Parallel Programming

Data-Parallel Programming with OpenCL

Professor Yi-Ping You (游逸平)
Department of Computer Science
http://www.cs.nctu.edu.tw/~ypyou/

Acknowledgement

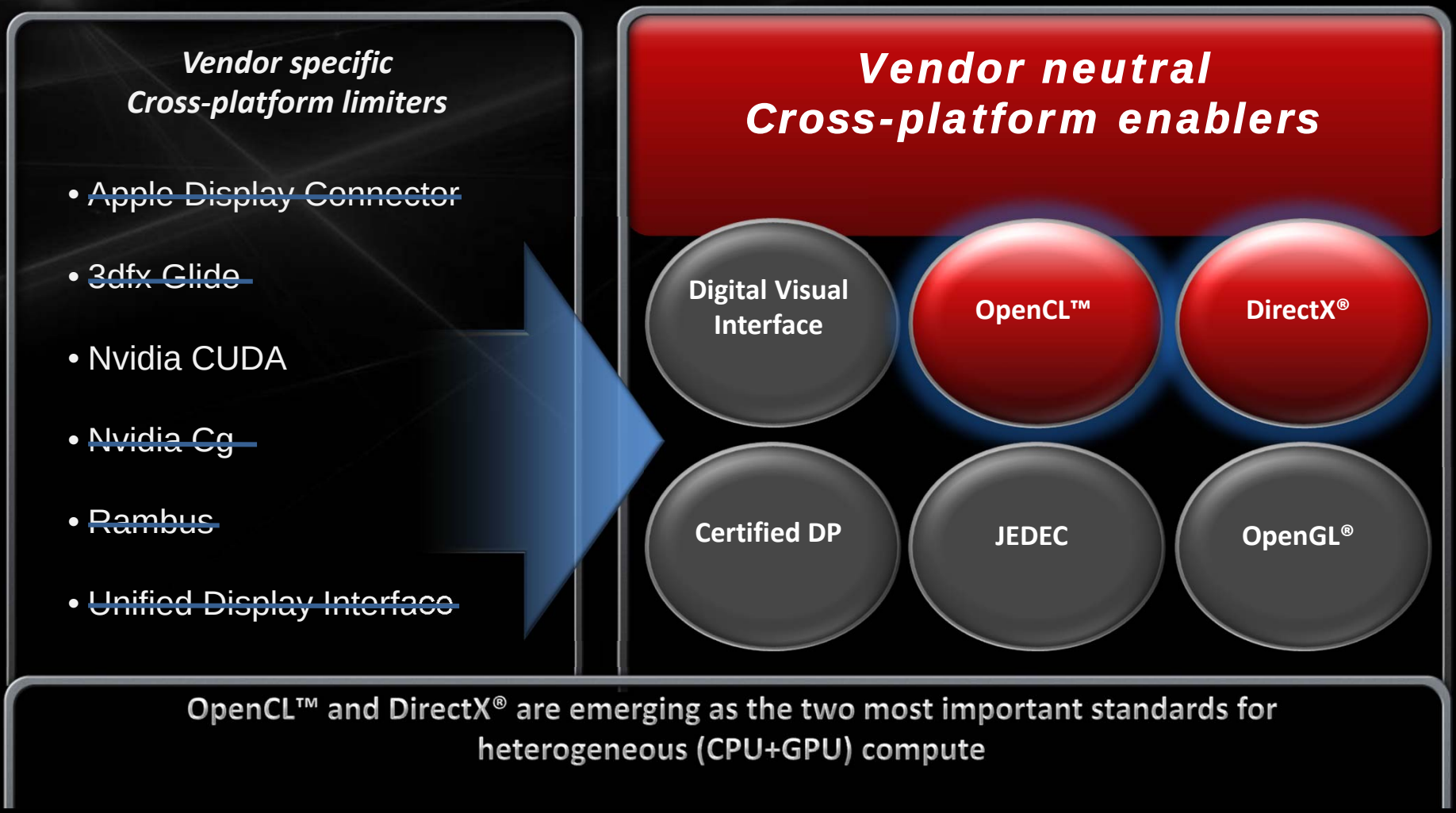
- This set of slides is adopted from
  - Introductory Tutorial to OpenCL™ for HPC at SAAHPC'10
  - Lectures slides from the Embedded Software Consortium (ESW Consortium)

Outline

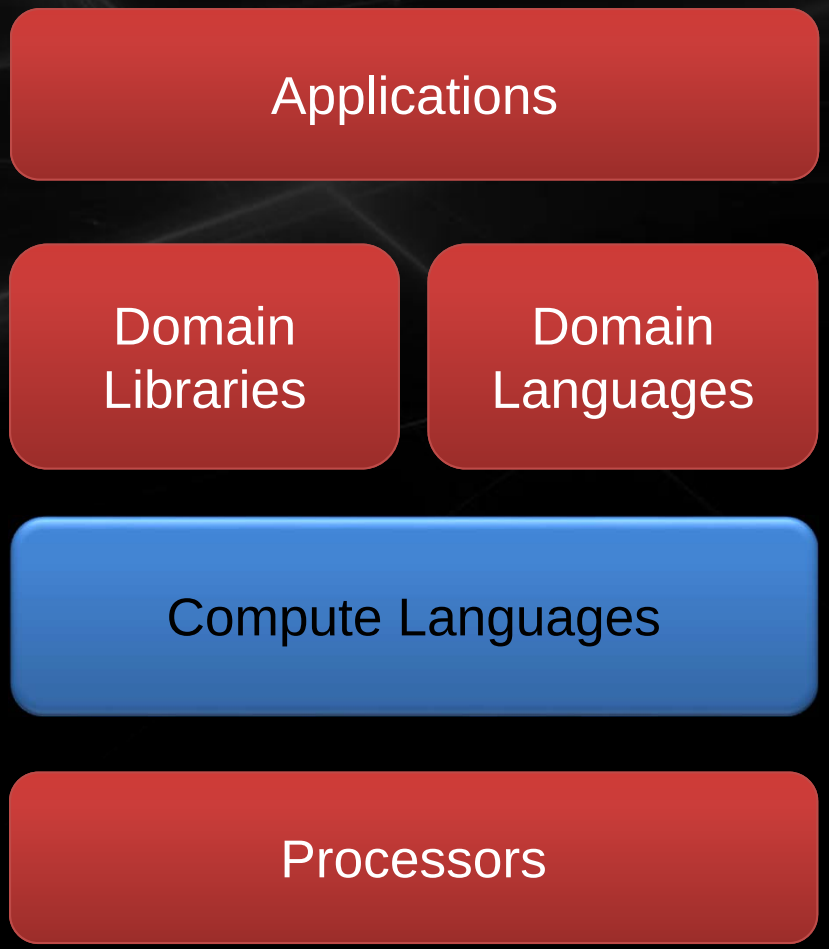
- What is OpenCL?
- OpenCL Architecture
  - Platform Model
  - Execution Model
  - Memory Model
  - Programming Model
- Case Studies
  - Image Rotation
  - Matrix Multiplication
- Memory Issues
  - Coalescing memory accesses
  - Memory bank conflicts
- Synchronization
  - Within a work group
  - Among commands in command queue(s)
- Profiling Using Events

Open Standards for GPU Programming

Maximize Developer Freedom and Addressable Market



Where do OpenCL™ and DirectCompute fit?



Imaging Processing, Video Processing, Games, Technical Computing, etc.

ACML, D3DCSX, Bullet Physics, MKL, cuFFT, Brook+, etc.

DirectCompute, OpenCL™, CUDA, etc.

CPU, GPU

DirectCompute and OpenCL™

OpenCL™ - Standard API from Khronos

- Supports multiple platforms: Windows®, Linux®, Mac
- Resource limits vary per implementation
- Optional sharing of objects with OpenGL® and DirectX® 9/10/11
- Resources are not all known at compile time
- Vendors can add extensions

DirectCompute - part of DirectX® 11 API

- Supports Windows® (Win 7 and Vista® via 7IP)
- Resource limits fixed
- Integrated with DirectX® Graphics with shared resources
- All resources known at compile time
- No extensions. Capabilities linked to DirectX® version support.

OpenCL Working Group Members

- Diverse industry participation – many industry experts
  - Processor vendors, system OEMs, middleware vendors, application developers
  - Academia and research labs, FPGA vendors
- NVIDIA is chair, Apple is specification editor

OpenCL™

With OpenCL™ you can…

- Leverage CPUs and GPUs to accelerate parallel computation
- Get dramatic speedups for computationally intensive applications
- Write accelerated portable code across different devices and architectures

With AMD's implementations you can…

- Leverage CPUs, AMD's GPUs, to accelerate parallel computation
- OpenCL 2.1 Public release for multi-core CPU and AMD's GPU's April 2010
- The closely related DirectX® 11 public release supporting DirectCompute on AMD GPUs in October 2009, as part of Win7 Launch*

*DirectX 11® currently supported by the following products: ATI Radeon ™ HD 5900, HD 5800, and HD 5700 Series, and ATI Mobility Radeon ™ HD 5800, HD 5700, HD 5600, and HD 5400 Series

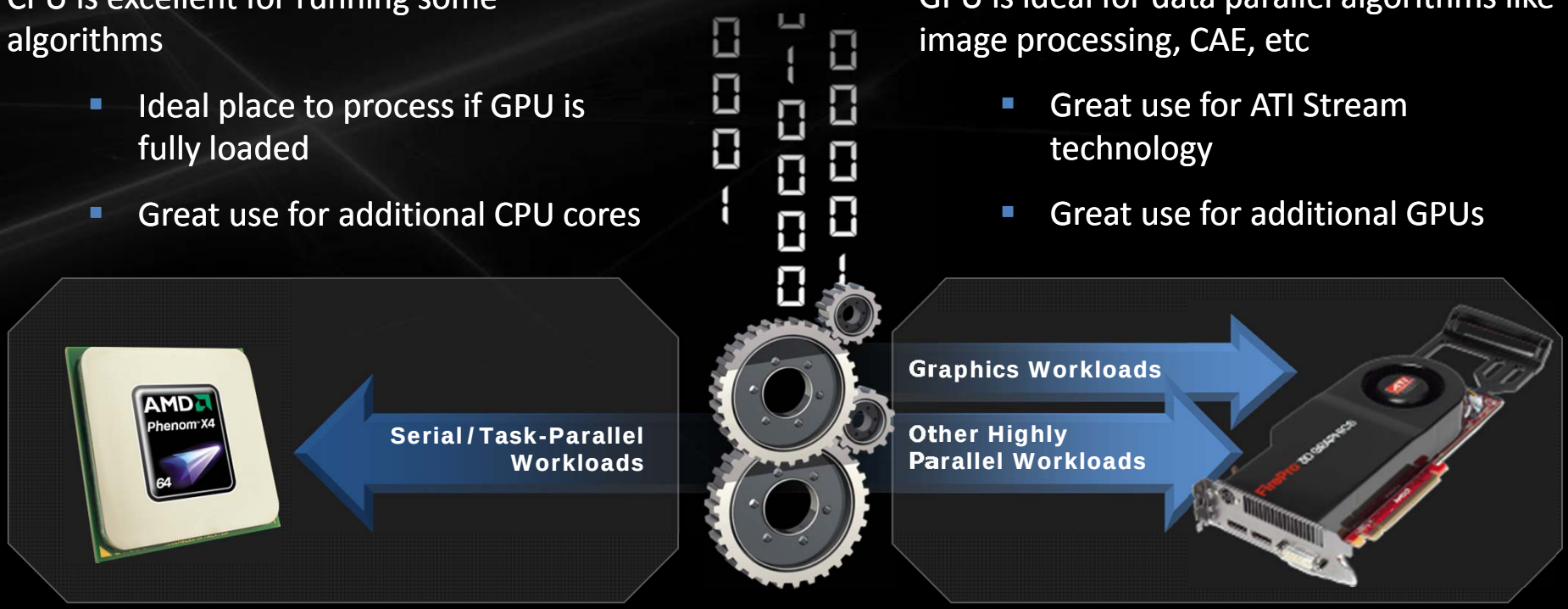
AMD Balanced Platform Advantage

CPU is excellent for running some algorithms

- Ideal place to process if GPU is fully loaded
- Great use for additional CPU cores

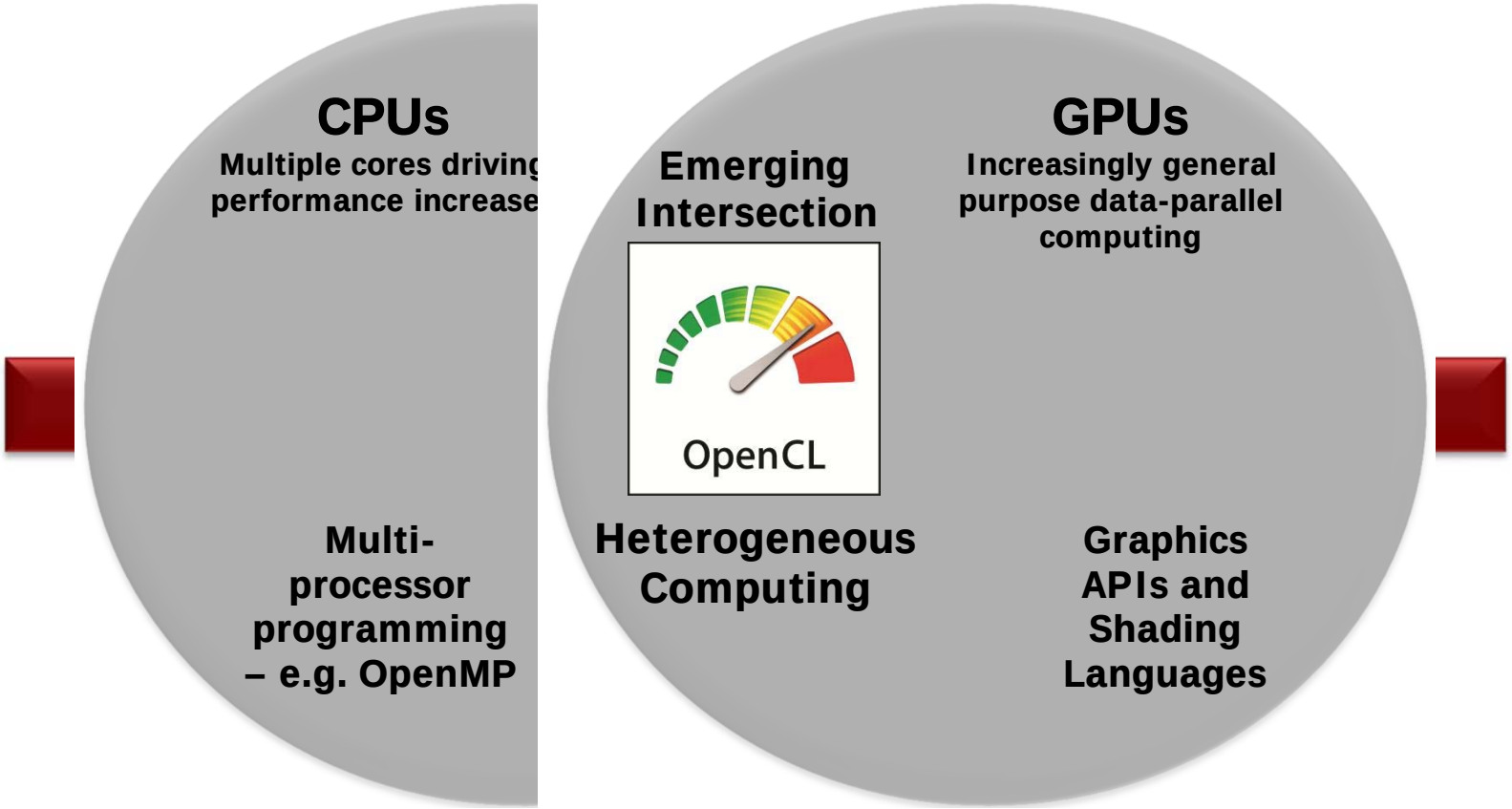
GPU is ideal for data parallel algorithms like image processing, CAE, etc

- Great use for ATI Stream technology
- Great use for additional GPUs



Delivers **optimal performance** for a wide range of platform configurations

Processor Parallelism



OpenCL is a programming framework for heterogeneous compute resources

Outline

- What is OpenCL?
- OpenCL Architecture
  - Platform Model
  - Execution Model
  - Memory Model
  - Programming Model
- Case Studies
  - Image Rotation
  - Matrix Multiplication
- Memory Issues
  - Coalescing memory accesses
  - Memory bank conflicts
- Synchronization
  - Within a work group
  - Among commands in command queue(s)
- Profiling Using Events

OpenCL Architecture

- OpenCL allows parallel computing on heterogeneous devices
  - CPUs, GPUs, other processors (Cell, DSPs, etc)
  - Provides portable accelerated code
- Defined in four parts
  - Platform Model
  - Execution Model
  - Memory Model
  - Programming Model

OpenCL

Outline

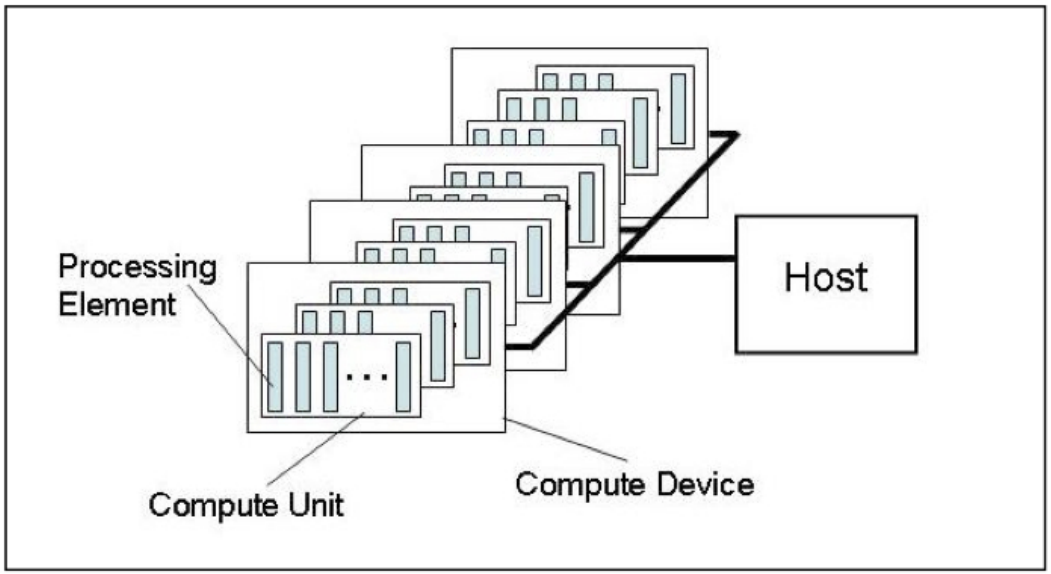
- What is OpenCL?
- OpenCL Architecture
  - Platform Model
  - Execution Model
  - Memory Model
  - Programming Model
- Case Studies
  - Image Rotation
  - Matrix Multiplication
- Memory Issues
  - Coalescing memory accesses
  - Memory bank conflicts
- Synchronization
  - Within a work group
  - Among commands in command queue(s)
- Profiling Using Events

Platforms

- Each OpenCL implementation (i.e. an OpenCL library from AMD, NVIDIA, etc.) defines *platforms* which enable the host system to interact with OpenCL-capable devices
  - Currently each vendor supplies only a single platform per implementation
- OpenCL uses an "Installable Client Driver" model
  - The goal is to allow platforms from different vendors to co-exist
  - Current systems' device driver model will not allow different vendors' GPUs to run at the same time

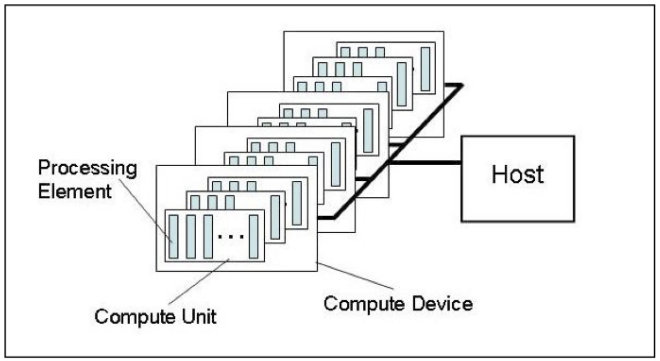
Platform Model

- The model consists of a host connected to one or more OpenCL devices
- A device is divided into one or more compute units
- Compute units are divided into one or more processing elements
  - Each processing element maintains its own program counter



Host/Devices

- The host is whatever the OpenCL library runs on
  - x86 CPUs for both NVIDIA and AMD
- Devices are processors that the library can talk to
  - CPUs, GPUs, and generic accelerators
- For AMD
  - All CPUs are combined into a single device (each core is a compute unit and processing element)
  - Each GPU is a separate device



Terminologies: CUDA vs OpenCL

CUDA term	OpenCL term
GPU	Device
Streaming multiprocessor (SM)	Compute unit (CU)
Stream processor (SP)	Processing element (PE)

Outline

- What is OpenCL?
- OpenCL Architecture
  - Platform Model
  - Execution Model
  - Memory Model
  - Programming Model
- Case Studies
  - Image Rotation
  - Matrix Multiplication
- Memory Issues
  - Coalescing memory accesses
  - Memory bank conflicts
- Synchronization
  - Within a work group
  - Among commands in command queue(s)
- Profiling Using Events

Selecting a Platform

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

- This function is usually called twice
  - The first call is used to get the number of platforms available to the implementation
  - Space is then allocated for the platform objects
  - The second call is used to retrieve the platform objects

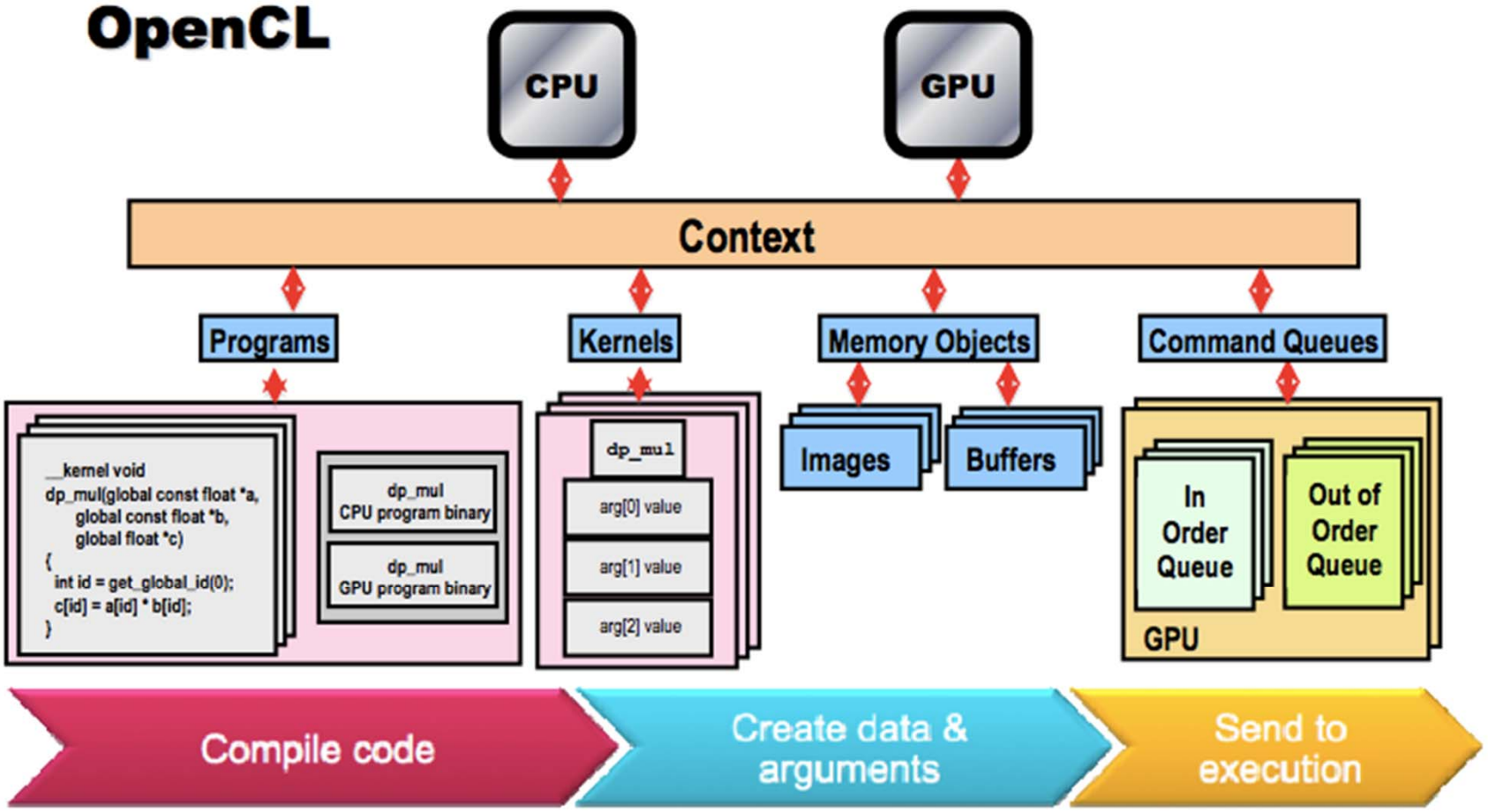
Selecting Devices

- Once a platform is selected, we can then query for the devices that it knows how to interact with

clGetDeviceIDs[4] (cl_platform_id *platform*,
cl_device_type *device_type*,
cl_uint *num_entries*,
cl_device_id **devices*,
cl_uint **num_devices*)

- We can specify which types of devices we are interested in (e.g. all devices, CPUs only, GPUs only)
- This call is performed twice as with clGetPlatformIDs
  - The first call is to determine the number of devices, the second retrieves the device objects

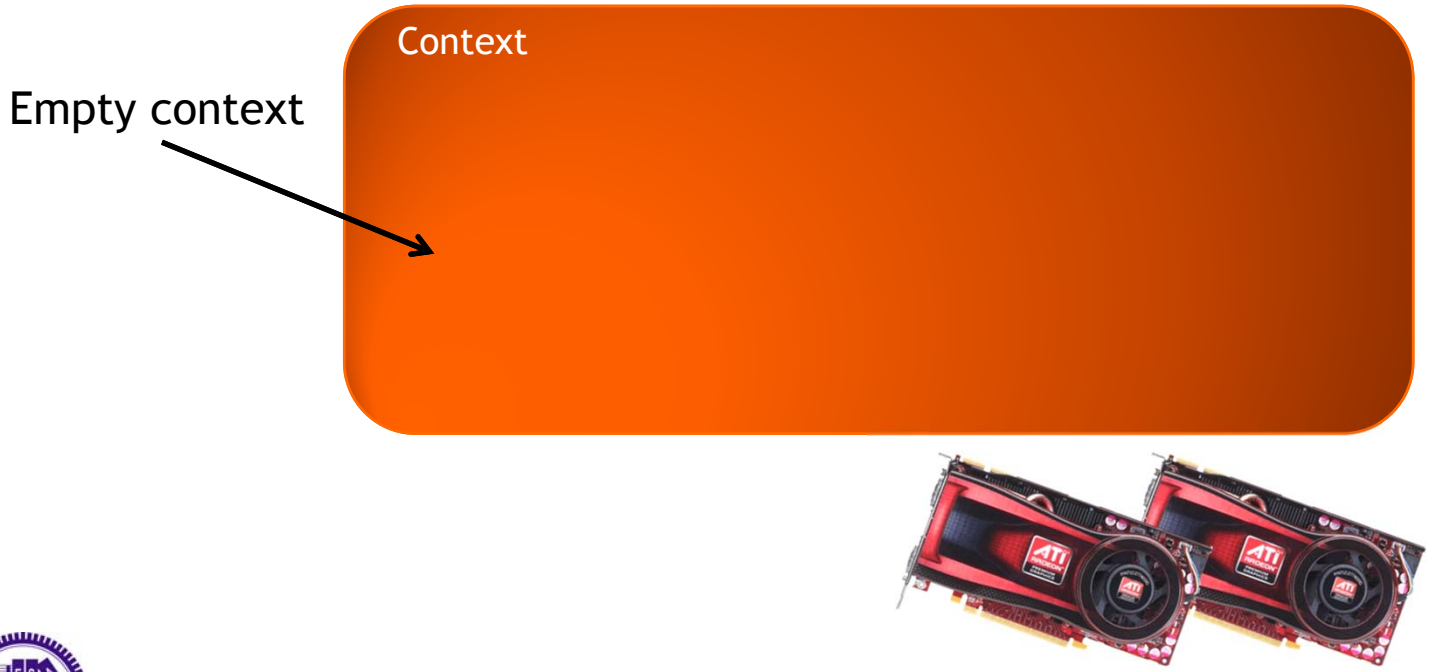
Big Picture



© Copyright Khronos Group, 2009 - Page 15

Contexts

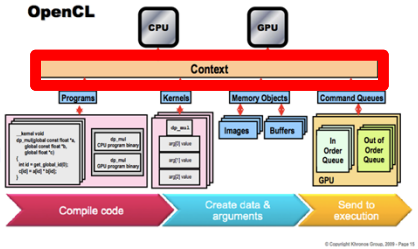
- A context refers to the environment for managing OpenCL objects and resources
- When you create a context, you will provide a list of devices to associate with it
  - For the rest of the OpenCL resources, you will associate them with the context as they are created



Contexts

- To manage OpenCL programs, the following are associated with a context
  - Devices: the things doing the execution
  - Program objects: the program source that implements the kernels
  - Kernels: functions that run on OpenCL devices
  - Memory objects: data that are operated on by the device
  - Command queues: mechanisms for interaction with the devices
    - Memory commands (data transfers)
    - Kernel execution
    - Synchronization

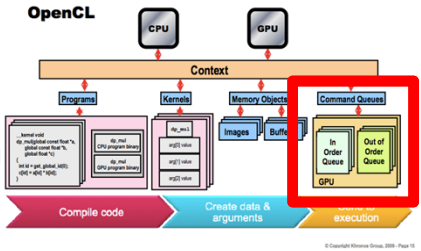
Contexts




```
cl_context    clCreateContext (const 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)
```

- This function creates a context given a list of devices
- The properties argument specifies which platform to use (if NULL, the default chosen by the vendor will be used)
- The function also provides a callback mechanism for reporting errors to the user

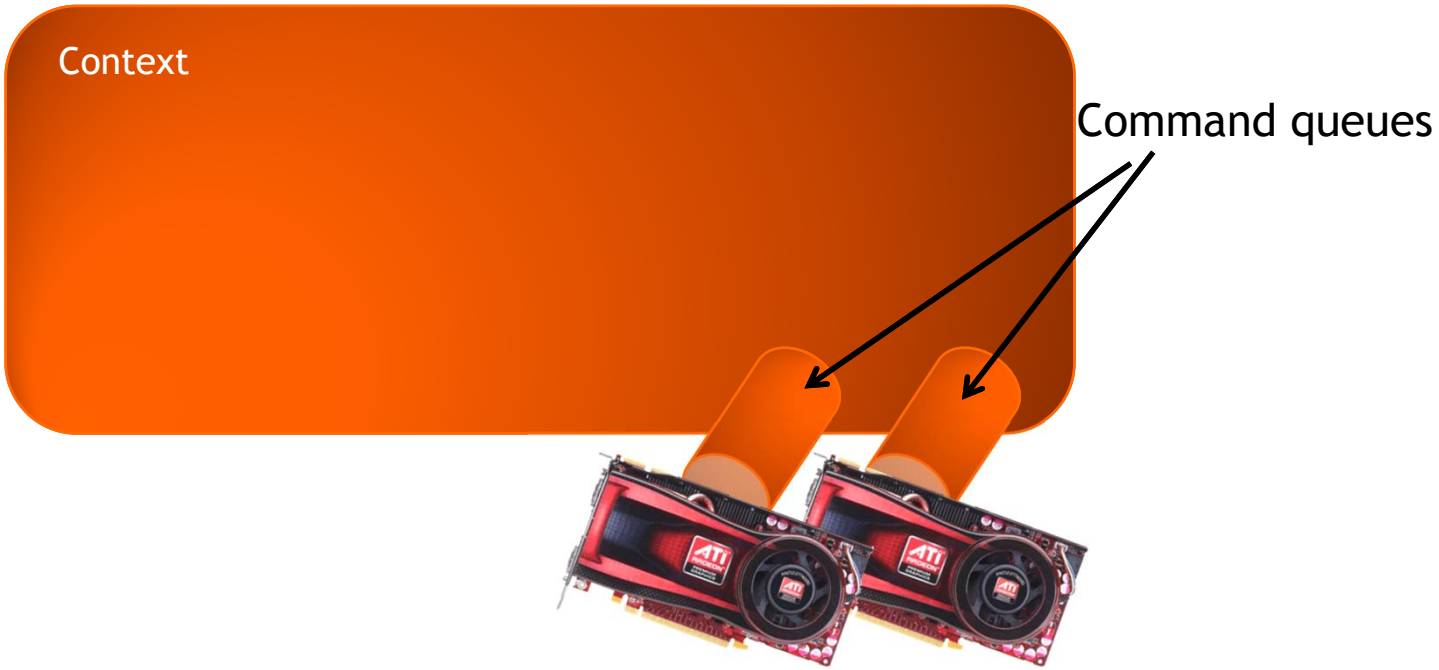
Command Queues



- A *command queue* is the mechanism for the host to request that an action be performed by the device
  - Perform a memory transfer, begin executing, etc.
- A separate command queue is required for each device
- Commands within the queue can be synchronous or asynchronous
- Commands can execute in-order or out-of-order

Command Queues

- Command queues associate a context with a device
  - Despite the figure below, they are not a physical connection

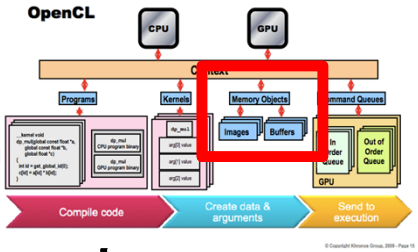


Command Queues

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

- A command queue establishes a relationship between a context and a device
- The command queue properties specify:
  - If out-of-order execution of commands is allowed
  - If profiling is enabled
    - Profiling is done using *events* (discussed later) and will create some overhead

Memory Objects



- Memory objects are OpenCL data that can be moved on and off devices
  - Objects are classified as either buffers or images
- Buffers
  - Contiguous chunks of memory – stored sequentially and can be accessed directly (arrays, pointers, structs)
  - Read/write capable
- Images
  - Opaque objects (2D or 3D)
  - Can only be accessed via read_image() and write_image()
  - Can either be read or written in a kernel, but not both

Creating Buffers

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

- This function creates a buffer (cl_mem object) for the given context
  - Images are more complex
- The flags specify:
  - the combination of reading and writing allowed on the data
  - if the host pointer itself should be used to store the data
  - if the data should be copied from the host pointer

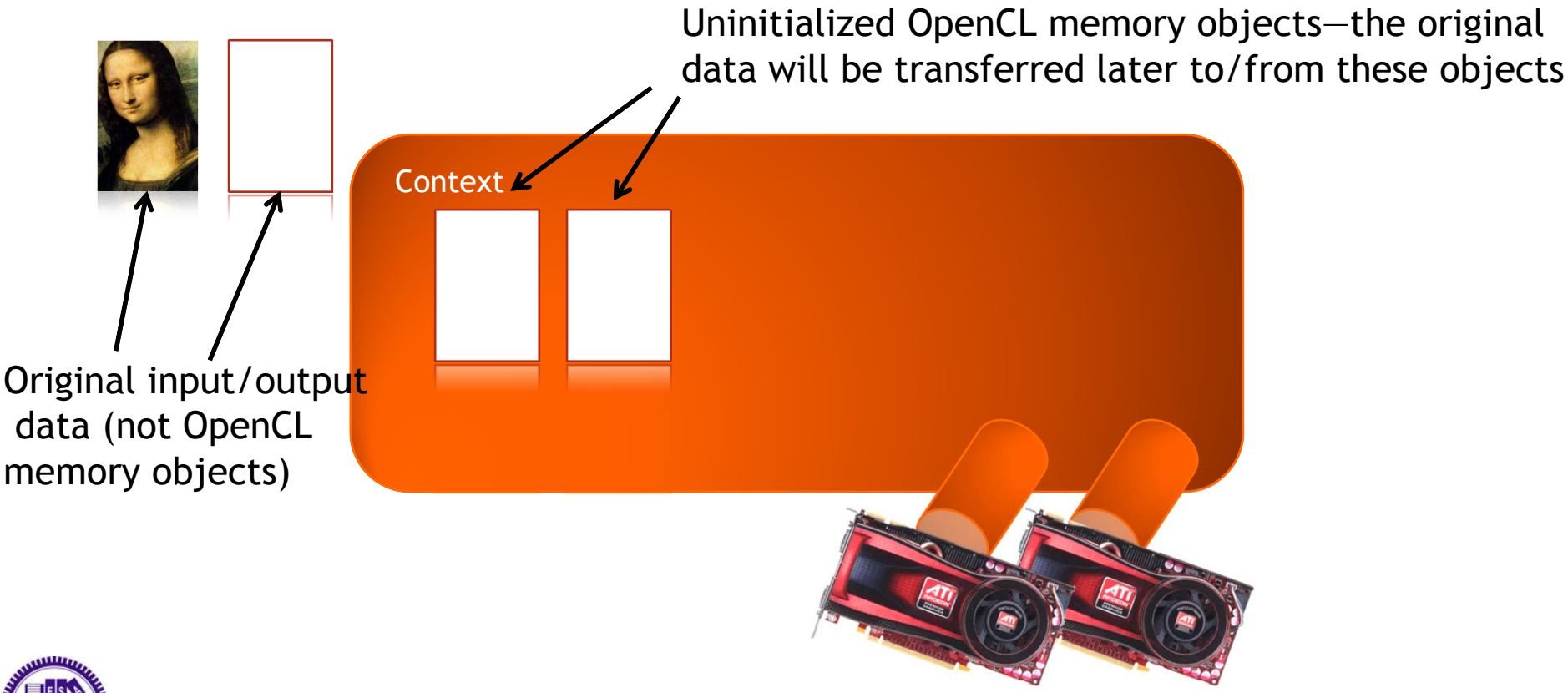
Memory Flags

- Memory flag field in clCreateBuffer() allows us to define characteristics of the buffer object

Memory Flag	Behavior
CL_MEM_READ_WRITE	Specifies memory read / write behavior
CL_MEM_WRITE_ONLY	
CL_MEM_READ_ONLY	
CL_MEM_USE_HOST_PTR	Implementations can cache the contents pointed to by host_ptr in device memory. This cached copy can be used when kernels are executed on a device.
CL_MEM_ALLOC_HOST_PTR	Specifies to the implementation to allocate memory from host accessible memory.
CL_MEM_COPY_HOST_PTR	Specifies to allocate memory for the object and copy the data from memory referenced by host_ptr.

Memory Objects

- Memory objects are associated with a context
  - They must be explicitly transferred to devices prior to execution (covered later)



Transferring Data

- OpenCL provides commands to transfer data to and from devices
  - clEnqueue{Read|Write}{Buffer|Image}
  - Copying from the host to a device is considered *writing*
  - Copying from a device to the host is *reading*
- The write command both initializes the memory object with data and places it on a device
  - The validity of memory objects that are present on multiple devices is undefined by the OpenCL spec (i.e. are vendor specific)
- OpenCL calls also exist to directly map part of a memory object to a host pointer

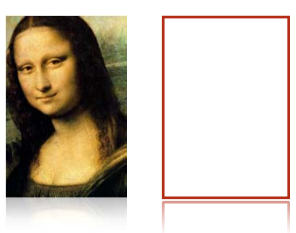
Transferring Data

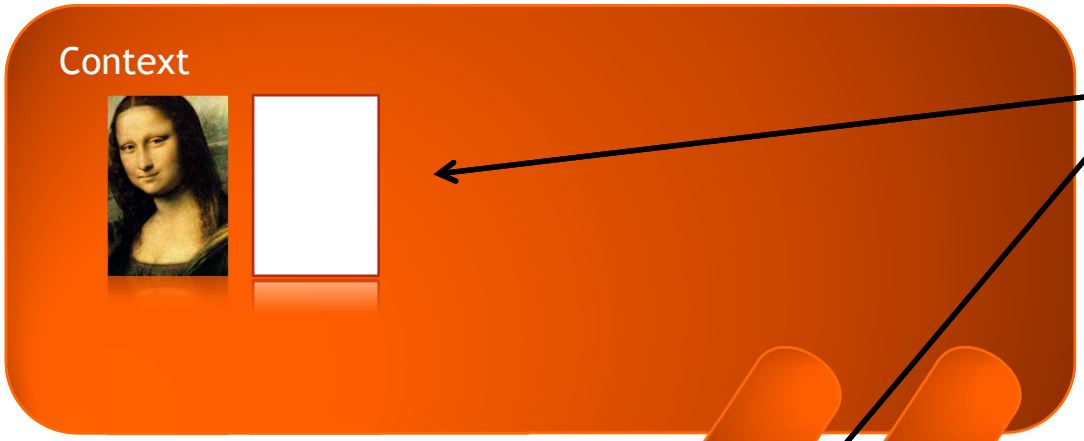
```
cl_int clEnqueueWriteBuffer (cl_command_queue command_queue,
                             cl_mem buffer,
                             cl_bool blocking_write,
                             size_t offset,
                             size_t cb,
                             const void *ptr,
        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

- This command initializes the OpenCL memory object and writes data to the device associated with the command queue
  - The command will write data from a host pointer (*ptr*) to the device
- The *blocking_write* parameter specifies whether or not the command should return before the data transfer is complete
- Events (discussed later) can specify which commands should be completed before this one runs

Transferring Data

- Memory objects are transferred to devices by specifying an action (read or write) and a command queue
  - The validity of memory objects that are present on multiple devices is undefined by the OpenCL spec (i.e. is vendor specific)

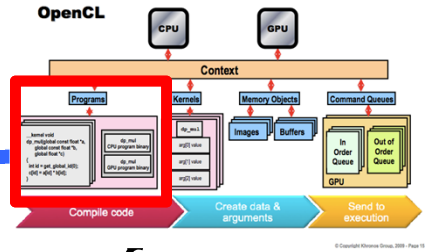




The images are redundant here to show that they are both part of the context (on the host) and physically on the device

Images are written to a device

Programs



- A program object is basically a collection of OpenCL kernels
  - Can be source code (text) or precompiled binary
  - Can also contain constant data and auxiliary functions
- Creating a program object requires either reading in a string (source code) or a precompiled binary
- To compile the program
    - Specify which devices are targeted
      - Program is compiled for each device
    - Pass in compiler flags (optional)
    - Check for compilation errors (optional, output to screen)

Programs

- A program object is created and compiled by providing source code or a binary file and selecting which devices to target

Creating Programs

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

- This function creates a program object from strings of source code
  - *count* specifies the number of strings
  - The user must create a function to read in the source code to a string
- If the strings are not NULL-terminated, the *lengths* fields are used to specify the string lengths

Compiling Programs

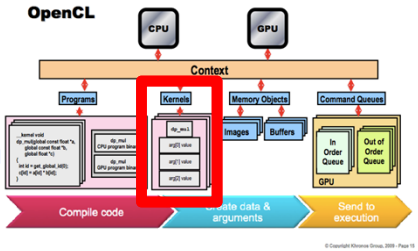
```
cl_int      clBuildProgram (cl_program program,
                            cl_uint num_devices,
                            const cl_device_id *device_list,
                            const char *options,
                            void (CL_CALLBACK *pfn_notify)(cl_program program,
          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

- This function compiles and links an executable from the program object for each device in the context
  - If *device_list* is supplied, then only those devices are targeted
- Optional preprocessor, optimization, and other options can be supplied by the *options* argument

Reporting Compile Errors

- If a program fails to compile, OpenCL requires the programmer to explicitly ask for compiler output
  - A compilation failure is determined by an error value returned from clBuildProgram()
  - Calling clGetProgramBuildInfo() with the program object and the parameter CL_PROGRAM_BUILD_STATUS returns a string with the compiler output

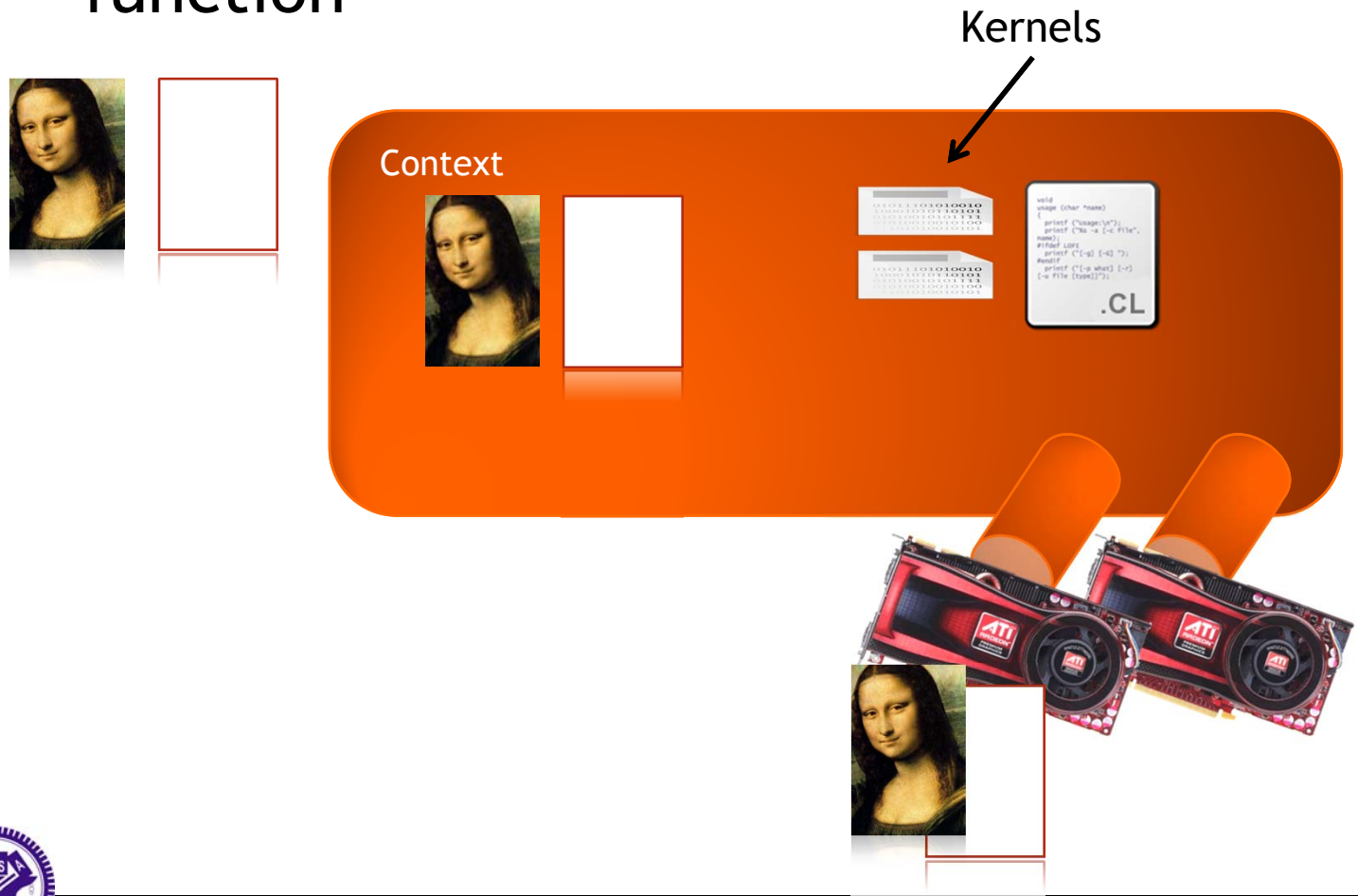
Kernels



- A kernel is a function declared in a program that is executed on an OpenCL device
  - A kernel object is a kernel function along with its associated arguments
- A kernel object is created from a compiled program
- Must explicitly associate arguments (memory objects, primitives, etc) with the kernel object

Kernels

- Kernel objects are created from a program object by specifying the name of the kernel function



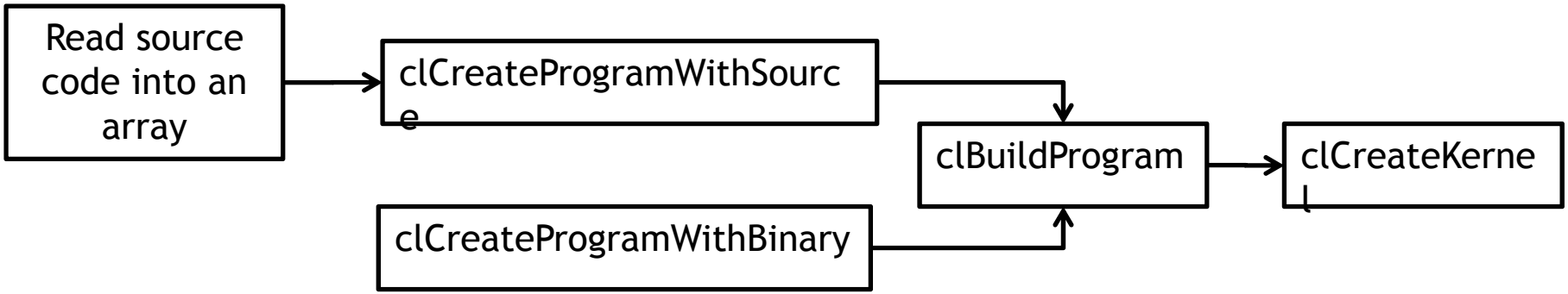
Creating Kernels

```
cl_kernel    clCreateKernel (cl_program program,
                             const char *kernel_name,
                             cl_int *errcode_ret)
```

- Creates a kernel from the given program
  - The kernel that is created is specified by a string that matches the name of the function within the program

Runtime Compilation

- There is a high overhead for compiling programs and creating kernels
  - Each operation only has to be performed once (at the beginning of the program)
    - The kernel objects can be reused any number of times by setting different arguments



Setting Kernel Arguments

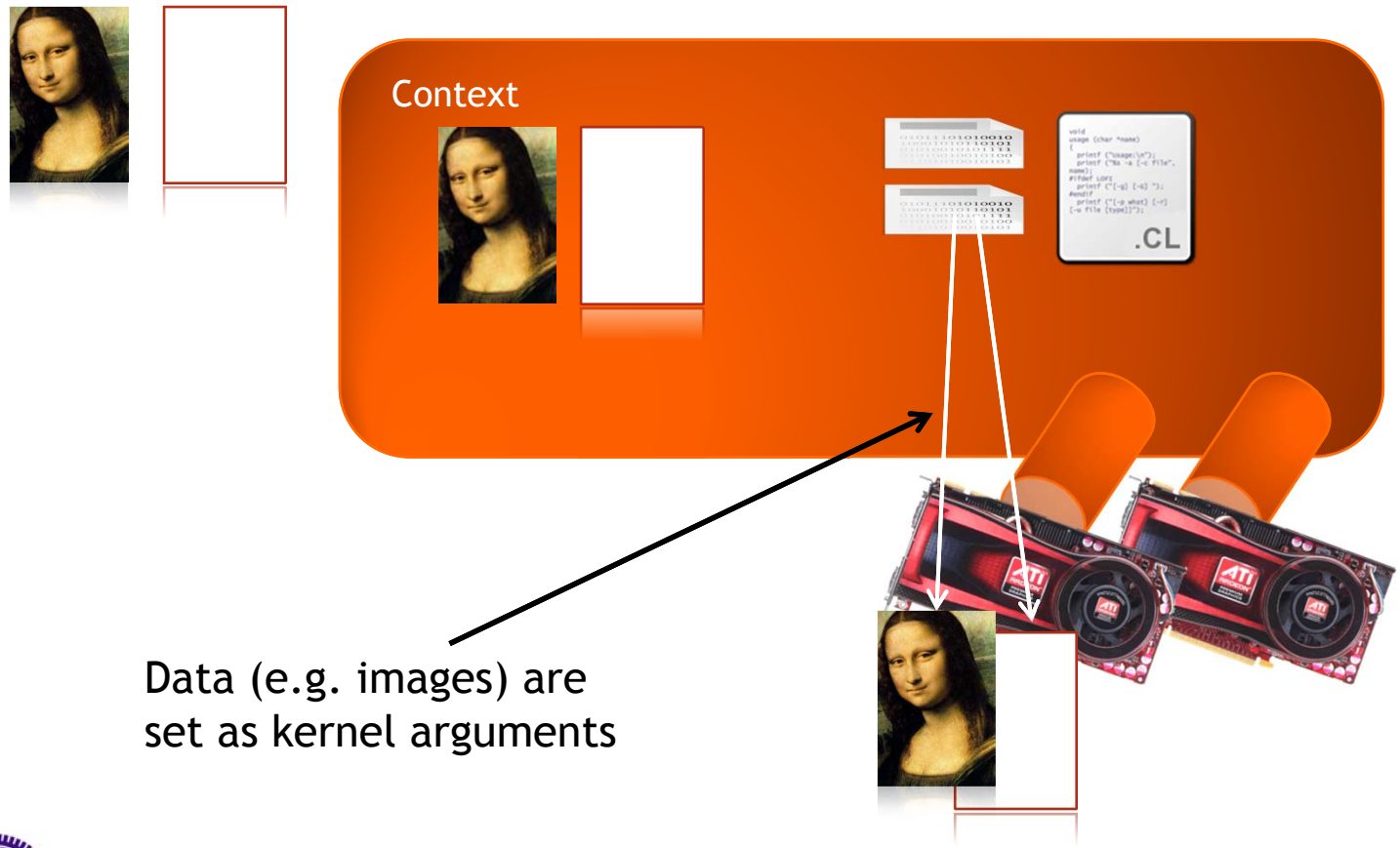
- Kernel arguments are set by repeated calls to clSetKernelArgs

```
cl_int  clSetKernelArg (cl_kernel kernel,
                        cl_uint arg_index,
                        size_t arg_size,
                        const void *arg_value)
```

- Each call must specify:
  - The index of the argument as it appears in the function signature, the size, and a pointer to the data
- Examples:
  - `clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*)&d_iImage);`
  - `clSetKernelArg(kernel, 1, sizeof(int), (void*)&a);`

Kernel Arguments

- Memory objects and individual data values can be set as kernel arguments

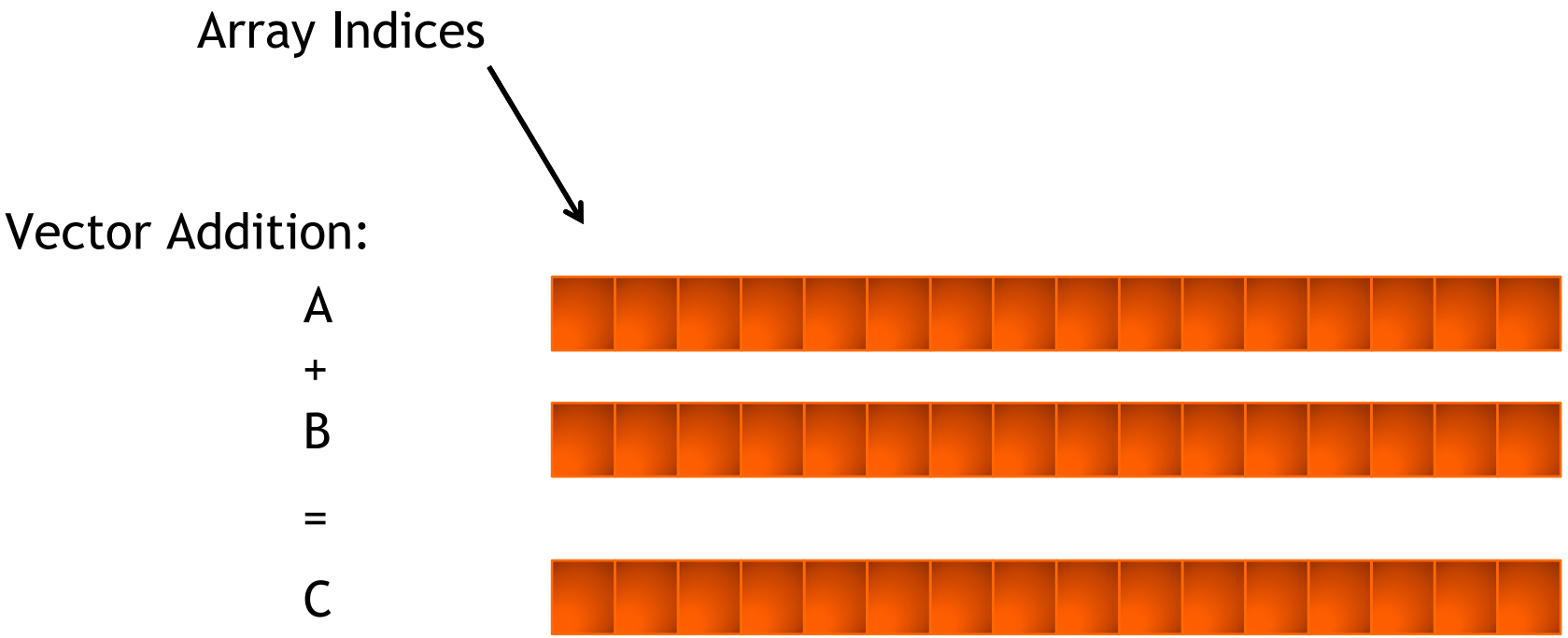


Thread Structure

- Massively parallel programs are usually written so that each thread computes one part of a problem
  - For vector addition, we will add corresponding elements from two arrays, so each thread will perform one addition
  - If we think about the thread structure visually, the threads will usually be arranged in the same shape as the data

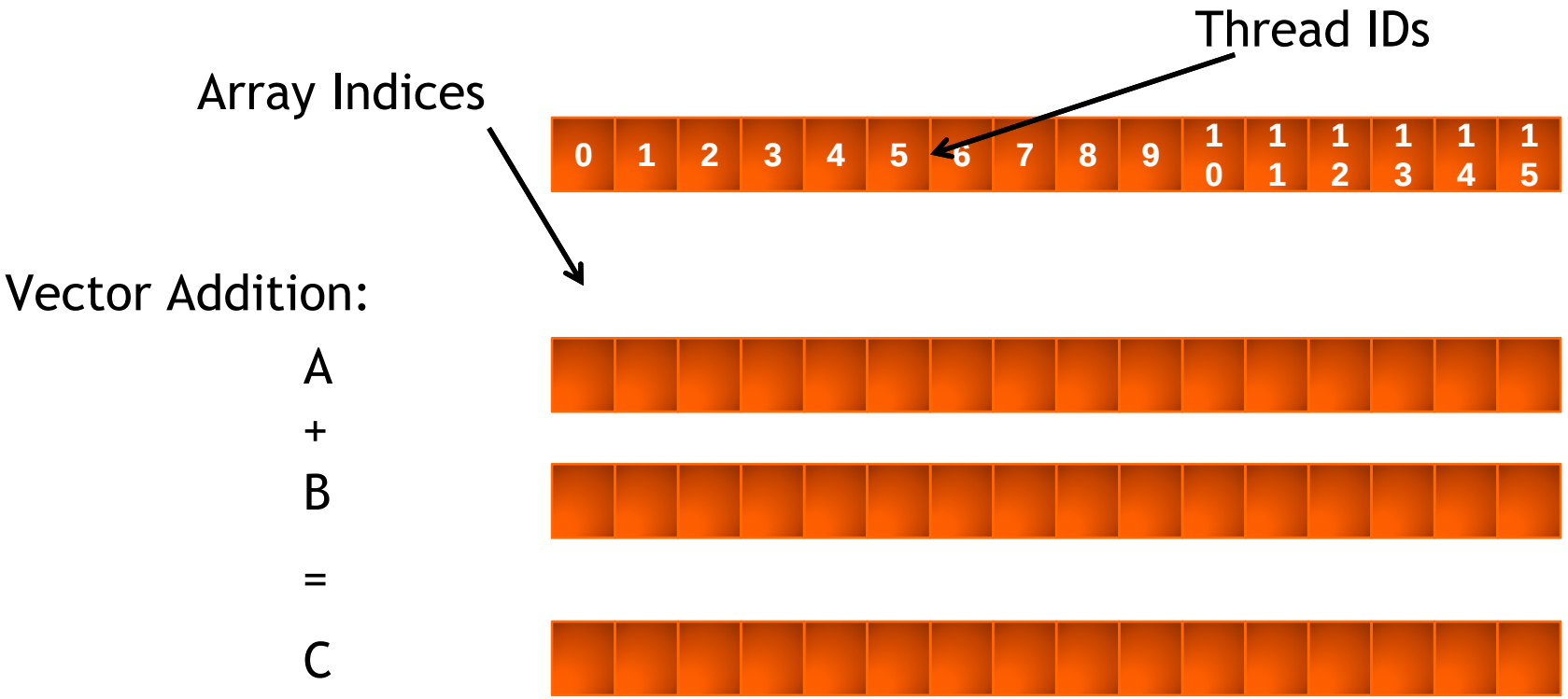
Thread Structure

- Consider a simple vector addition of 16 elements
  - 2 input buffers (A, B) and 1 output buffer (C) are required



Thread Structure

- Create thread structure to match the problem
  - 1-dimensional problem in this case



Thread Structure

- Each thread is responsible for adding the indices corresponding to its ID

Vector Addition:

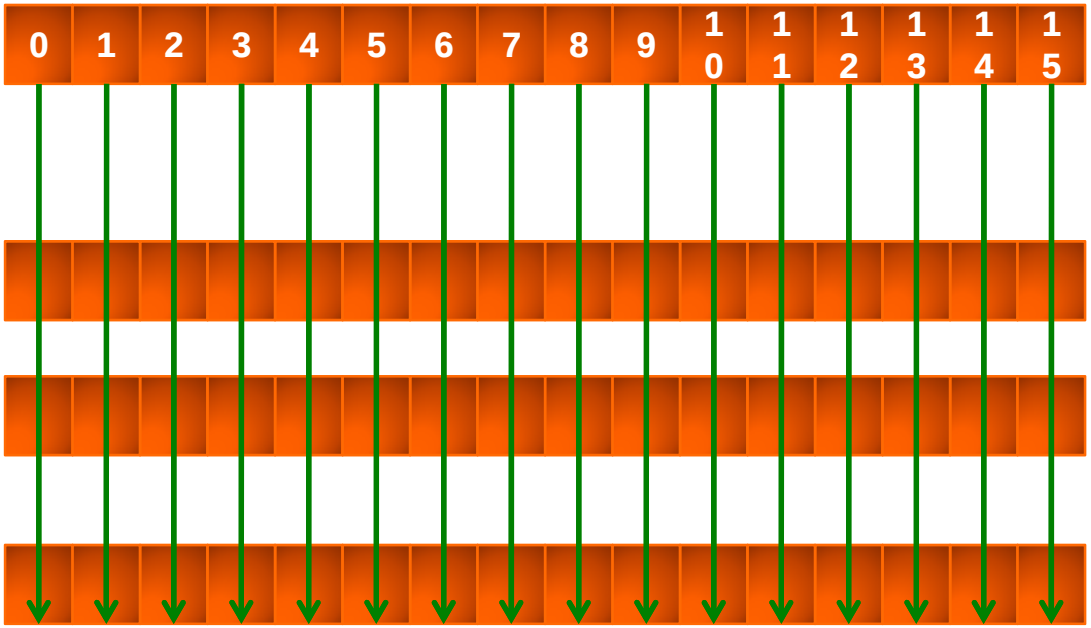
A

+

B

=

C

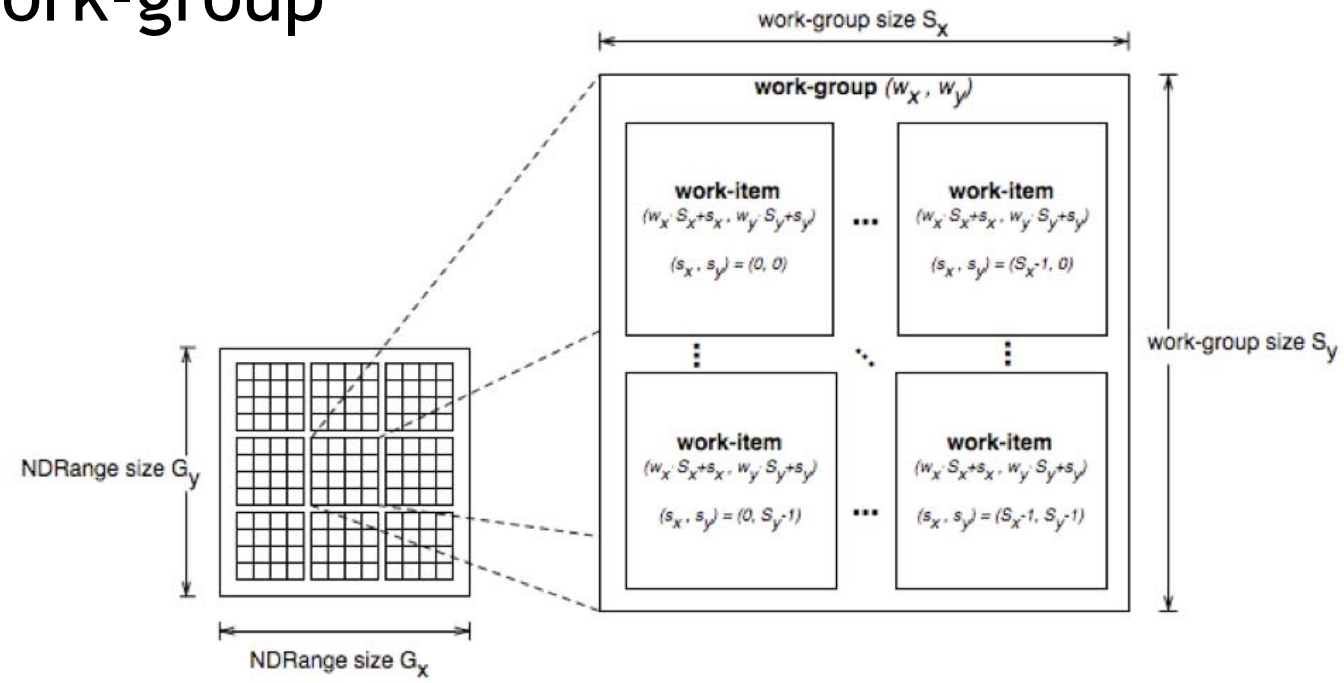


Thread Structure

- OpenCL's thread structure is designed to be scalable
- Each instance of a kernel is called a work-item (though "thread" is commonly used as well)
- Work-items are organized as work-groups
  - Work-groups are independent from one-another (this is where scalability comes from)
- An index space defines a hierarchy of work-groups and work-items

Thread Structure

- Work-items can uniquely identify themselves based on:
  - A global id (unique within the index space)
  - A work-group ID and a local ID within the work-group



Thread Structure

- API calls allow threads to identify themselves and their data
- Threads can determine their global ID in each dimension
  - get_global_id(dim)
  - get_global_size(dim)
- Or they can determine their work-group ID and ID within the workgroup
  - get_group_id(dim)
  - get_num_groups(dim)
  - get_local_id(dim)
  - get_local_size(dim)
- get_global_id(0) = column, get_global_id(1) = row
- get_num_groups(0) * get_local_size(0) == get_global_size(0)

Terminologies: CUDA vs OpenCL

CUDA term	OpenCL term
Grid	NDRange
Thread block	Work group
Thread	Work item
gridDim	get_num_groups()
blockDim	get_local_size()
blockIdx	get_group_id()
threadIdx	get_local_id()
No direct global index (needs to be calculated)	get_global_id()
No direct global size (needs to be calculated)	get_global_size()

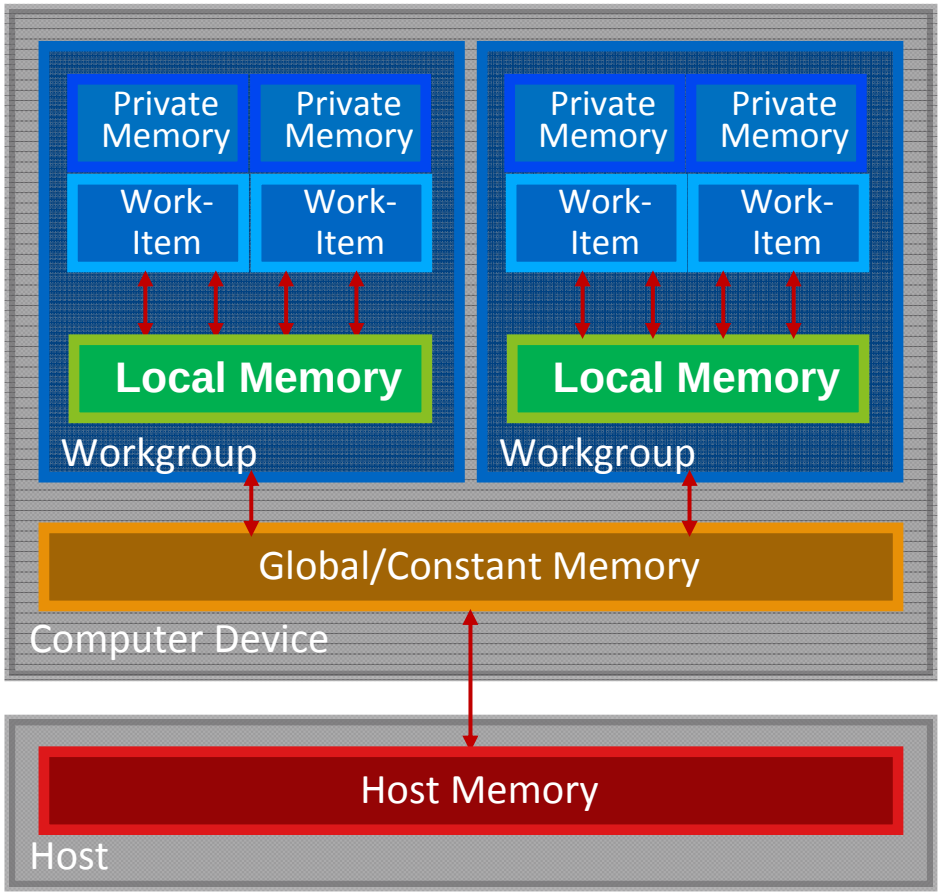
Outline

- What is OpenCL?
- OpenCL Architecture
  - Platform Model
  - Execution Model
  - Memory Model
  - Programming Model
- Case Studies
  - Image Rotation
  - Matrix Multiplication
- Memory Issues
  - Coalescing memory accesses
  - Memory bank conflicts
- Synchronization
  - Within a work group
  - Among commands in command queue(s)
- Profiling Using Events

Memory Model

- The OpenCL memory model defines the various types of memories (closely related to GPU memory hierarchy)

Memory	Description
Global	Accessible by all work-items
Constant	Read-only, global
Local	Local to a work-group
Private	Private to a work-item



Terminologies: CUDA vs OpenCL

CUDA term	OpenCL term
Global memory	Global memory
Constant memory	Constant memory
Shared memory	Local memory
Local memory	Private memory

Memory Model

- Memory management is explicit
  - Must move data from host memory to device global memory, from global memory to local memory, and back
- Work-groups are assigned to execute on compute-units
  - No guaranteed communication/coherency between different work-groups (no software mechanism in the OpenCL specification)

Writing a Kernel

- One instance of the kernel is created for each thread
- Kernels:
  - Must begin with keyword __kernel
  - Must have return type void
  - Must declare the address space of each argument that is a memory object (next slide)
  - Use API calls (such as get_global_id()) to determine which data a thread will work on

Address Space Identifiers

- __global – memory allocated from global address space
- __constant – a special type of read-only memory
- __local – memory shared by a work-group
- __private – private per work-item memory
- __read_only/__write_only – used for images
- Kernel arguments that are memory objects must be global, local, or constant

Terminologies: CUDA vs OpenCL

CUDA term	OpenCL term
__global__ function	__kernel function
__device__ function	function (no qualifier required)
__constant__ variable declaration	__constant variable declaration
__device__ variable declaration	__global variable declaration
__shared__ variable declaration	__local variable declaration

Example Kernel

- Simple vector addition kernel:

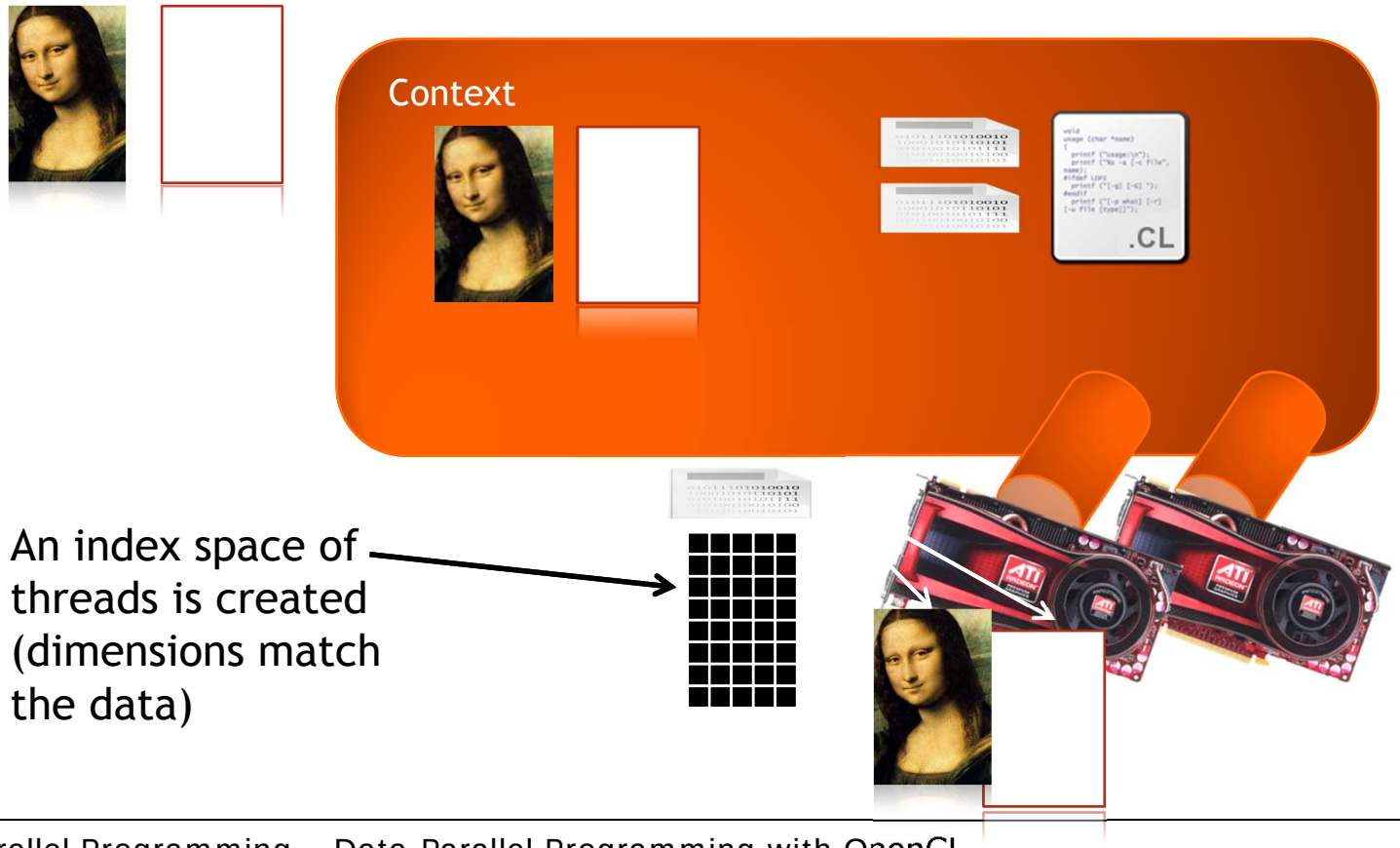
```
__kernel
void vecadd(__global int* A,
            __global int* B,
            __global int* C) {
   int tid = get_global_id(0);
   C[tid] = A[tid] + B[tid];
}
```

Executing the Kernel

- Need to set the dimensions of the index space, and (optionally) of the work-group sizes
- Kernels execute asynchronously from the host
  - clEnqueueNDRangeKernel just adds is to the queue, but doesn't guarantee that it will start executing

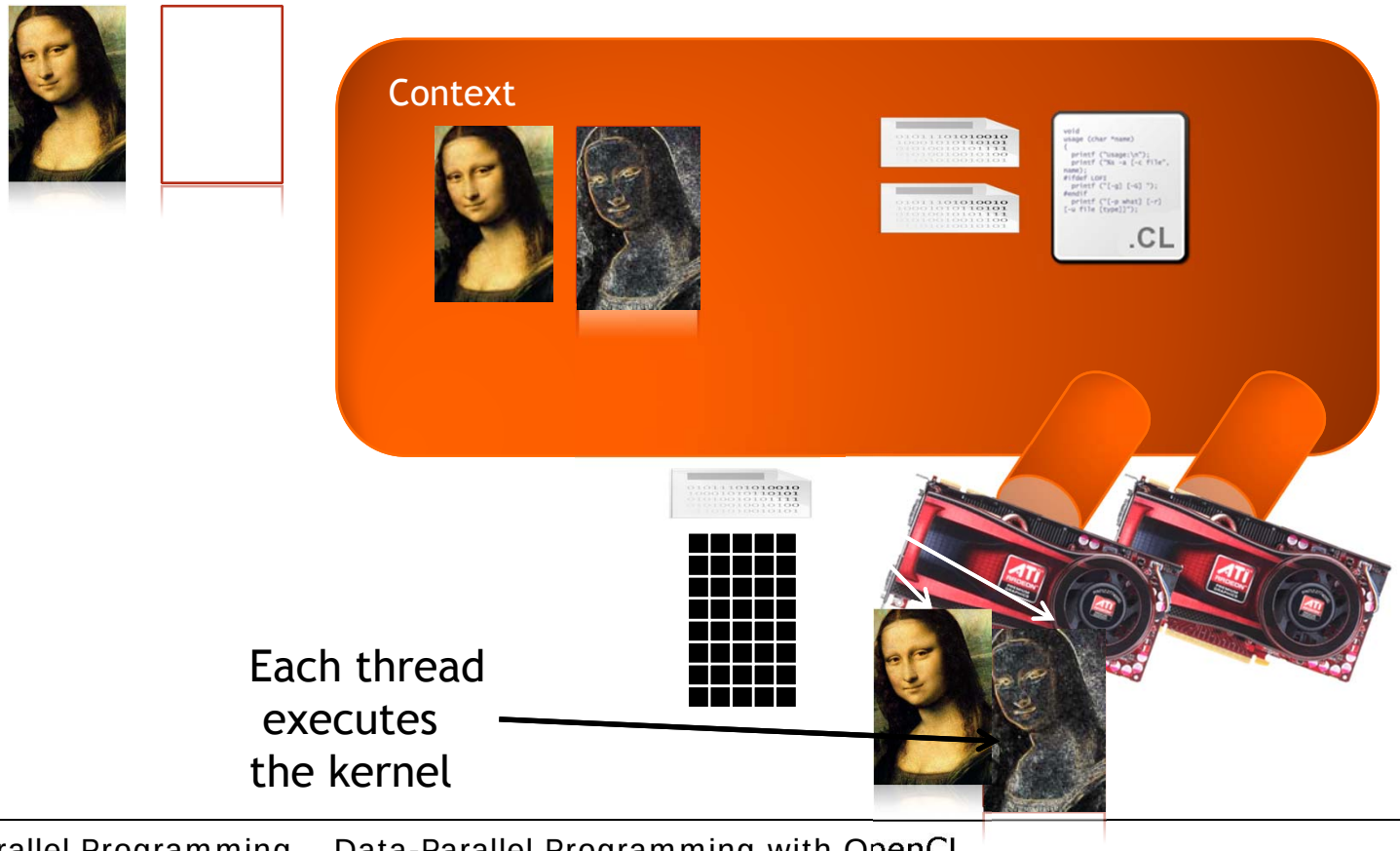
Executing the Kernel

- A thread structure defined by the index-space that is created
  - Each thread executes the same kernel on different data



Executing the Kernel

- A thread structure defined by the index-space that is created
  - Each thread executes the same kernel on different data

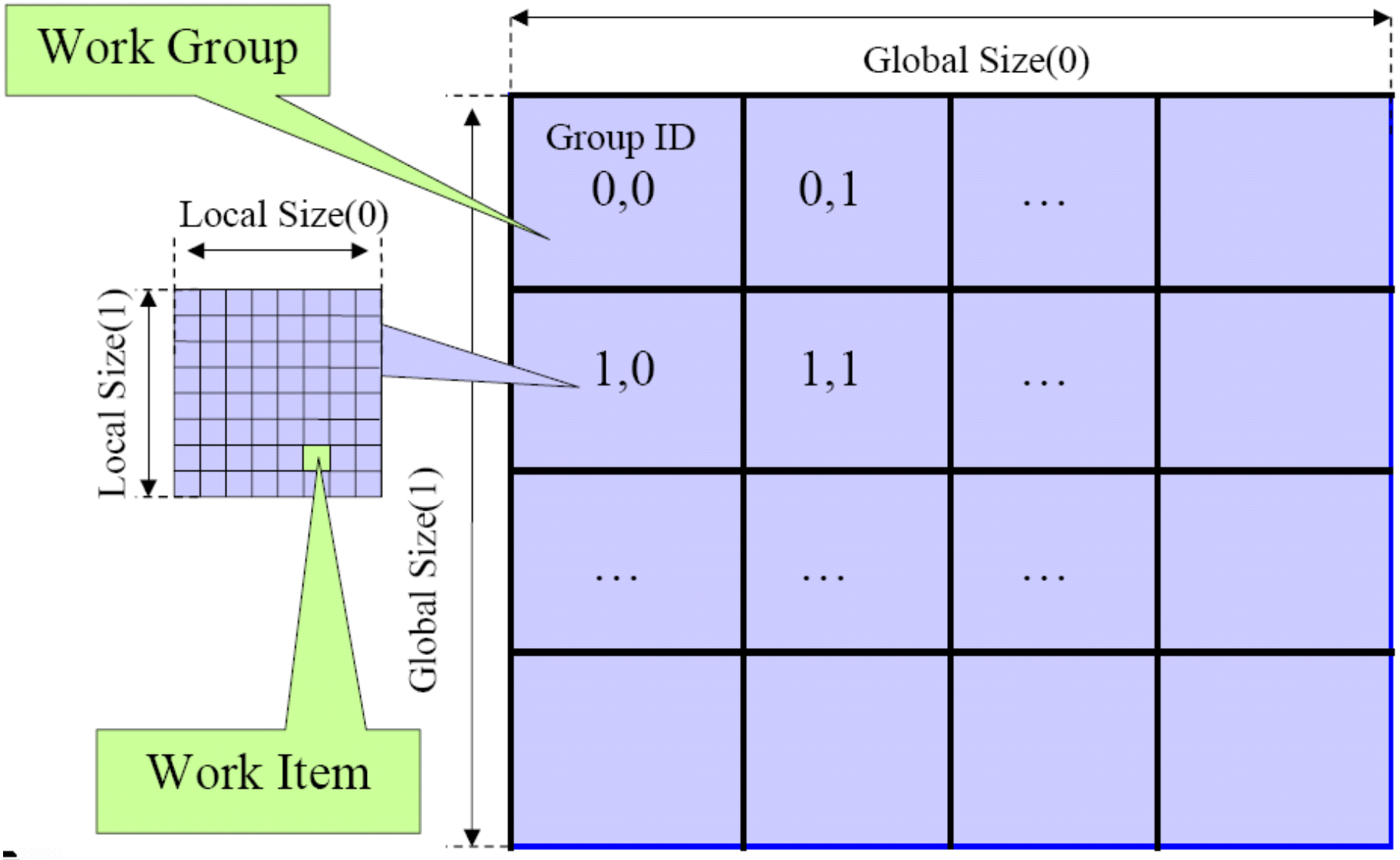


Executing the Kernel

```
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)
```

- Tells the device associated with a command queue to begin executing the specified kernel
- The global (index space) must be specified and the local (work-group) sizes are optionally specified
- A list of events can be used to specify prerequisite operations that must be complete before executing

NDRange


Work Group
Global Size(0)
Group ID
0,0
0,1
…
Local Size(0)
Local Size(1)
1,0
1,1
…
…
…
…
Global Size(1)
Work Item

Copying Data Back

- The last step is to copy the data back from the device to the host
- Similar call as writing a buffer to a device, but data will be transferred back to the host

```
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)
```

Copying Data Back

- The output data is read from the device back to the host

Releasing Resources

- Most OpenCL resources/objects are pointers that should be freed after they are done being used
- There is a clRelase{Resource} command for most OpenCL types
  - Ex: clReleaseProgram(), clReleaseMemObject()

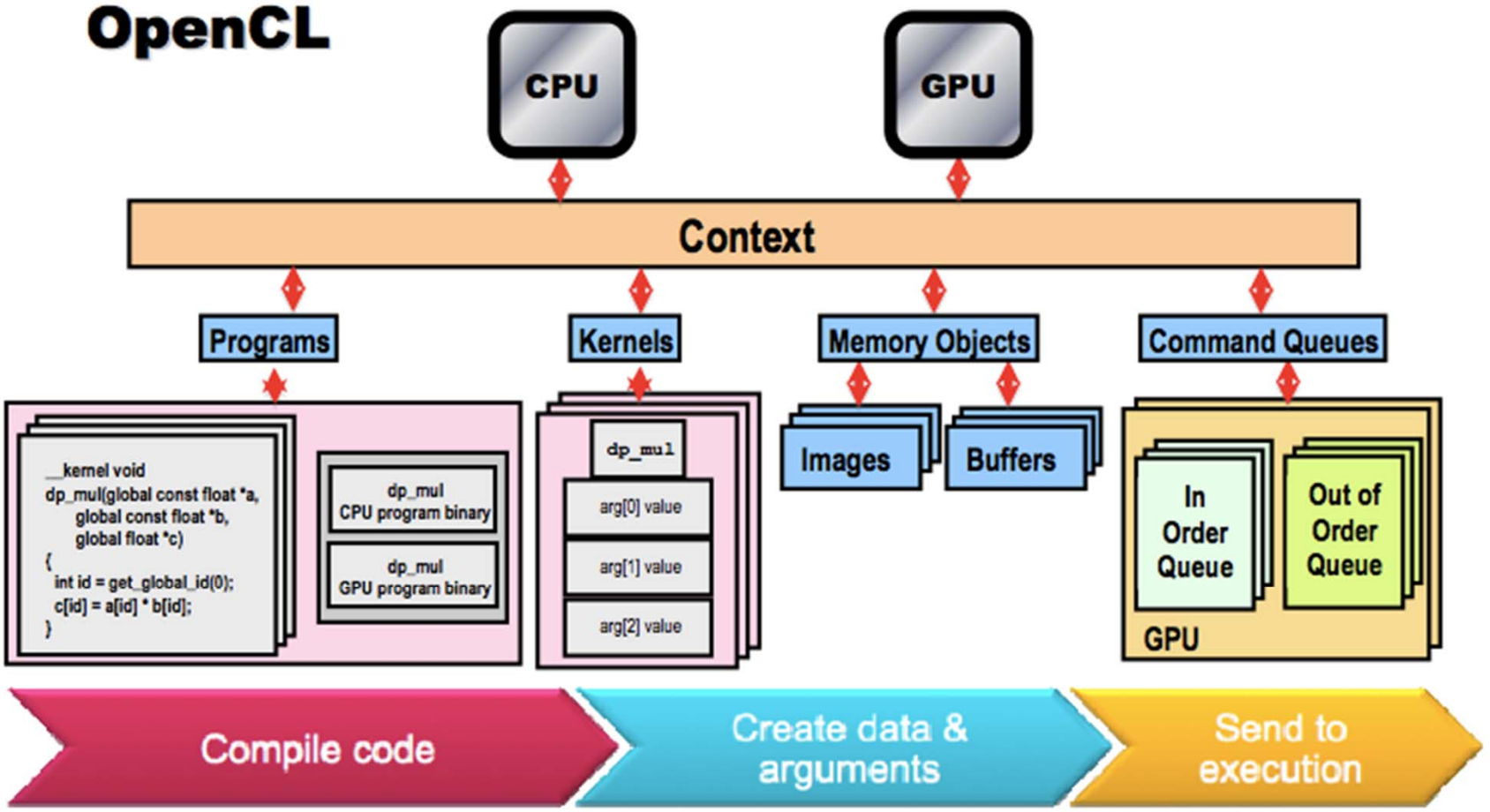
Error Checking

- OpenCL APIs return error codes as negative integer values
  - Return value of 0 indicates CL_SUCCESS
  - Negative values indicates an error
    - cl.h defines meaning of each return value

CL_DEVICE_NOT_FOUND	-1
CL_DEVICE_NOT_AVAILABLE	-2
CL_COMPILER_NOT_AVAILABLE	-3
CL_MEM_OBJECT_ALLOCATION_FAILURE	-4
CL_OUT_OF_RESOURCES	-5

- **Note:** Errors are sometimes reported asynchronously

Big Picture



© Copyright Khronos Group, 2009 - Page 15

Outline

- What is OpenCL?
- OpenCL Architecture
  - Platform Model
  - Execution Model
  - Memory Model
  - Programming Model
- Case Studies
  - Image Rotation
  - Matrix Multiplication
- Memory Issues
  - Coalescing memory accesses
  - Memory bank conflicts
- Synchronization
  - Within a work group
  - Among commands in command queue(s)
- Profiling Using Events

Programming Model - Data Parallel

- Computation is defined in terms of a sequence of instructions executed on multiple elements of a memory object
- These elements are in an index space, which defines how the execution maps onto the work-items
- In the OpenCL data-parallel model, it is not a strict requirement that there be one work-item for every element in a memory object over which a kernel is executed in parallel
  - You may define global_work_offset

Programming Model - Data Parallel

- The OpenCL data-parallel programming model is hierarchical
- The hierarchical subdivision can be specified in two ways:
  - Explicitly - the developer defines the total number of work-items to execute in parallel, as well as the division of work-items into specific work-groups
  - Implicitly - the developer specifies the total number of work-items to execute in parallel, and OpenCL manages the division into work-groups

Programming Model - Task Parallel

- Kernel instance is executed independent of any index space
- Parallelism is expressed using vector data types implemented by the device, enqueuing multiple tasks, and/or enqueuing native kernels developed using a programming model orthogonal to OpenCL
  - Vector data types
    - char2, char4, char8, char16, int2, int4, int8, int16, etc.
  - Native kernels: native C/C++ functions
    - Using clEnqueueNativeKernel(...)

Programming Model - Synchronization

- The two domains of synchronization in OpenCL are
  - Work-items in a single workgroup and
  - Commands in a single context
- Work-group barriers (`barrier()`) enable synchronization of work-items in a work-group
- Each work-item in work-group must first execute the barrier before executing any beyond the work-group barrier
- Either all of, or none of, the work-items in a work-group must encounter the barrier
- Global synchronization is not allowed

Programming Model - Synchronization

- There are two types of synchronization between commands in a command-queue:
  - command-queue barrier - enforces ordering within a single queue
    - Any resulting changes to memory are available to the following commands in the queue
  - events - enforces ordering between or within queues
    - Enqueued commands in OpenCL return an event identifying the command as well as the memory object updated by it. This ensures that following commands waiting on that event see the updated memory objects before they execute

Outline

- What is OpenCL?
- OpenCL Architecture
  - Platform Model
  - Execution Model
  - Memory Model
  - Programming Model
- Case Studies
  - Image Rotation
  - Matrix Multiplication
- Memory Issues
  - Coalescing memory accesses
  - Memory bank conflicts
- Synchronization
  - Within a work group
  - Among commands in command queue(s)
- Profiling Using Events

Image Rotation

- A common image processing routine
  - Applications in matching, alignment, etc.
- New coordinates of point (x_1,y_1) when rotated by an angle Θ around (x_0,y_0)

$$\begin{bmatrix} x^* \\ y^* \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Original Image

Rotated Image (90°)

- Each coordinate for every point in the image can be calculated independently

Image Rotation

- Input: To copy to device
  - Image (2D Matrix of floats)
  - Rotation parameters
  - Image dimensions
- Output: From device
  - Rotated Image

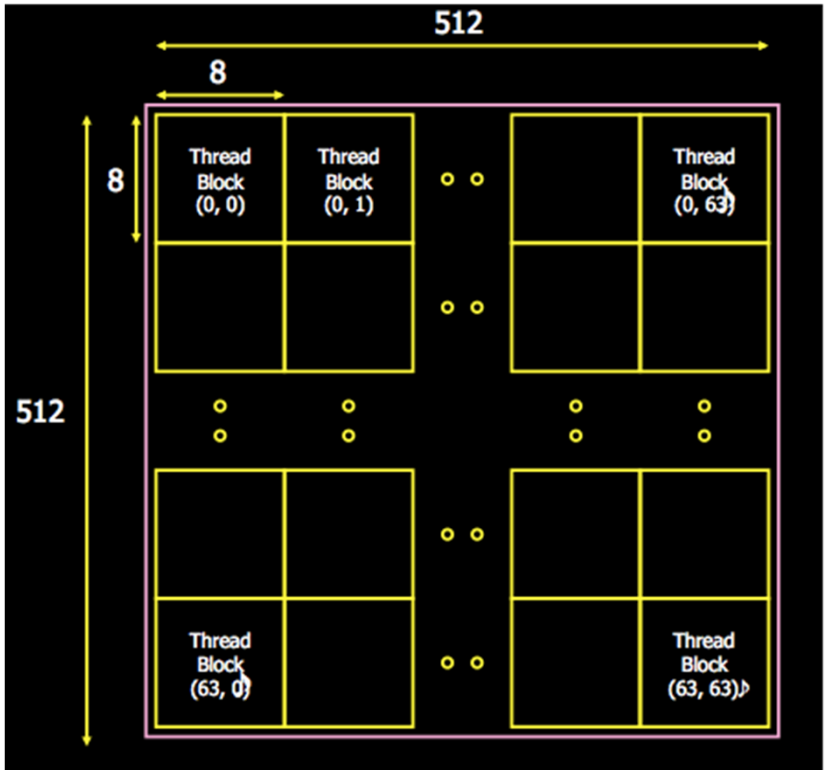
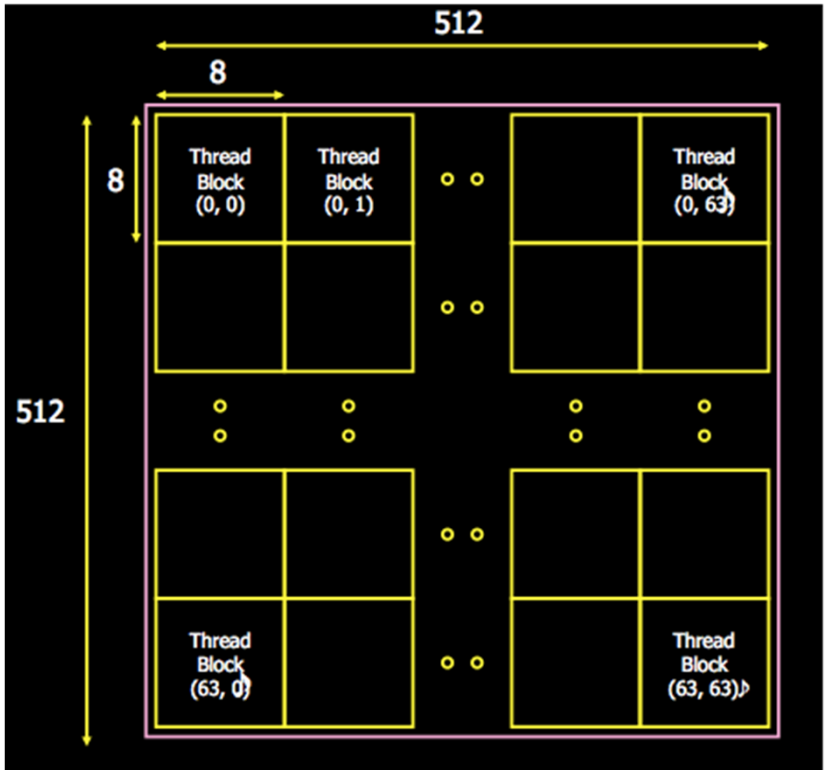


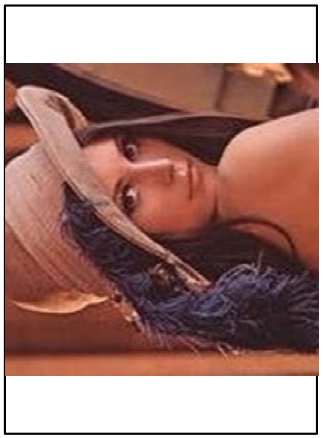
Image Rotation

- Main Steps
  - Copy image to device by enqueueing a write to a buffer on the device from the host
  - Run the Image rotation kernel on input image
  - Copy output image to host by enqueueing a read from a buffer on the device



The OpenCL Kernel

- Parallel kernel is not always this obvious
  - Profiling of an application is often necessary to find the bottlenecks and locate the data parallelism
- In this example grid of output image decomposed into work items
  - Not all parts of the input image copied to the output image after rotation, corners of I/P image could be lost after rotation



OpenCL Kernel

```
__kernel  void image_rotate(
  __global float * src_data,        //Data in global memory
  __global float * dest_data,       //Data in global memory
  int W,    int H,                  //Image Dimensions
  float sinTheta, float cosTheta ) //Rotation Parameters
{
  //Thread gets its index within index space
  const int ix = get_global_id(0);
  const int iy = get_global_id(1);

  //Calculate location of data to move into ix and iy
  float xpos = ( ((float)ix)*cosTheta + ((float)iy)*sinTheta );
  float ypos = ( ((float)iy)*cosTheta - ((float)ix)*sinTheta );

  if ( (((int)xpos>=0) && ((int)xpos<W))   //Bound Checking
      && (((int)ypos>=0) && ((int)ypos<H)) ) {
    //Read (xpos,ypos) src_data and store at (ix,iy) in dest_data
    dest_data[iy*W+ix]=src_data[(int)(floor(ypos*W+xpos))];
  }
}
```

$$x^* = x\cos\theta + y\sin\theta$$
$$y^* = -x\sin\theta + y\cos\theta$$

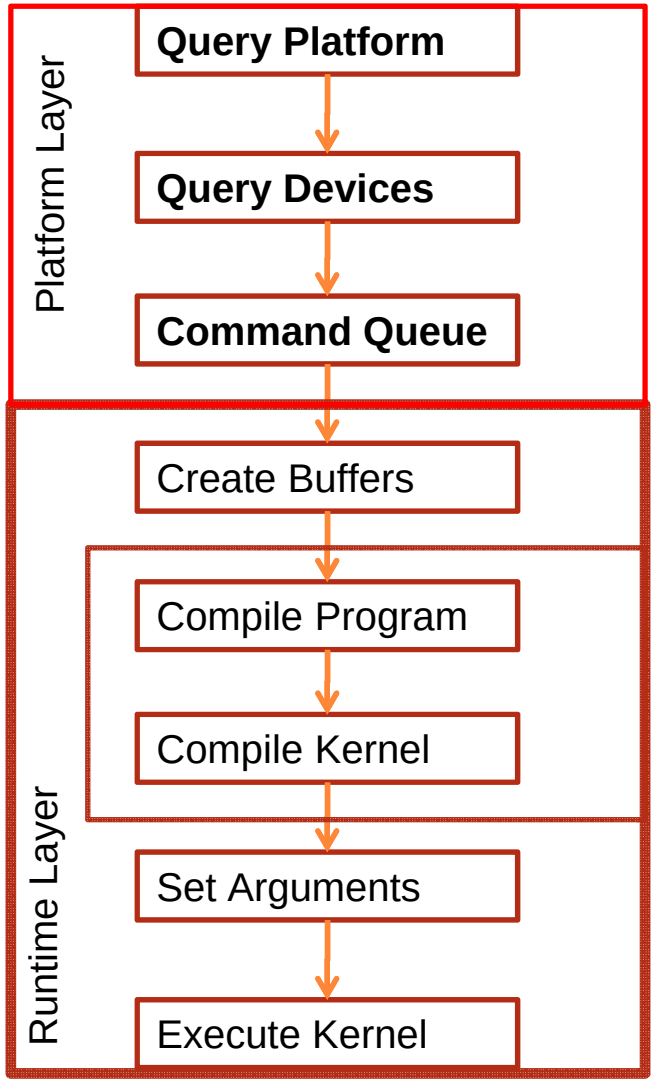
Step0: Initialize Device

- Declare context
- Choose a device from context
- Using device and context create a command queue

```
cl_context myctx = clCreateContextFromType (
            0, CL_DEVICE_TYPE_GPU,
                    NULL, NULL, &ciErrNum);
```

```
ciErrNum = clGetDeviceIDs (0,
            CL_DEVICE_TYPE_GPU,
            1, &device,  cl_uint *num_devices)
```

```
cl_commandqueue myqueue ;
myqueue = clCreateCommandQueue(
                    myctx, device, 0, &ciErrNum);
```



Step1: Create Buffers

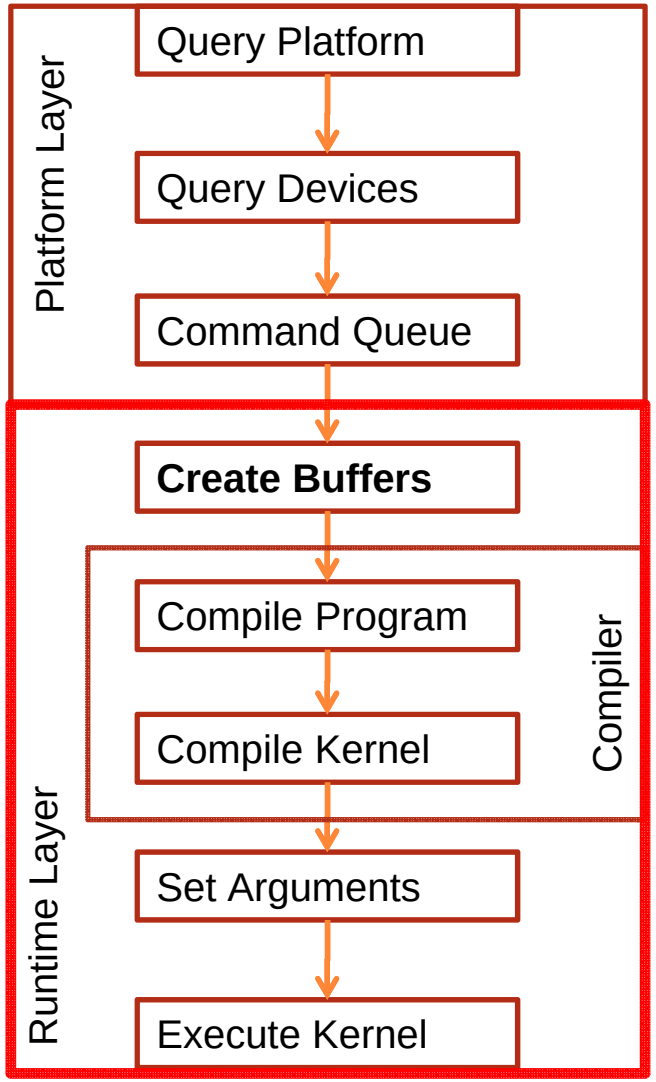
- Create buffers on device
  - Input data is read-only
  - Output data is write-only

```
cl_mem d_ip = clCreateBuffer(
              myctx, CL_MEM_READ_ONLY,
              mem_size,
              NULL, &ciErrNum);
```

```
cl_mem d_op = clCreateBuffer(
              myctx, CL_MEM_WRITE_ONLY,
              mem_size,
              NULL, &ciErrNum);
```

- Transfer input data to the device

```
ciErrNum = clEnqueueWriteBuffer (
              myqueue , d_ip, CL_TRUE,
              0, mem_size, (void *)src_image,
              0, NULL,  NULL)
```

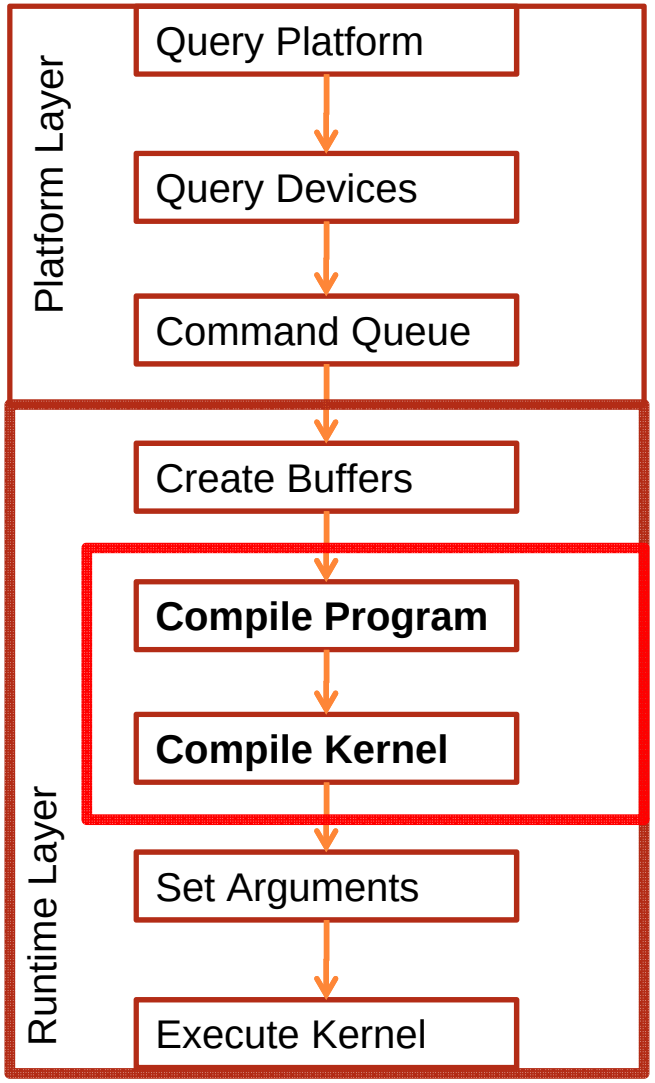


Step2: Build Program, Select Kernel

```
// create the program
cl_program myprog = clCreateProgramWithSource
            ( myctx,1, (const char **)&source,
            &program_length, &ciErrNum);
```

```
// build the program
ciErrNum = clBuildProgram( myprog, 0,
            NULL, NULL, NULL, NULL);
```

```
//Use the “image_rotate” function as the kernel
cl_kernel mykernel = clCreateKernel (
            myprog , “image_rotate” ,
             error_code)
```

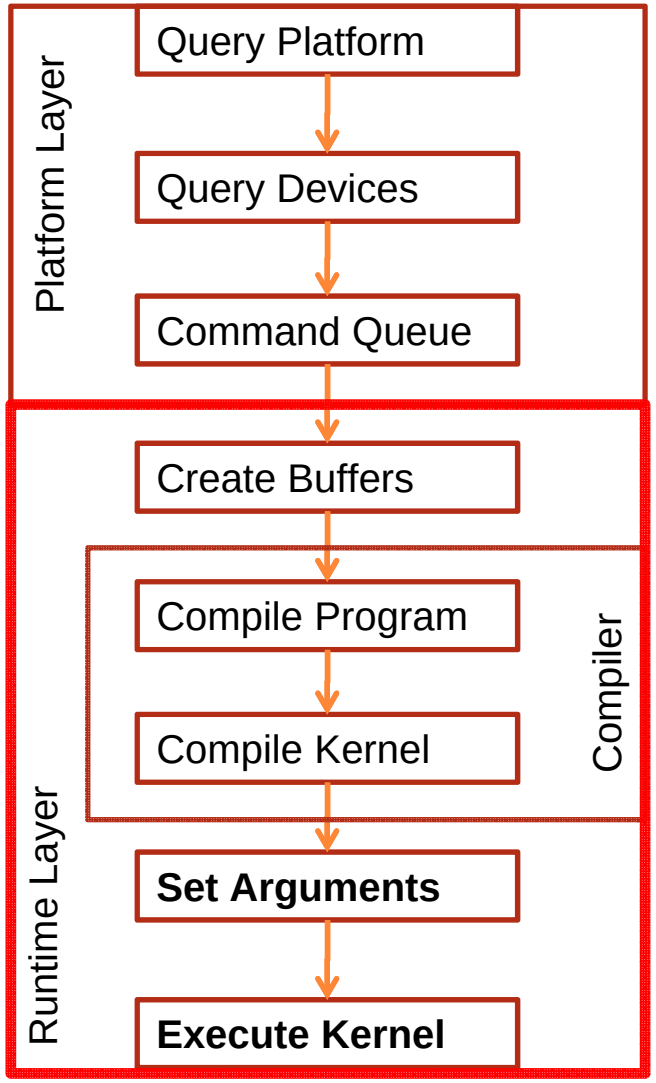


Step3: Set Arguments, Enqueue Kernel

```
// Set Arguments
clSetKernelArg(mykernel, 0, sizeof(cl_mem),
               (void *)&d_ip);
clSetKernelArg(mykernel, 1, sizeof(cl_mem),
               (void *)&d_op);
 clSetKernelArg(mykernel, 2, sizeof(cl_int),
               (void *)&W);
...
```

```
//Set local and global workgroup sizes
size_t localws[2] = {16,16} ;
size_t globalws[2] = {W, H};//Assume divisible by 16
```

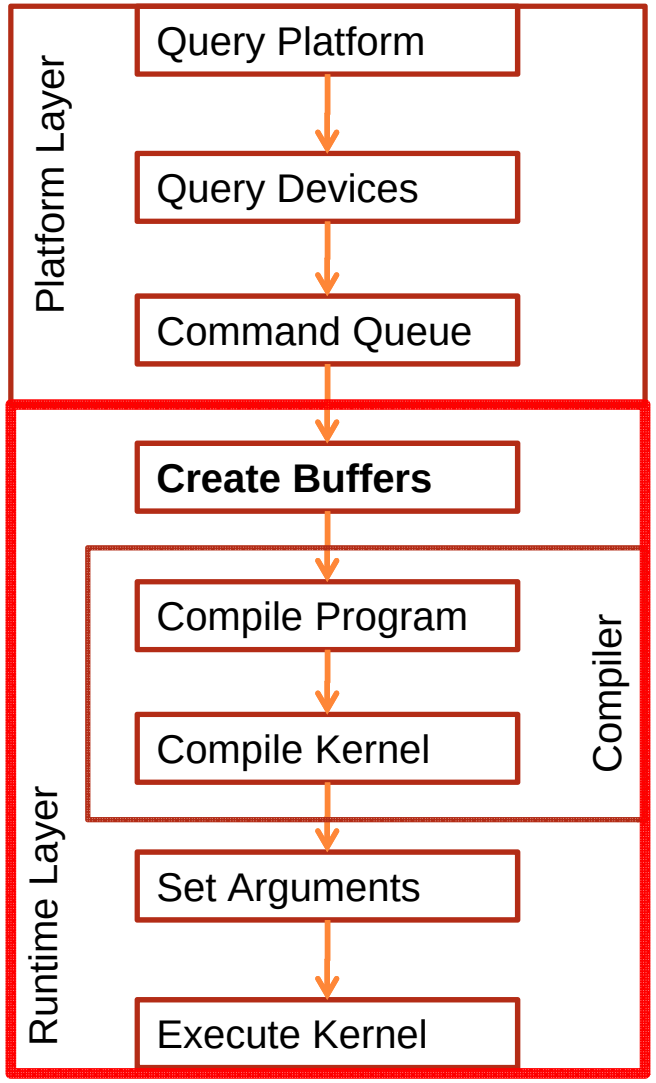
```
// execute kernel
clEnqueueNDRangeKernel(
               myqueue , myKernel,
               2, 0, globalws, localws,
               0, NULL, NULL);
```



Step4: Read Back Result

- Only necessary for data required on the host
- Data output from one kernel can be reused for another kernel
  - Avoid redundant host-device IO

```
// copy results from device back to host
clEnqueueReadBuffer(
                myctx, d_op,
                CL_TRUE,                //Blocking Read Back
                0, mem_size,  (void *) op_data,
                 NULL, NULL, NULL);
```



Outline

- What is OpenCL?
- OpenCL Architecture
  - Platform Model
  - Execution Model
  - Memory Model
  - Programming Model
- Case Studies
  - Image Rotation
  - Matrix Multiplication
- Memory Issues
  - Coalescing memory accesses
  - Memory bank conflicts
- Synchronization
  - Within a work group
  - Among commands in command queue(s)
- Profiling Using Events

Basic Matrix Multiplication

- Non-blocking (non-tiling) matrix multiplication
  - Doesn't use local memory
    - Each element of matrix reads its own data independently
- Serial matrix multiplication

```
for(int i = 0; i < Ha; i++) {
  for(int j = 0; j < Wb; j++) {
    c[i][j] = 0;
    for(int k = 0; k < Wa; k++) {
      c[i][j] +=  a[i][k] + b[k][j];
    }
  }
}
```

Basic Matrix Multiplication

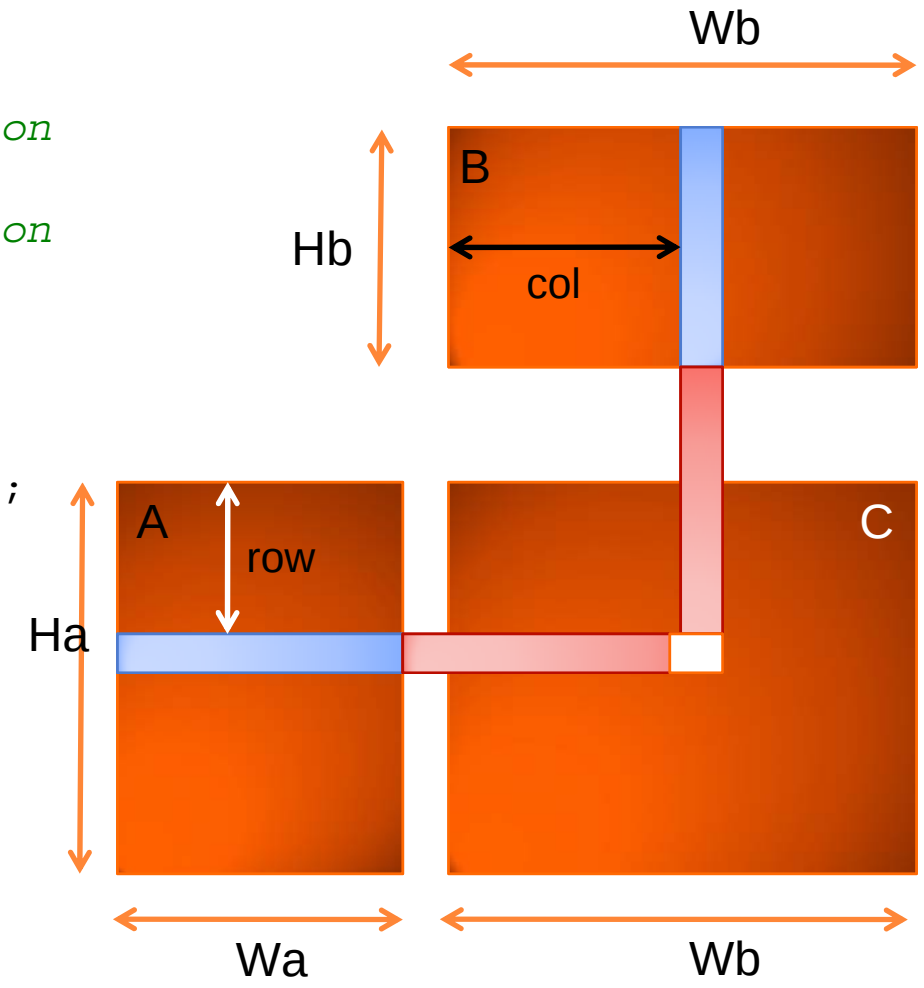
```
for(int i = 0; i < Ha; i++) {
  for(int j = 0; j < Wb; j++) {
    c[i][j] = 0;
    for(int k = 0; k < Wa; k++) {
      c[i][j] +=  a[i][k] + b[k][j];
    }
  }
}
```

- Reuse code from image rotation
  - Create context, command queues and compile program
  - Only need one more input memory object for 2nd matrix

Simple Matrix Multiplication

```
__kernel void simpleMultiply(
    __global float* c, int Wa, int Wb,
    __global float* a, __global float* b)
{

  //Get global position in Y direction
  int row = get_global_id(1);
  //Get global position in X direction
  int col   = get_global_id(0);
  float sum = 0.0f;
  //Calculate result of one element
  for (int i = 0; i < Wa; i++)  {
    sum += a[row*Wa+i] * b[i*Wb+col];
  }
  c[row*Wb+col] = sum;
}
```

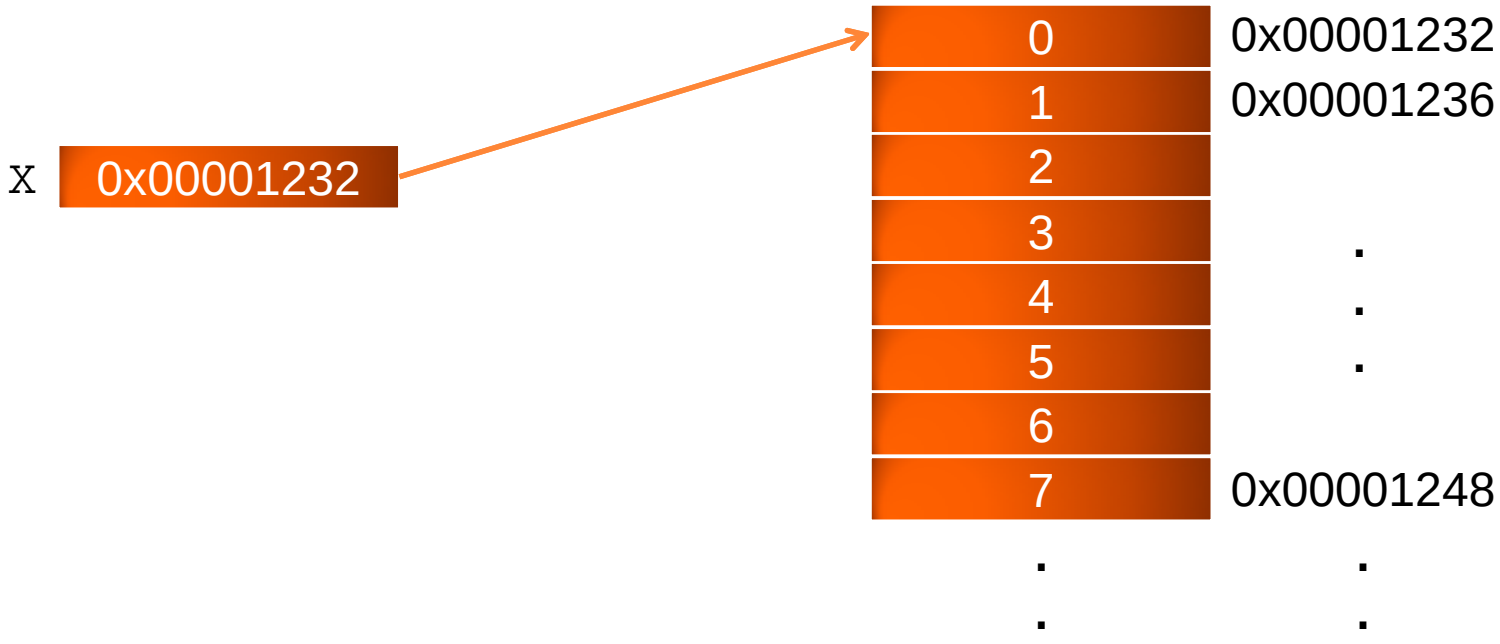


Outline

- What is OpenCL?
- OpenCL Architecture
  - Platform Model
  - Execution Model
  - Memory Model
  - Programming Model
- Case Studies
  - Image Rotation
  - Matrix Multiplication
- Memory Issues
  - Coalescing memory accesses
  - Memory bank conflicts
- Synchronization
  - Within a work group
  - Among commands in command queue(s)
- Profiling Using Events

How GPU's Wide Memory Buses Work

- Array `X` is a pointer to an array of integers (4-bytes each) located at address 0x00001232



- A thread wants to access the data at `X[0]`

```
int tmp = X[0];
```

Bus Addressing

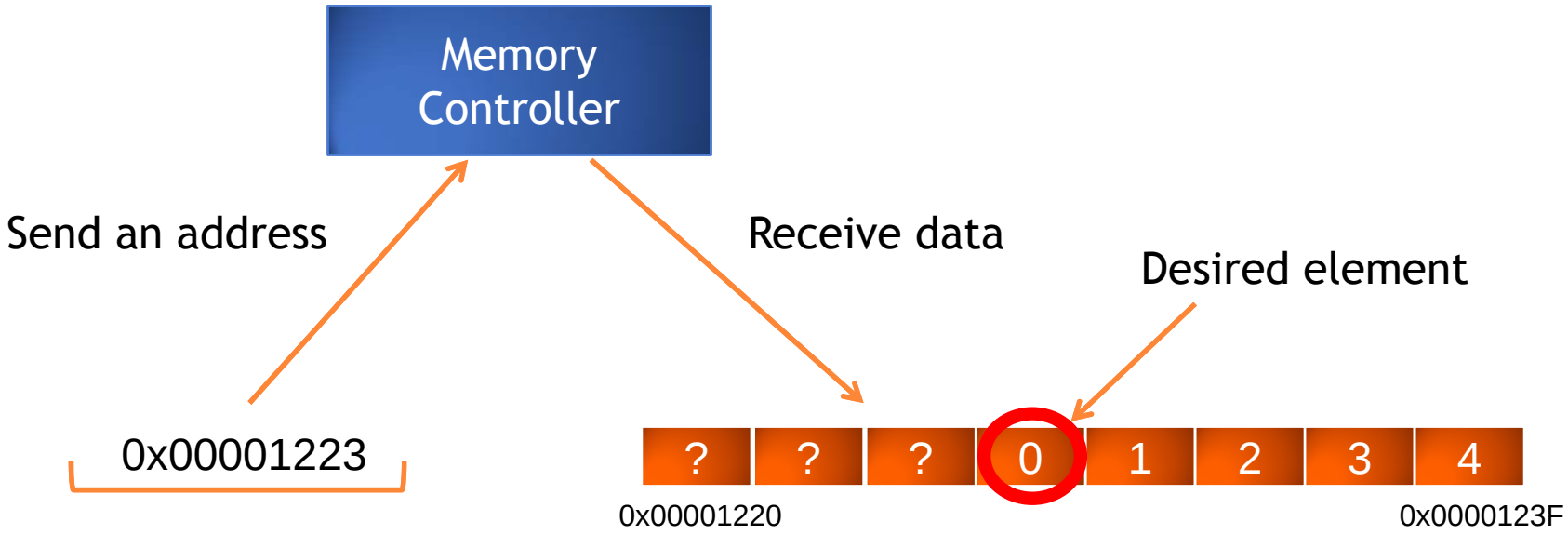
- Assume that the memory bus is 32-bytes (256-bits) wide
  - This is the width on a Radeon 5870 GPU
- The byte-addressable bus must make accesses that are aligned to the bus width, so the bottom 5 bits are masked off

Desired address:	`0x00001232`
Bus mask:	`0xFFFFFFE0`
Bus access:	`0x00001220`

- Any access in the range 0x00001220 to 0x0000123F will produce the address 0x00001220

Bus Addressing

- All data in the range 0x00001220 to 0x0000123F is returned on the bus
- In this case, 4 bytes are useful and 28 bytes are wasted



Outline

- What is OpenCL?
- OpenCL Architecture
  - Platform Model
  - Execution Model
  - Memory Model
  - Programming Model
- Case Studies
  - Image Rotation
  - Matrix Multiplication
- Memory Issues
  - Coalescing memory accesses
  - Memory bank conflicts
- Synchronization
  - Within a work group
  - Among commands in command queue(s)
- Profiling Using Events

Coalescing Memory Accesses

- To fully utilize the bus, GPUs combine the accesses of multiple threads into fewer requests when possible
- Consider the following OpenCL kernel code:

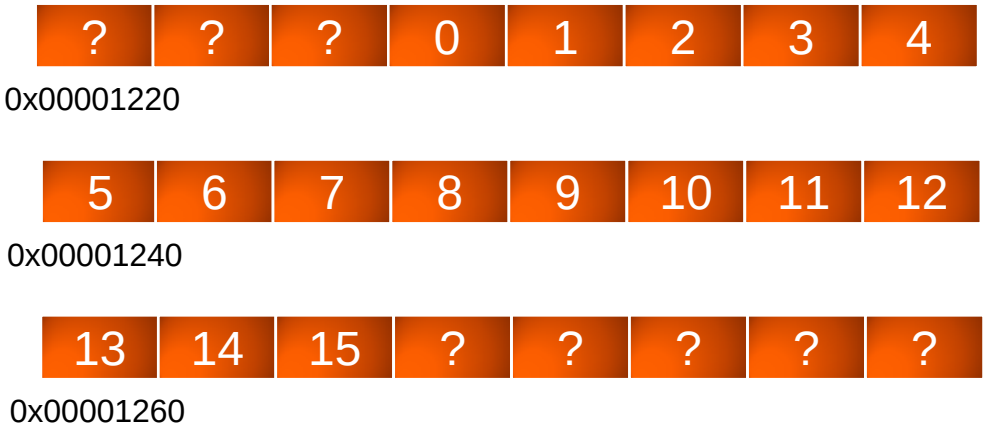
```
int tmp = X[get_global_id(0)];
```

Coalescing Memory Accesses

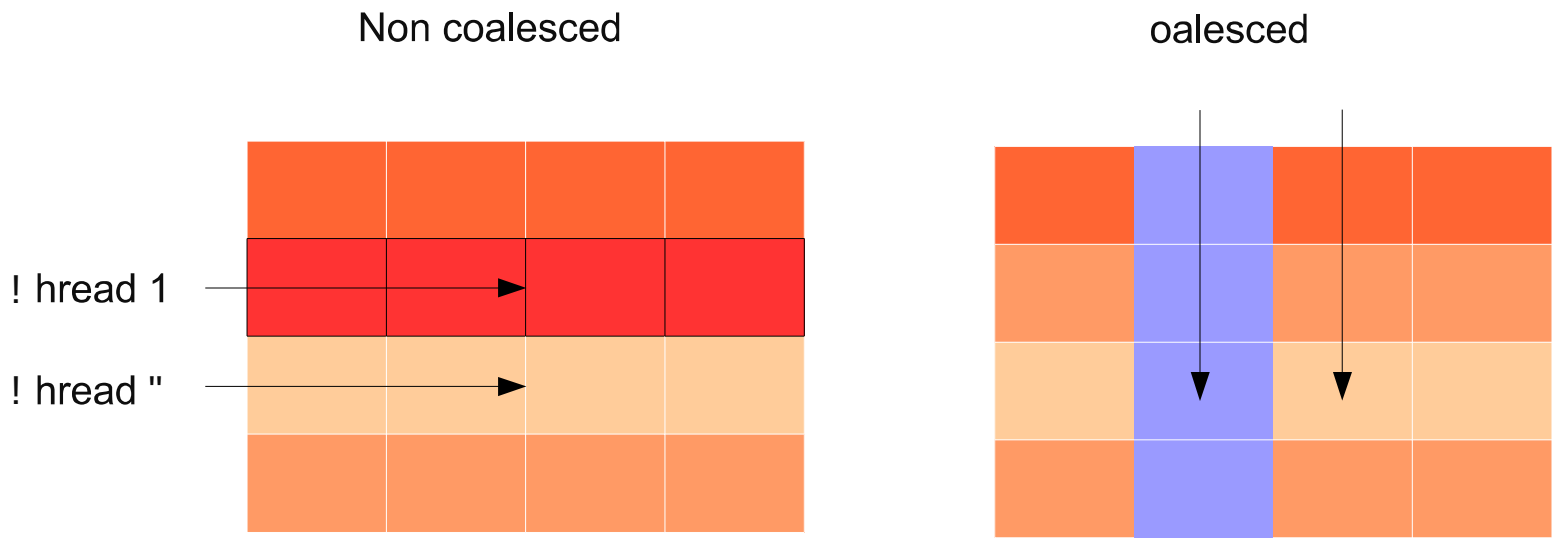
- Assuming that array X is the same array from the example, the first 16 threads will access addresses 0x00001232 through 0x00001272
- If each request was sent out individually, there would be 16 accesses total, with 64 useful bytes of data and 448 wasted bytes
  - Notice that each access in the same 32-byte range would return exactly the same data

Coalescing Memory Accesses

- When GPU threads access data in the same 32-byte range, multiple accesses are combined so that each range is only accessed once
  - Combining accesses is called *coalescing*
- For this example, 3 accesses are required
  - If the start of the array was 256-bit aligned, only two accesses would be required



Coalescing Memory Accesses



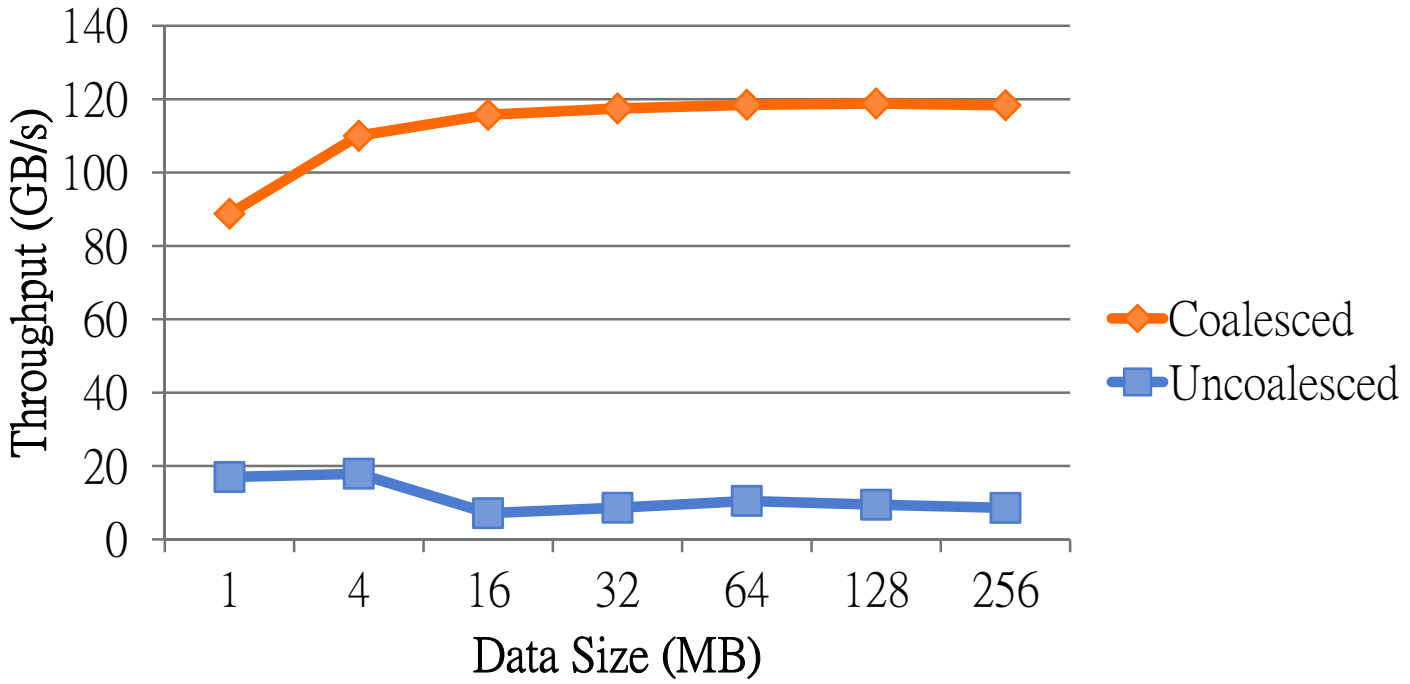
- On the case of the left, at iteration 0
  - Threads 1 to 4 read the element 0 of rows 1 to 4 (non consecutive data) → no accesses can be grouped
- On the case of the right, at iteration 0
  - Threads 1 to 4 read the element 0 to 4 of row0 (consecutive data) → accesses can be grouped
- The hardware is able to automatically detects when accesses can be grouped

Coalescing Memory Accesses

- Memory coalescing techniques are often used in conjunction with tiling techniques to allow the device to reach its performance potential by efficiently using the memory bandwidth

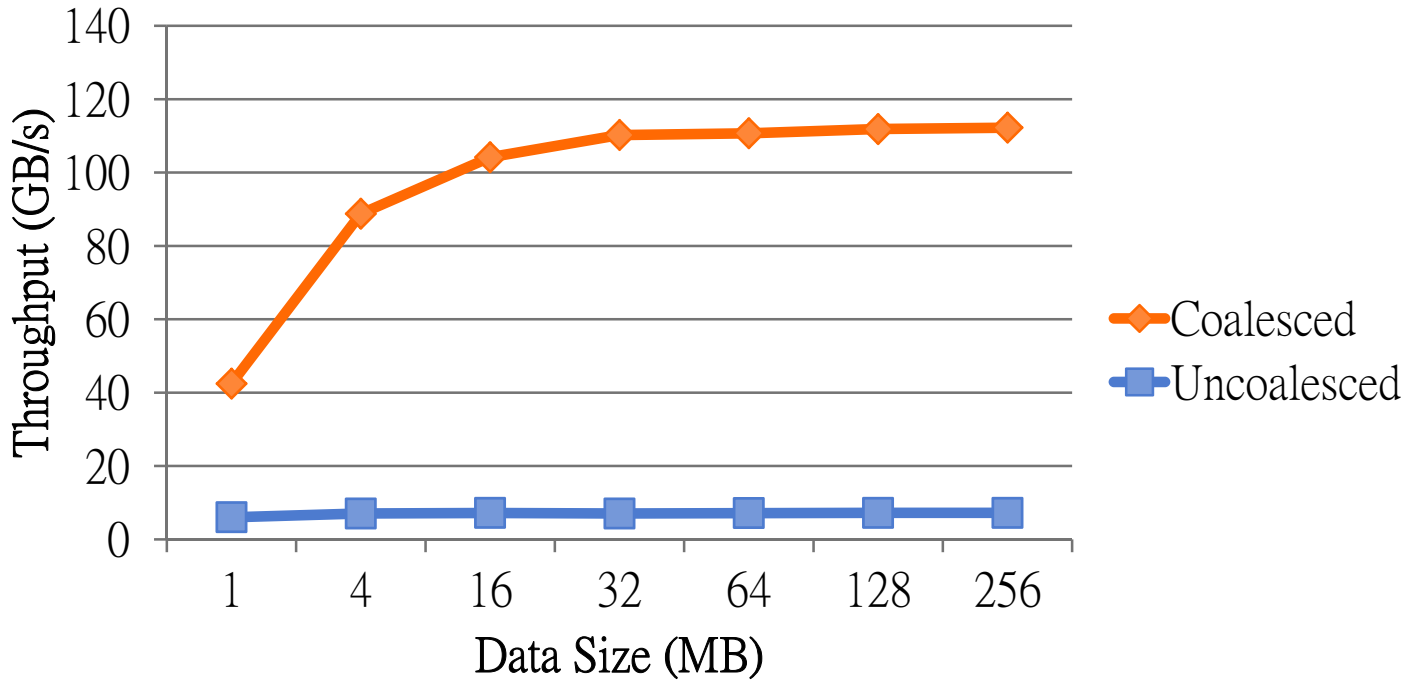
Coalescing Memory Accesses

- Global memory performance for a simple data copying kernel of entirely coalesced and entirely non-coalesced accesses on an ATI Radeon 5870



Coalescing Memory Accesses

- Global memory performance for a simple data copying kernel of entirely coalesced and entirely non-coalesced accesses on an NVIDIA GTX 285

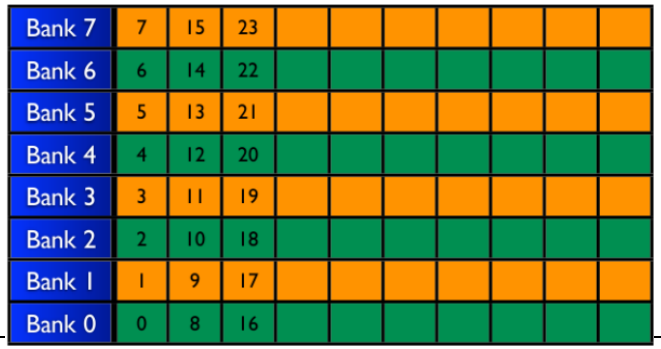


Outline

- What is OpenCL?
- OpenCL Architecture
  - Platform Model
  - Execution Model
  - Memory Model
  - Programming Model
- Case Studies
  - Image Rotation
  - Matrix Multiplication
- Memory Issues
  - Coalescing memory accesses
  - Memory bank conflicts
- Synchronization
  - Within a work group
  - Among commands in command queue(s)
- Profiling Using Events

Memory Banks

- Memory is made up of *banks*
  - Memory banks are the hardware units that actually store data
- The memory banks targeted by a memory access depend on the address of the data to be read/written
  - Note that on current GPUs, there are more memory banks than can be addressed at once by the global memory bus, so it is possible for different accesses to target different banks
    - Bank response time, not access requests, is the bottleneck
- Successive data are stored in successive banks (strides of 32-bit words on GPUs) so that a group of threads accessing successive elements will produce no bank conflicts

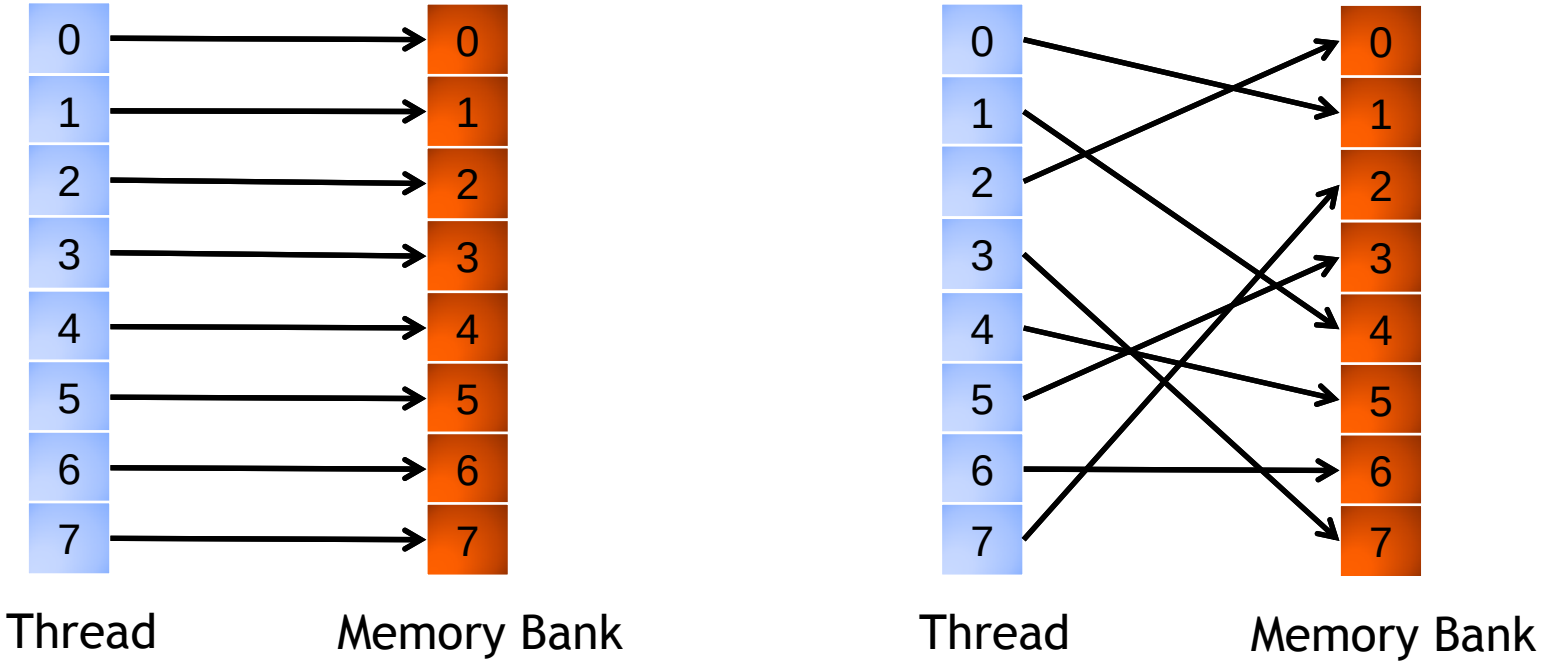


Bank Conflicts – Local Memory

- Accesses from successive threads should target different memory banks
  - Threads accessing sequentially increasing data will fall into this category
- Bank conflicts have the largest negative effect on local memory operations
  - Local memory does not require that accesses are to sequentially increasing elements
  - Local memory is on-chip, so the access latency is much shorter
- On AMD, a wavefront that generates bank conflicts stalls until all local memory operations complete
  - The hardware does not hide the stall by switching to another wavefront
- The examples in the following slides show local memory access patterns and whether conflicts are generated
  - For readability, only 8 memory banks are shown

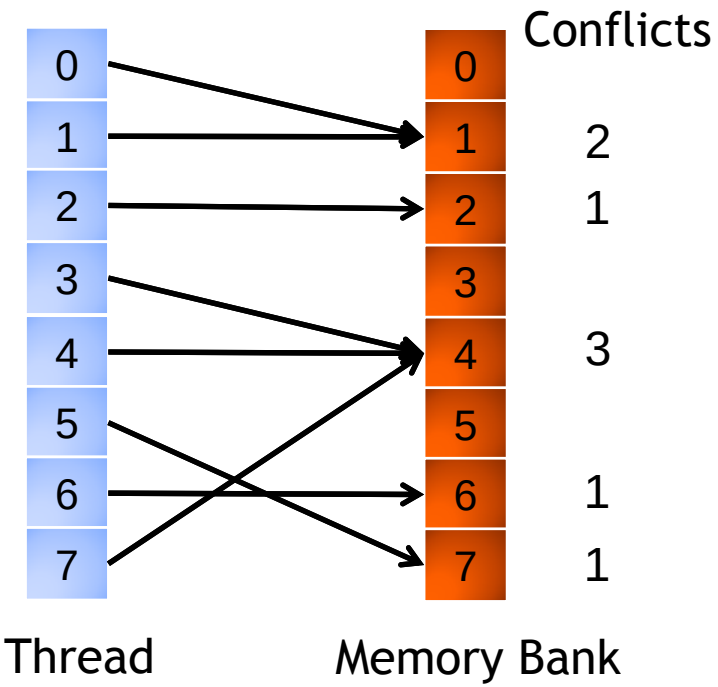
Bank Conflicts – Local Memory

- If there are no bank conflicts, each bank can return an element without any delays
  - Both of the following patterns will complete without stalls on current GPU hardware



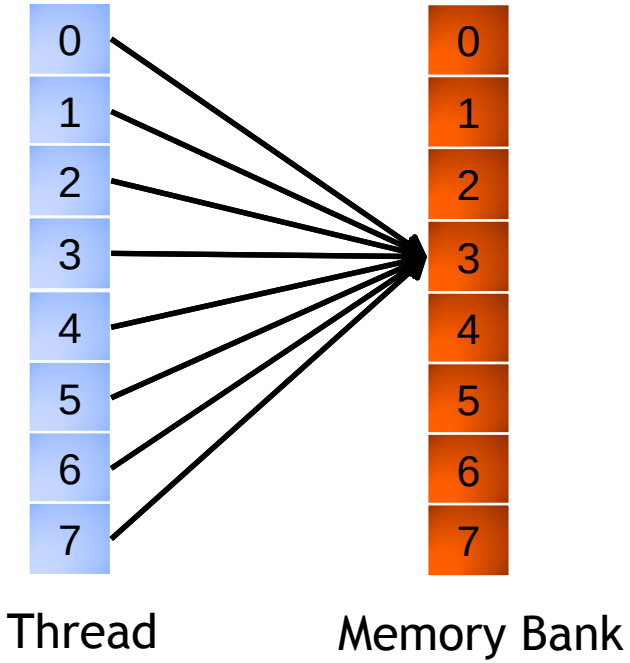
Bank Conflicts – Local Memory

- If multiple accesses occur to the same bank, then the bank with the most conflicts will determine the latency
  - The following pattern will take 3 times the access latency to complete



Bank Conflicts – Local Memory

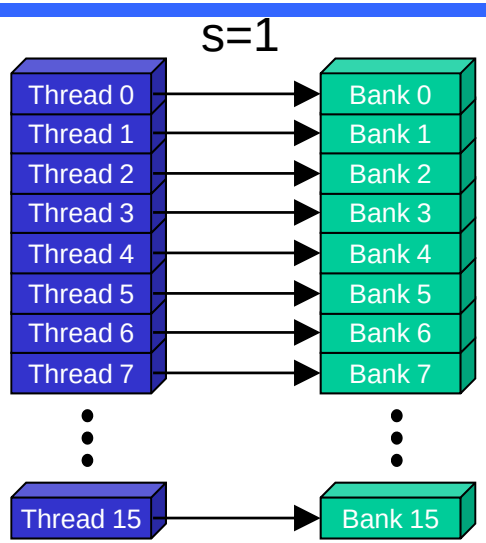
- If all accesses are to the same address, then the bank can perform a broadcast and no delay is incurred
  - The following will only take one access to complete assuming the same data element is accessed



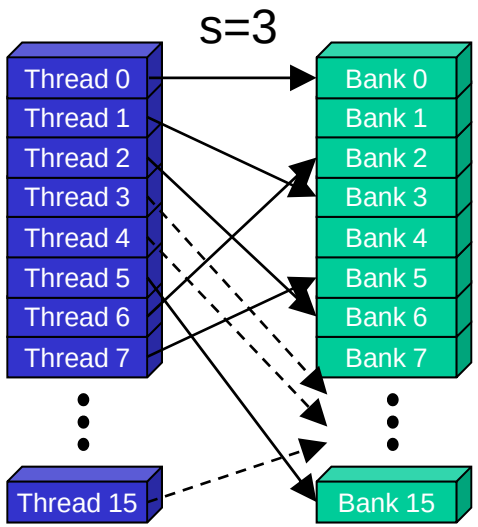
Linear Addressing

- Given:

```
__local float shared[256];
float foo =
  shared[get_group_id(0) + s *
                  get_local_id(0)];
```



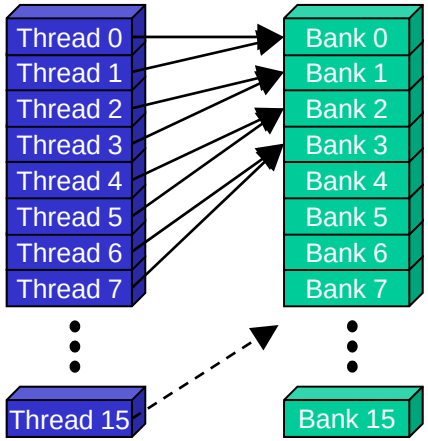
- This is only bank-conflict-free if s shares no common factors with the number of banks
  - 16 on G80, so s must be odd



Data Types and Bank Conflicts

- This has no conflicts if type of `shared` is 32-bits:

```
foo = shared[get_group_id(0) +
                          get_local_id(0)];
```

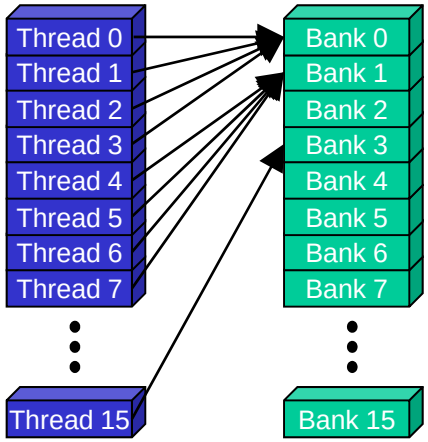


- But not if the data type is smaller
  - 2-way bank conflicts:

```
__shared__ short shared[];
foo = shared[get_group_id(0) +
                        get_local_id(0)];
```

  - 4-way bank conflicts:

```
__shared__ char shared[];
foo = shared[get_group_id(0) +
                        get_local_id(0)];
```



Bank Conflicts – Global Memory

- Bank conflicts in global memory rely on the same principles, however the global memory bus makes the impact of conflicts more subtle
  - Since accessing data in global memory requires that an entire bus-line be read, bank conflicts within a work-group have a similar effect as non-coalesced accesses
    - If threads reading from global memory had a bank conflict then by definition it manifest as a non-coalesced access
    - Not all non-coalesced accesses are bank conflicts, however
- The ideal case for global memory is when different work-groups read from different banks
  - In reality, this is a very low-level optimization and should not be prioritized when first writing a program

Remarks

- GPU memory is different than CPU memory
  - The goal is high throughput instead of low-latency
  - Memory bus for GPUs is much wider than bus for CPUs
    - Bus width: 256 vs 64 bits
- Memory access patterns have a huge impact on bus utilization
  - Low utilization means low performance
- Having coalesced memory accesses and avoiding bank conflicts are required for high performance code
- Specific hardware information (such as bus width, number of memory banks, bank width, and number of threads that coalesce memory requests) is GPU-specific and can be found in vendor documentation

Outline

- What is OpenCL?
- OpenCL Architecture
  - Platform Model
  - Execution Model
  - Memory Model
  - Programming Model
- Case Studies
  - Image Rotation
  - Matrix Multiplication
- Memory Issues
  - Coalescing memory accesses
  - Memory bank conflicts
- Synchronization
  - Within a work group
  - Among commands in command queue(s)
- Profiling Using Events

Synchronization

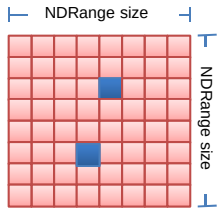
- Memory objects are allocated per context
- Changes made by one device are only guaranteed to be visible by another device at well-defined synchronization point
- Synchronization can only occur:
  - Between work-items in a single work-group
    - barrier()
    - mem_fence()
  - Among commands enqueued to command-queue(s) in a single context
    - A single out-of-order command queue
    - Multiple in-order/out-of-order command queues

Outline

- What is OpenCL?
- OpenCL Architecture
  - Platform Model
  - Execution Model
  - Memory Model
  - Programming Model
- Case Studies
  - Image Rotation
  - Matrix Multiplication
- Memory Issues
  - Coalescing memory accesses
  - Memory bank conflicts
- Synchronization
  - Within a work group
  - Among commands in command queue(s)
- Profiling Using Events

Synchronization within a Work-Group

- Synchronization within a work-group occurs via either a memory fence or barrier function
  - Does not permit such synchronization between different work-groups or work-items in separate work-groups
    - Work-items in different work-groups may coordinate execution through the use of atomic memory transactions



Barrier

- `void barrier(cl_mem_fence_flags flags)`
  - Blocks current work-item until all work-item in the work-group hits the barrier
  - Flags: `LOCAL_MEM_FENCE`/ `GLOBAL_MEM_FENCE`
    - Flush and ensure ordering for local/global memory

```
__kernel void fill_tiles(__global float* a,
                         __global float* b,
                         __global float* c) {
  // find our coordinates in the grid
  int row = get_global_id(1);
  int col = get_global_id(0);

  // allocate local memory for the workgroup
  __local float aTile[TILE_DIM_Y][TILE_DIM_X];
  __local float bTile[TILE_DIM_Y][TILE_DIM_X];

  // define the coordinates of this workitem
thread
  // in the 2D tile
  int y = get_local_id(1);
  int x = get_local_id(0);

  aTile[y][x] = a[row*N + col];
  bTile[y][x] = b[row*N + col];
  barrier(CLK_LOCAL_MEM_FENCE);

  //Note the change in tile location in bTile!
  c[row*N + col] = aTile[x][y] * bTile[y][x];
}
```

Memory Fence

- `void mem_fence(cl_mem_fence_flags flags)`
  - Flags: `CLK_LOCAL_MEM_FENCE/ CLK_GLOBAL_MEM_FENCE`
    - Flush and ensure ordering for local/global memory
  - Orders all preceding global or local (or both) reads and writes
    - Means all loads/stores committed to memory before any following loads/stores
  - mem_fence_write();
    - same as above, but only for stores
  - mem_fence_read();
    - same as above, but only for loads
- Different to barrier()
  - non-blocking

Terminologies: CUDA vs OpenCL

CUDA term	OpenCL term
__syncthreads()	barrier()
__threadfence()	No direct equivalent
__threadfence_block()	mem_fence()
No direct equivalent	read_mem_fence()
No direct equivalent	write_mem_fence()

Outline

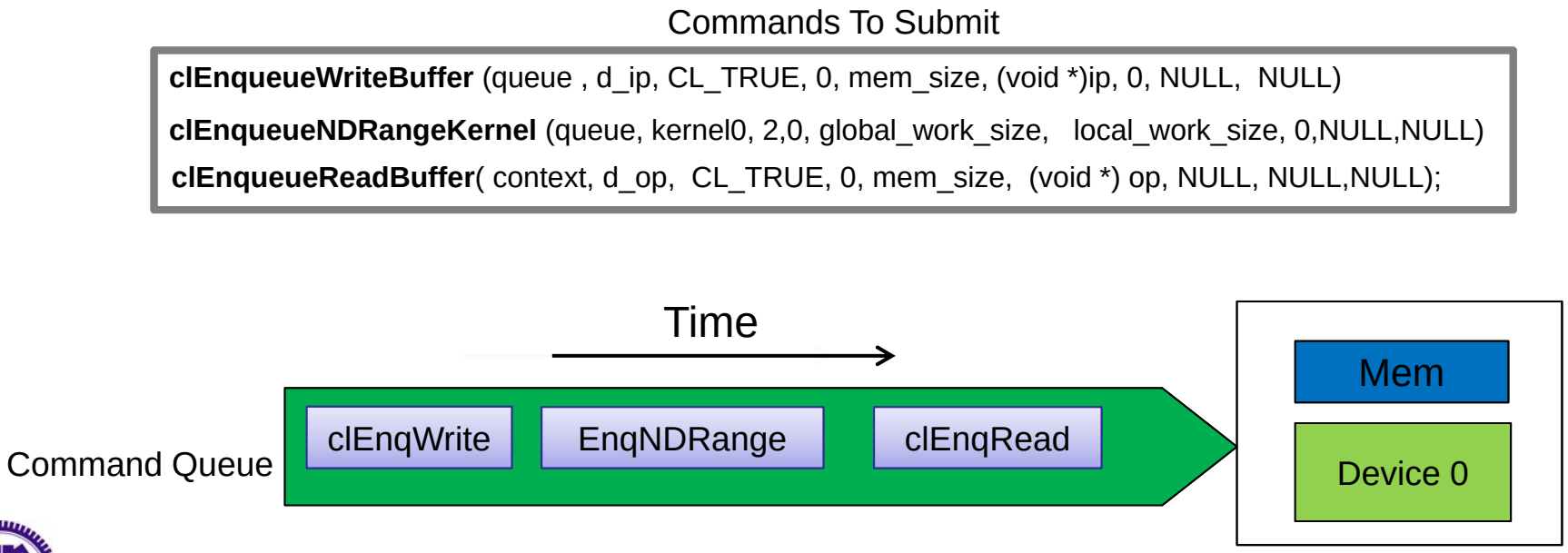
- What is OpenCL?
- OpenCL Architecture
  - Platform Model
  - Execution Model
  - Memory Model
  - Programming Model
- Case Studies
  - Image Rotation
  - Matrix Multiplication
- Memory Issues
  - Coalescing memory accesses
  - Memory bank conflicts
- Synchronization
  - Within a work group
  - Among commands in command queue(s)
- Profiling Using Events

Types of Command Queue

- In-order
  - Each command in the queue executes only when the preceding command has completed (including memory writes)
  - Commands executed in the order of issuing
- Out-of-order
  - No guaranteed order of completion for commands
  - Execution dependent only on its event list completion

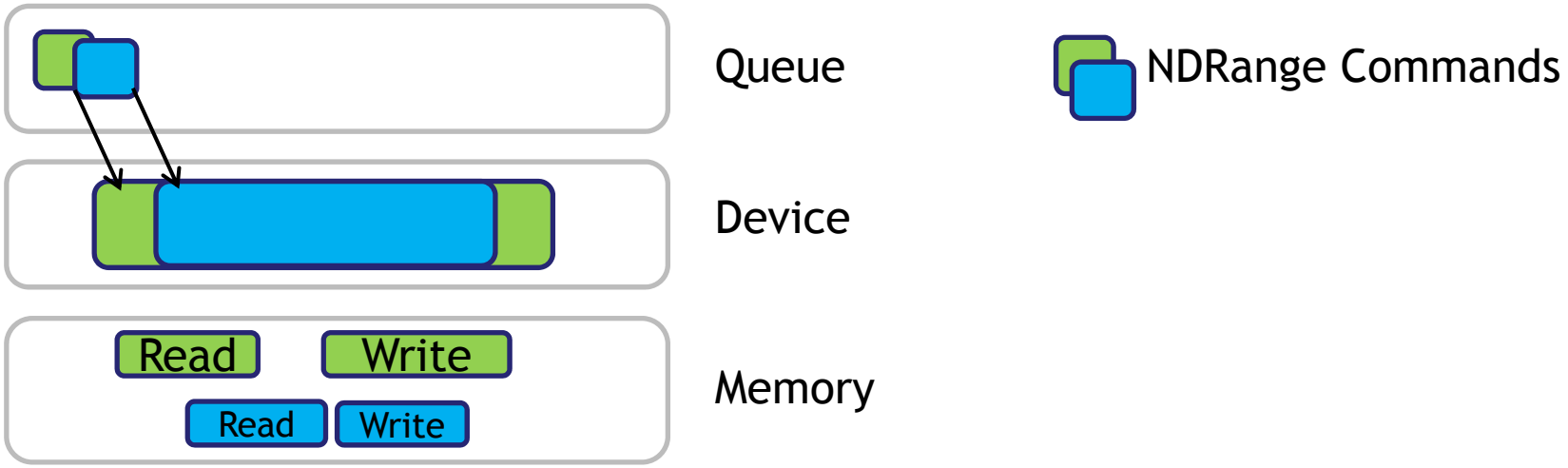
In-Order Execution

- In an in-order command queue, each command executes after the previous one has finished
  - For the set of commands shown, the read from the device would start after the kernel call has finished
- Memory transactions have consistent view



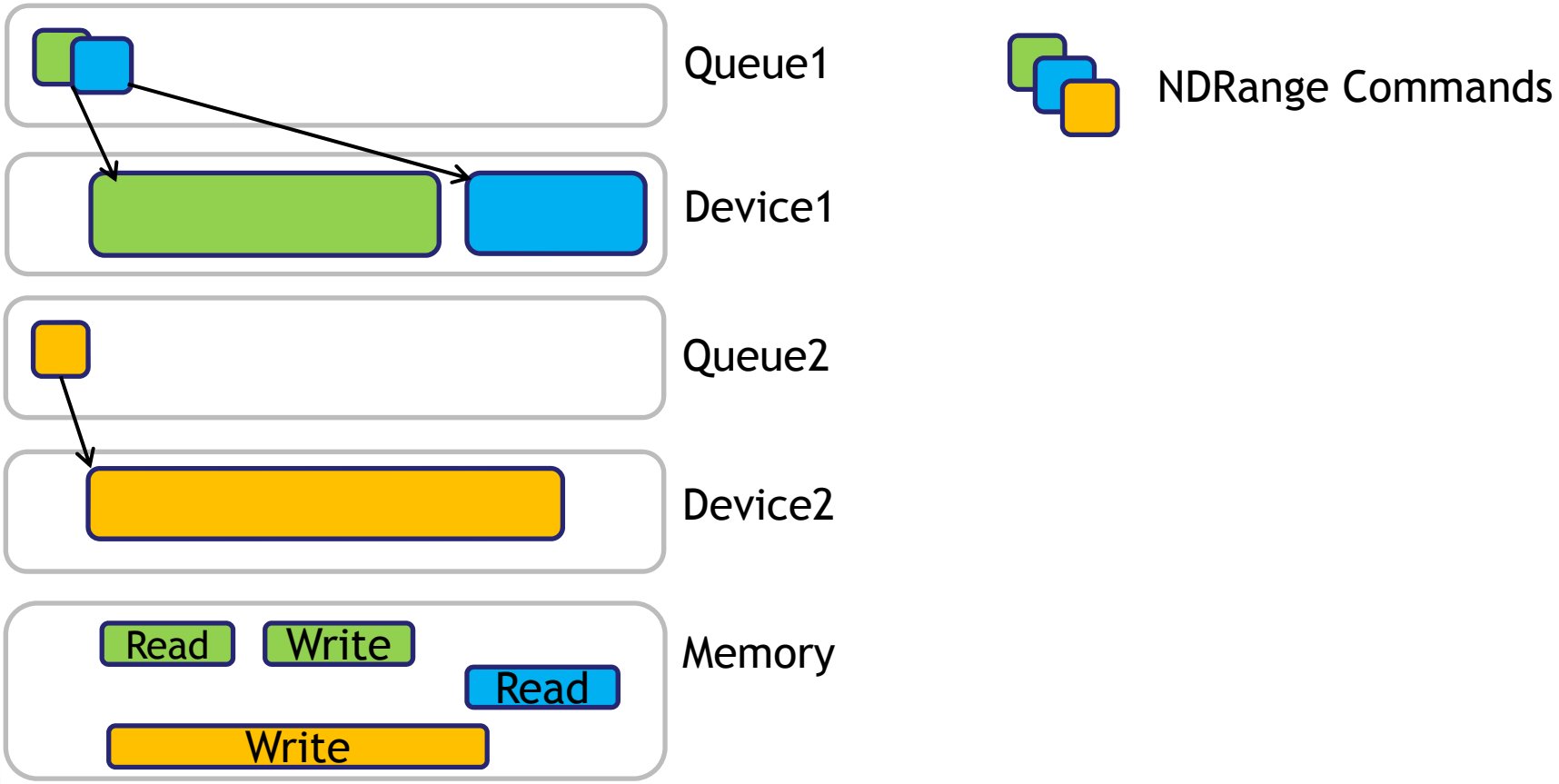
Single-Device Out-of-Order Execution

- All commands execute on single device
- Using out-of-order command queue
  - Execution order is not guaranteed
- All memory operations occur in single memory pool (context)
  - Since device starts executing commands as soon as it can
  - The memory transactions may overlap and clobber data



Multi-Device In-Order Execution

- Multiple in-order queues, 1 context, multiple devices
- Multiple devices modify a single memory pool
  - May incur clobbered data!



Synchronization Among Commands

- Several synchronization mechanisms help control the command execution
  - Command-queue control
    - Coarse-grained synchronization
  - Command-queue barrier
    - Ordering within a single queue
    - Any resulting changes to memory are available to the following commands in the queue
  - Events
    - Ordering between or within queues
    - Commands in the queue return an event, containing the identification of the command as well as the memory object updated
    - In this way, it is sure that the commands followed by waiting event accesses the updated memory objects before they execute

Command-Queue Control

- clFlush()
  - Send all commands in the queue to the compute device
  - No guarantee that they will be complete when clFlush returns
- clFinish()
  - Waits for all commands in the command queue to complete before proceeding (host blocks on this call)

Command-Queue Barrier

- Sequence of commands scheduled for execution on a specific device
  - Several queues can execute on the same device
  - Function: clEnqueue*()
- Command-queue barrier
  - clEnqueueBarrier()
  - Commands after the barrier start executing only after all commands before the barrier have completed

Events

- OpenCL provides events, with the ability to synchronize on a given event to enforce the correct order of execution
- OpenCL events are data-types defined by the specification for storing timing information returned by the device
  - It determines a specific command status

Event Objects

- All clEnqueue*() methods can return event objects
  - clEnqueueNDRangeKernel
  - clEnqueueTask
  - clEnqueueNativeKernel
  - clEnqueueCopyImageToBuffer
  - clEnqueueCopyBufferToImage
  - clEnqueueCopyBufferToImage
  - clEnqueueRead {Image|Buffer}
  - clEnqueueWrite {Image|Buffer}
  - clEnqueueMap {Image|Buffer}
  - clEnqueueCopy {Image|Buffer}
  - clEnqueueCopyImageToBuffer
  - clEnqueueCopyBufferToImage
- Event wait list can be specified while calling the enqueue method
  - The command will be hung in the command queue until the events specified in wait list are fulfilled

Capturing Event Information

- The function

```
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)
```

returns information about the event object

- *param_name* specifies the info to query, includes the *command queue*, *context*, *type of command* associated with *event* and the command execution status

CL_EVENT_COMMAND_EXECUTION_STATUS	Description
CL_QUEUED	Command has been enqueued in the command queue
CL_SUBMITTED	Enqueued command has been submitted by the host to the device
CL_RUNNING	Device is currently executing this command
CL_COMPLETED	The command has completed or error

Examples for Using Events Status

- Busy waiting

```
cl_event readEvt;
clEnqueueReadBuffer(commandQueue, data, …..,NULL, &readEvt);
cl_int eventStatus = CL_QUEUED;
while(eventStatus != CL_COMPLETE)
//constantly query the command status
//use a while loop to wait up until the status become completed
    {
        clGetEventInfo(
                        readEvt,
CL_EVENT_COMMAND_EXECUTION_STATUS,
                        sizeof(cl_int), &eventStatus,  NULL);
    }
…
```

Registering Callback Function

- The function

```
cl_int      clSetEventCallback (cl_event event,
                                cl_int command_exec_callback_type,
                                void (CL_CALLBACK *pfn_event_notify)(cl_event event,
                                                                     cl_int event_command_exec_status,
                                                                     void *user_data),
                                void *user_data)
```

registers a callback function for a specific command execution status

- E.g. `clSetEventCallback(wrtEvt, CL_COMPLETE, onWriteComplete, data)`

Examples for Using Event Callbacks

- Setting an event handler

```
{
…
cl_event readEvt;
clEnqueueReadBuffer(commandQueue, data, …., &readEvt);
//register a handler for readEvt
clSetEventCallback(readEvt, CL_COMPLETE, onDataReadComplete,
     (void *)&param);
…
}
//the callback function (event handler) is here
onDataReadComplete( cl_event evt, cl_int cmd_exe_status, void
*param)
{
 …//do somthing
}
```

Waiting for Events

- The function

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

puts host thread to wait for the events in the *event_list* to complete

- CPU waits until all of the status of the events in the list are CL_COMPLETE or a negative value

Examples for Using Event Lists

- Acting as a sync point of a command group (1/2)

```
cl_event wrtEvts[2];
clEnqueueWriteBuffer(commandQueue, data1, … , &wrtEvt[0]);
clEnqueueWriteBuffer(commandQueue, data2, … , &wrtEvt[1]);
clWaitForEvents(2, wrtEvts); //wait for two events
// clEnqueueBarrier(commandQueue);
clEnqueueReadBuffer(commandQueue, data1, … , &readEvt[1]);
```

- Acting as a sync point of a group of commands (2/2)

```
cl_event wrtEvts[2];
clEnqueueWriteBuffer(commandQueue, data1, … , &wrtEvts[0]);
clEnqueueWriteBuffer(commandQueue, data2, … , &wrtEvts[1]);
/*specify a wait list wrtEvts to wait until all events are
fulfilled*/
clEnqueueReadBuffer(commandQueue, data1, … , 2 ,wrtEvts, NULL);
```

User Events

- The function defined in OpenCL 1.1

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

creates a user event object.

  - User events are only set by user program, not clEnqueue* commands
  - When a user event is created, the status of the event is set to CL_SUBMITTED

- The function

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

sets the execution status of a user event.

  - A user event can only be set to CL_COMPLETE once

Examples for Using User Events

- Using user events (1/2)

```
ev1 = clCreateUserEvent(ctx, NULL);
clEnqueueWriteBuffer(cq, buf1, CL_FALSE,…,1,&ev1, NULL);
//wait for user event ev1
clEnqueueWriteBuffer(cq, buf2, CL_FALSE,…);
clSetUserEventStatus(ev1, CL_COMPLETE); //after setting to
CL_COMPLETE, the first clEnqueueWriteBuffer will work
clReleaseMemObject(buf2);
```

- Using user events (2/2)

```
ev1 = clCreateUserEvent(ctx, NULL);
clEnqueueWriteBuffer(cq, buf1, CL_FALSE,…,1,&ev1, NULL);
//wait for user event ev1
userDefinedAction(cq, buf2, CL_FALSE,…);

userDefinedAction()
{
…//something, for instance waiting a computation request is
arrived
clSetUserEventStatus(ev1, CL_COMPLETE);
}
```

Releasing Events

- The function

  cl_int **clReleaseEvent** (cl_event *event*)

  release an event.
  - Must be careful that there's no command wait on this event
  - The commands will continue to wait for the event status to reach CL_COMPLETE or error, even the event is released
    - Causes deadlock

Outline

- What is OpenCL?
- OpenCL Architecture
  - Platform Model
  - Execution Model
  - Memory Model
  - Programming Model
- Case Studies
  - Image Rotation
  - Matrix Multiplication
- Memory Issues
  - Coalescing memory accesses
  - Memory bank conflicts
- Synchronization
  - Within a work group
  - Among commands in command queue(s)
- Profiling Using Events

Capturing Profiling Information

- If profiling is enabled, the function

```
cl_int        clGetEventProfilingInfo (cl_event event,
                                       cl_profiling_info param_name,
                                       size_t param_value_size,
                                       void *param_value,
                                       size_t *param_value_size_ret)
```

provides profiling information for the command associated with *event*

- *param_name* specifies the info to query (next slide)
- The `clGetEventProfilingInfo` will return the data in the space pointed by *param_value*
- The size of the return value space must larger then 64 bits

Capturing Profiling Information

```
cl_int      clGetEventProfilingInfo (cl_event event,
                                     cl_profiling_info param_name,
                                     size_t param_value_size,
                                     void *param_value,
                                     size_t *param_value_size_ret)
```

- The following table shows the event types info that can be queried by users

Event Type	Description
CL_PROFILING_COMMAND_QUEUED	Command is enqueued in a command-queue by the host.
CL_PROFILING_COMMAND_SUBMIT	Command is submitted by the host to the device associated with the command queue.
CL_PROFILING_COMMAND_START	Command starts execution on device.
CL_PROFILING_COMMAND_END	Command has finished execution on device.

Examples for Using Events

- Profiling a command

```
cl_event ndrEvt;
//enqueue a kernel command
clEnqueueNDRangeKernel( commandQueue, … , &ndrEvt);
//wait for command to finish
clWaitForEvents(1, &ndrEvt);
clGetEventProfilingInfo(ndrEvt, CL_PROFILING_COMMAND_START,
                              sizeof(cl_ulong), &kstartTime, 0);

clGetEventProfilingInfo(ndrEvt, CL_PROFILING_COMMAND_END,
                               sizeof(cl_ulong), &kendTime, 0);
/* Print performance numbers */
kernelTime = 1e-6 * (kendTime - kstartTime);
std::cout << "KernelTime (ms) : " << kernelTimes << std::endl;
```

Summary

- OpenCL provides an interface for the interaction of hosts with accelerator devices
- A context is created that contains all of the information and data required to execute an OpenCL program
  - Memory objects are created that can be moved on and off devices
  - Command queues allow the host to request operations to be performed by the device
  - Programs and kernels contain the code that devices need to execute