M2090 GPU hit the expected value of around 119304.6 MGFLOPS.

```
 // upper-left-corner and inc for A and B
 int Abase = Iblk*N*blksz;       int Ainc  = blksz;
 int Bbase = Jblk*blksz;    int Binc  = blksz*N;

// C(Iblk,Jblk) = (sum over Kblk) A(Iblk,Kblk)*B(Kblk,Jblk)
 for (Kblk = 0;  Kblk<Num_BLK;  Kblk++)
 {
    //Load A(Iblk,Kblk) and B(Kblk,Jblk).
    //Each work-item loads a single element of the two
    //blocks which are shared with the entire work-group

    Awrk[jloc*blksz+iloc] = A[Abase+jloc*N+iloc];
    Bwrk[jloc*blksz+iloc] = B[Bbase+jloc*N+iloc];

    barrier(CLK_LOCAL_MEM_FENCE);

    #pragma unroll
    for(kloc=0; kloc<blksz; kloc++)
      Ctmp+=Awrk[jloc*blksz+kloc]*Bwrk[kloc*blksz+iloc];

    barrier(CLK_LOCAL_MEM_FENCE);
    Abase += Ainc;       Bbase += Binc;
 }
 C[j*N+i] = Ctmp;
}
```

# Performance issues with matrix multiplication

- This is a good object lesson on the importance of paying attention to memory coalescence.
- How did I make this mistake? Getting the indices right in this code was tough. I developed my code on a CPU. On a CPU, the effect was not apparent … I got the expected performance with the memory coalescence bug when running on a CPU. IT only shows up on the GPU.
- This points to the importance of exploring a range of platforms during the debug and optimization phase of software development.
- Still, after making the change on my CPU, the performance went from 9.8 GFLOPS to 12 GFLOPS.