

Introduction to OpenCL

A Collaboration Between
David Kaeli, Northeastern University
Benedict R. Gaster, AMD
© 2011

Instructor Notes

- This is a straight-forward lecture. It introduces the OpenCL specification while building a simple vector addition program
 - The Mona Lisa images in the slides may be misleading in that we are not actually using OpenCL images, but they were nicer to look at than diagrams of buffers
- It would probably be a good idea to open up the example code and walk through it along with the lecture

OpenCL Architecture

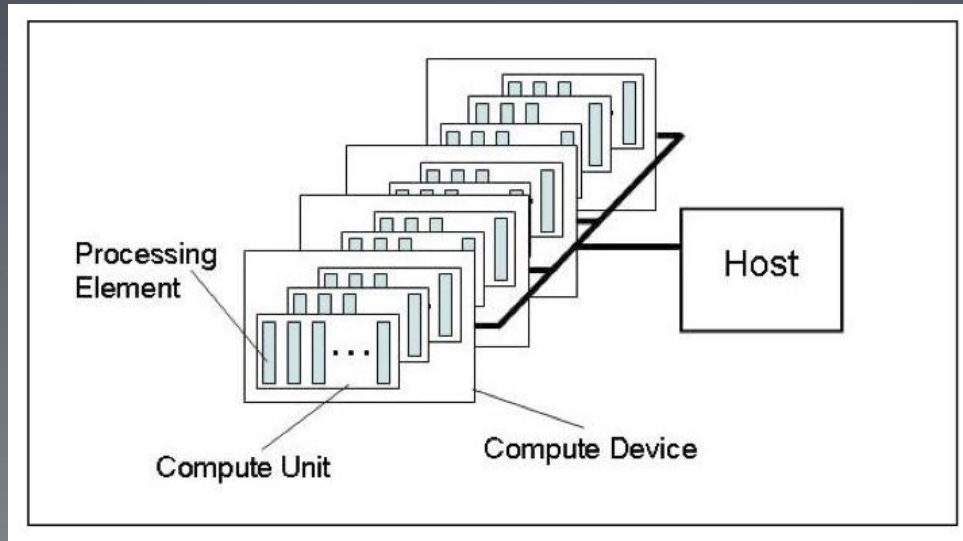
- OpenCL allows parallel computing on heterogeneous devices
 - CPUs, GPUs, other processors (Cell, DSPs, etc)
 - Provides portable accelerated code
- Defined in four parts
 - Platform Model
 - Execution Model
 - Memory Model
 - Programming Model
 - (We're going to diverge from this structure a bit)

Platform Model

- 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

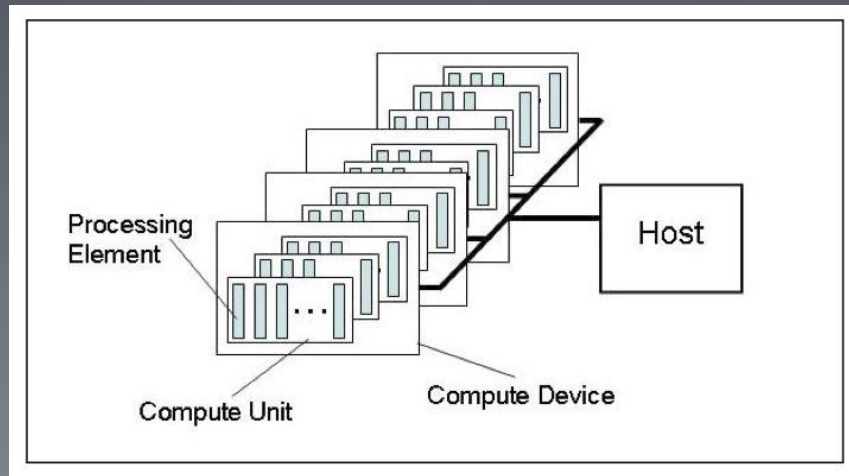
Platform Model

- 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



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

```
clGetDeviceIDs4 (cl_platform_id platform,  
                 cl_device_type device_type,  
                 cl_uint num_entries,  
                 cl_device_id *devices,  
                 cl_uint *num_devices)
```

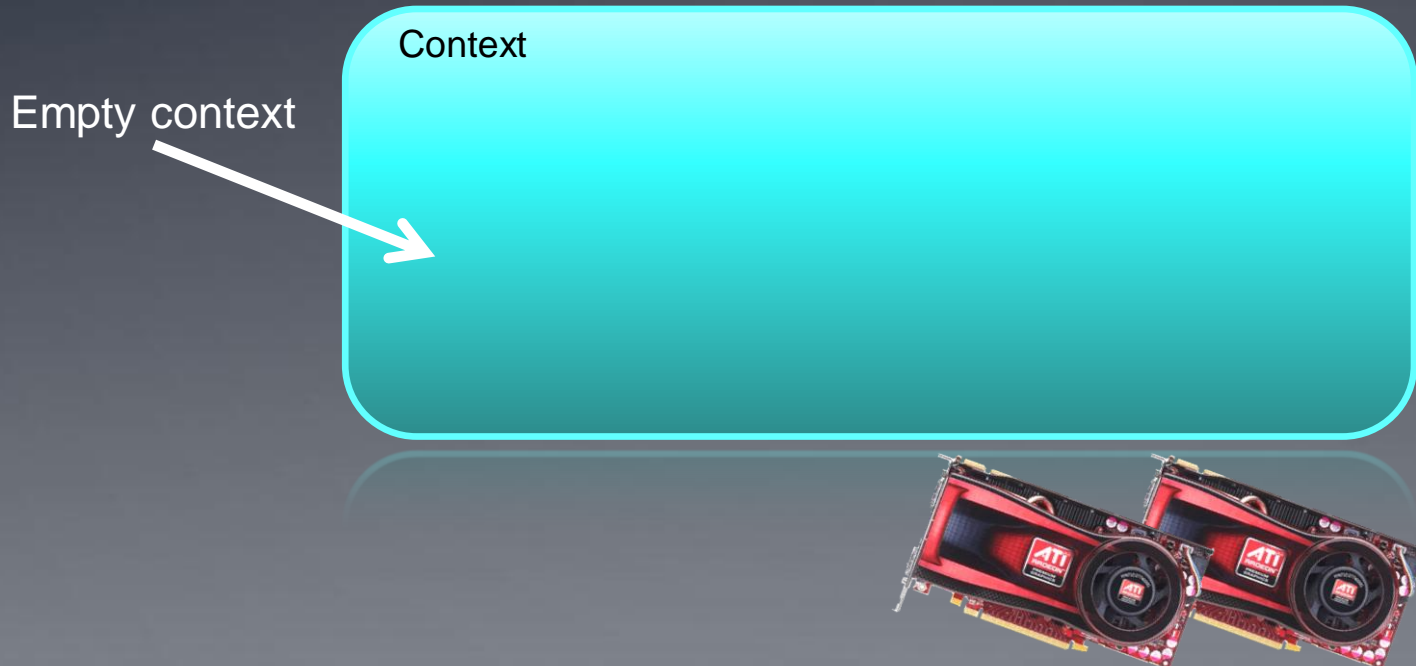
- We can specify which types of devices we are interested in (e.g. all devices, CPUs only, GPUs only)
- This call is performed twice as with `clGetPlatformIDs`
 - The first call is to determine the number of devices, the second retrieves the device objects

Contexts

- A context refers to the environment for managing OpenCL objects and resources
- 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

- 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

```
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

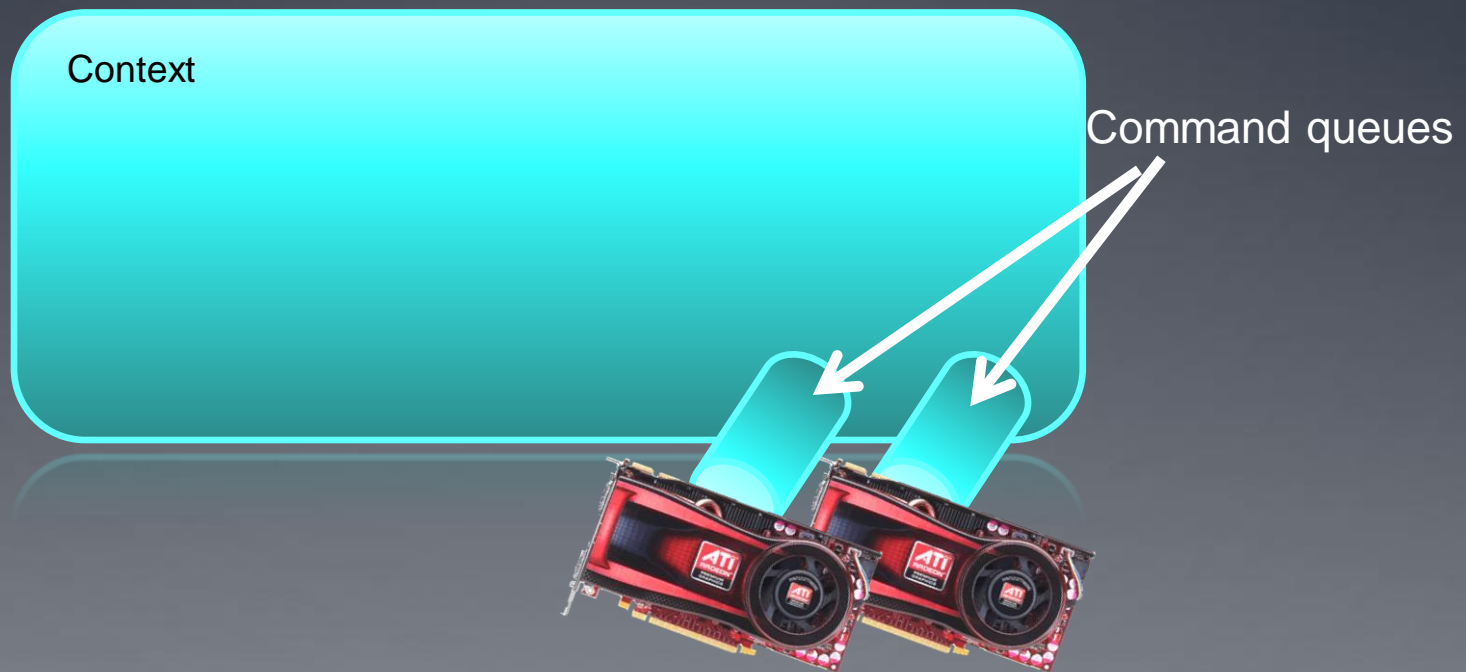
Command Queues

```
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 in a later lecture) and will create some overhead

Command Queues

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



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

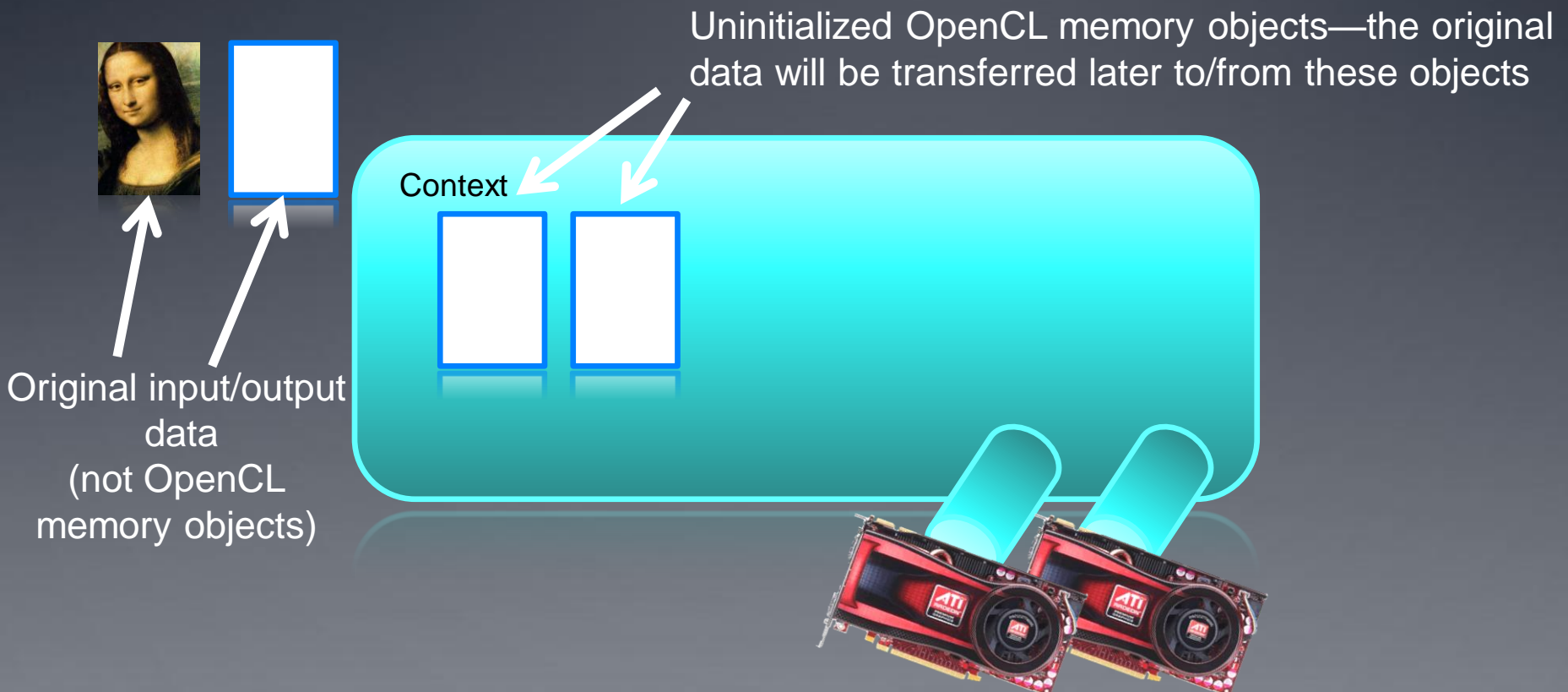
Creating buffers

```
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 and will be covered in a later lecture
- 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 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)
```

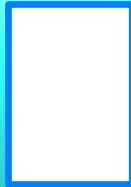
- This command initializes the OpenCL memory object and writes data to the device associated with the command queue
 - The command will write data from a host pointer (*ptr*) to the device
- The *blocking_write* parameter specifies whether or not the command should return before the data transfer is complete
- Events (discussed in another lecture) 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)



Context



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

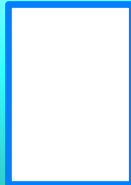
- A program object is basically a collection of OpenCL kernels
 - Can be source code (text) or precompiled binary
 - Can also contain constant data and auxiliary functions
- Creating a program object requires either reading in a string (source code) or a precompiled binary
- To compile the program
 - 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



Context



Program

```
void  
usage (char *name)  
{  
    printf ("usage:%s\n",  
           printf ("h\n -s [-w file",  
           name);  
    printf ("[-w] [-h] ");  
    printf ("[-w what] [-v]  
           [-w file [open]]");  
}
```

.CL



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

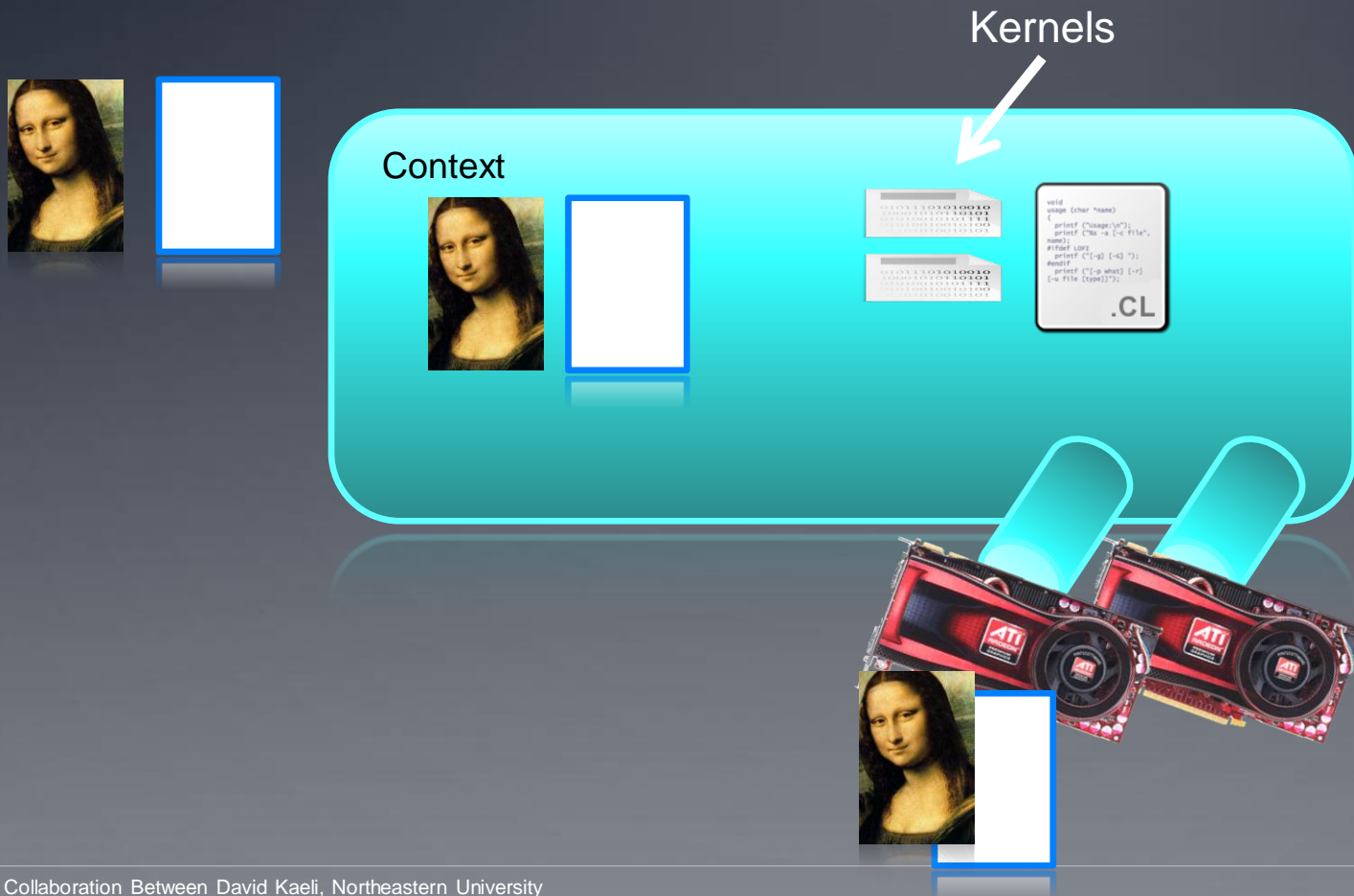
- If a program fails to compile, OpenCL requires the programmer to explicitly ask for compiler output
 - A compilation failure is determined by an error value returned from `clBuildProgram()`
 - Calling `clGetProgramBuildInfo()` with the program object and the parameter `CL_PROGRAM_BUILD_STATUS` returns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



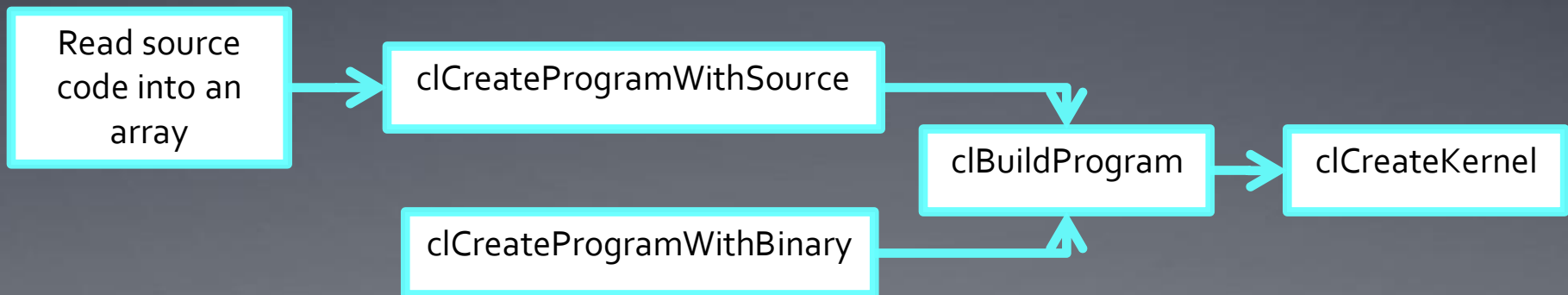
Kernels

```
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

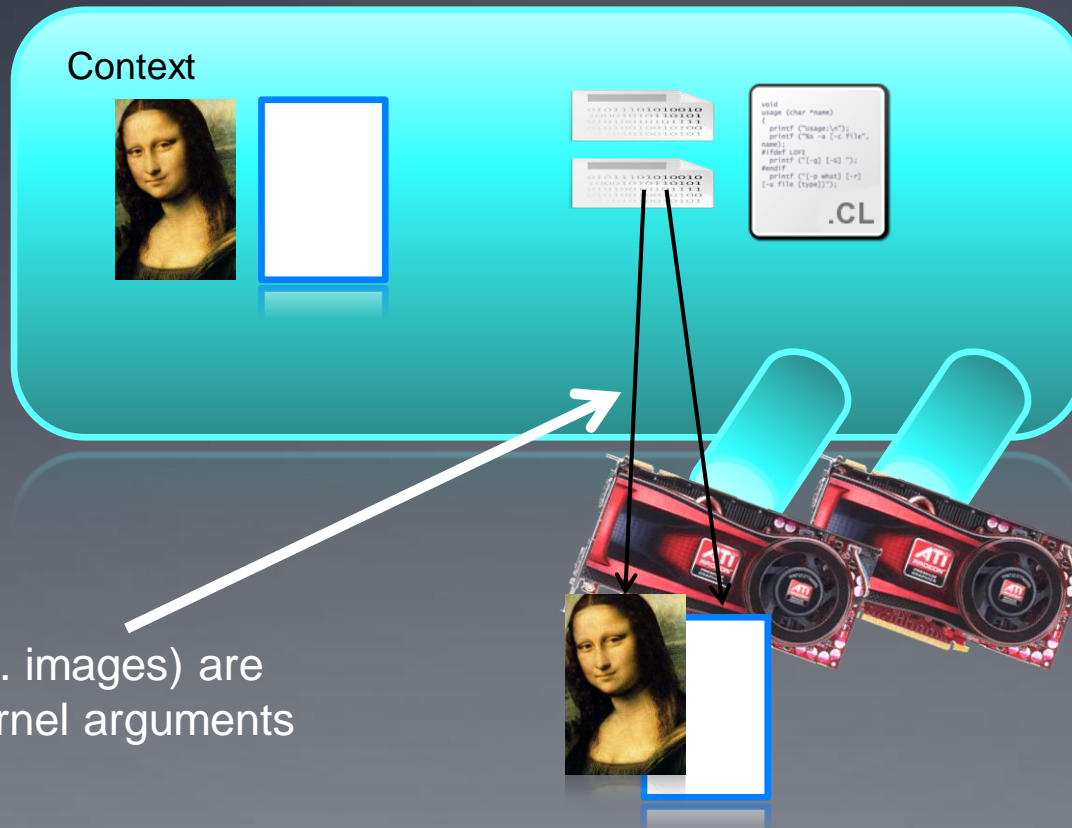
- Kernel arguments are set by repeated calls to `clSetKernelArgs`

```
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

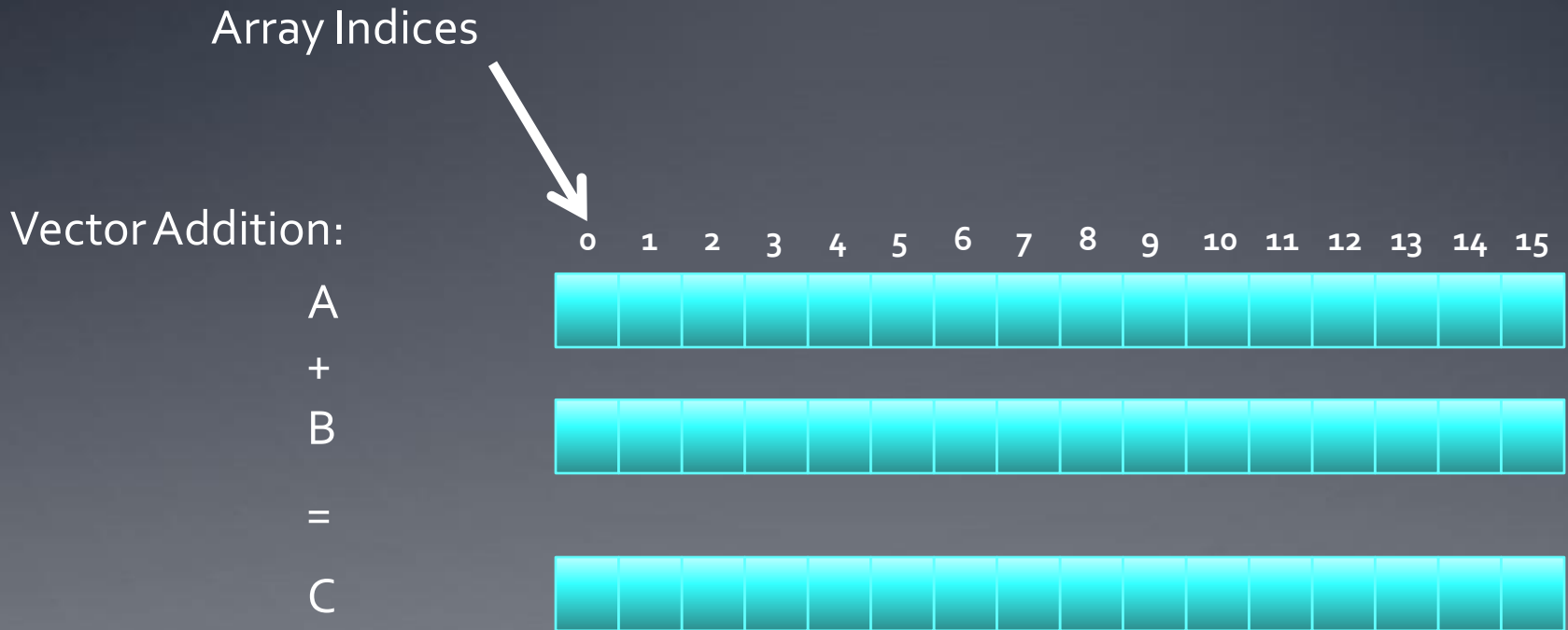


Thread Structure

- 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

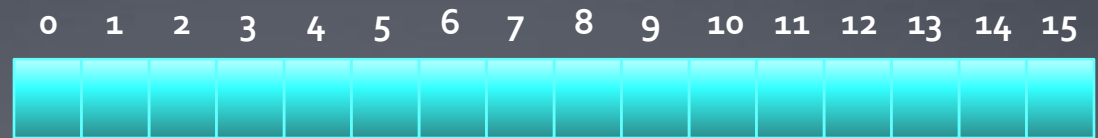
- Create thread structure to match the problem
 - 1-dimensional problem in this case

Thread structure:



Vector Addition:

A



+

B



=

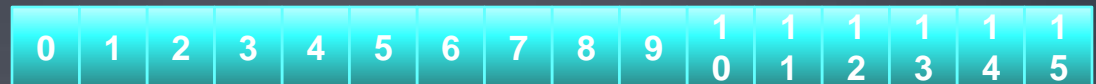
C



Thread Structure

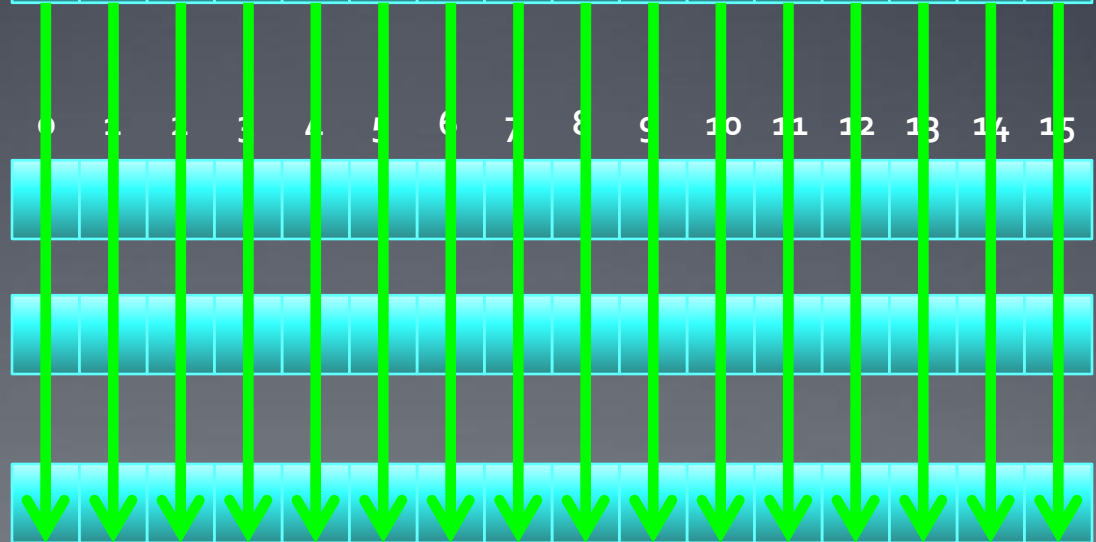
- Each thread is responsible for adding the indices corresponding to its ID

Thread structure:



Vector Addition:

A
+
B
=
C

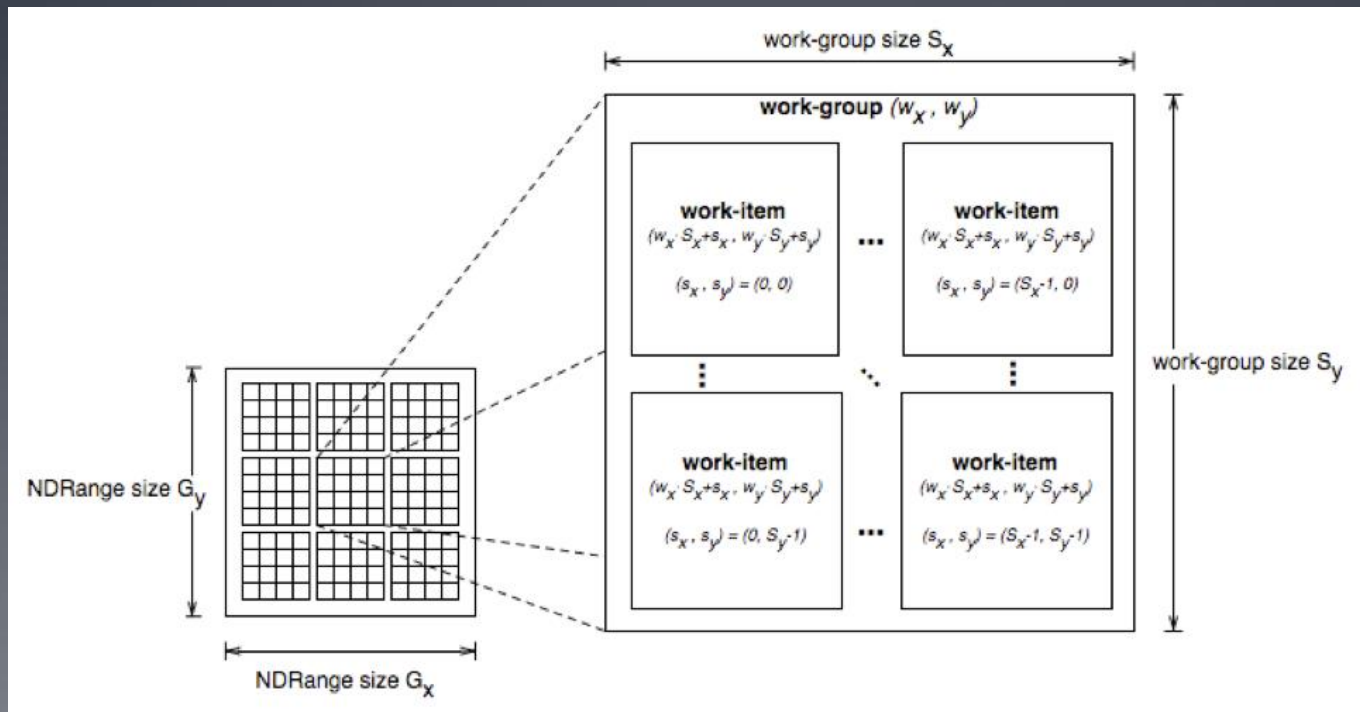


Thread Structure

- 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



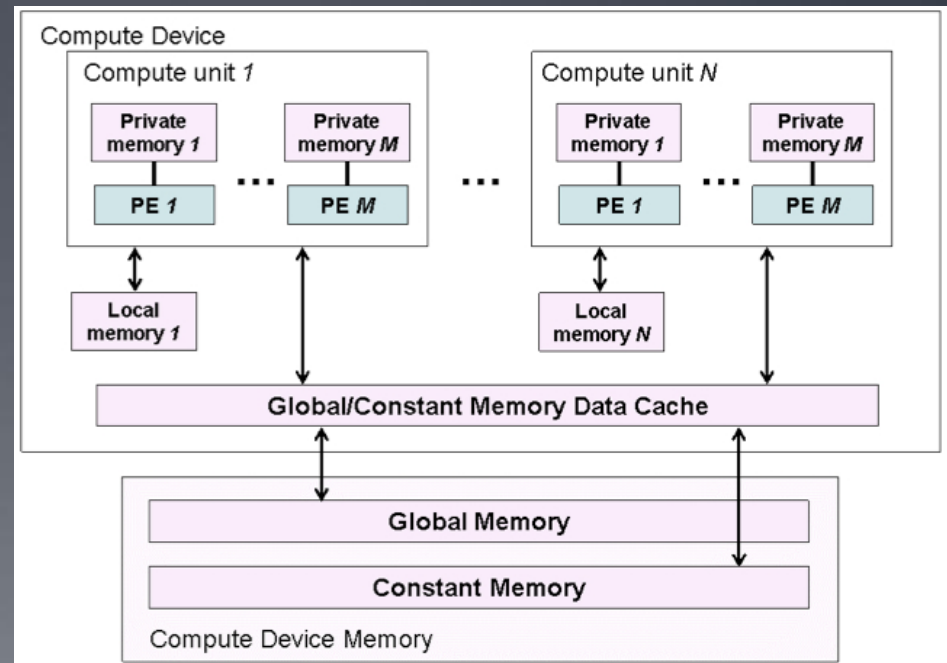
Thread Structure

- 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)`

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



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

Example Kernel

- Simple vector addition kernel:

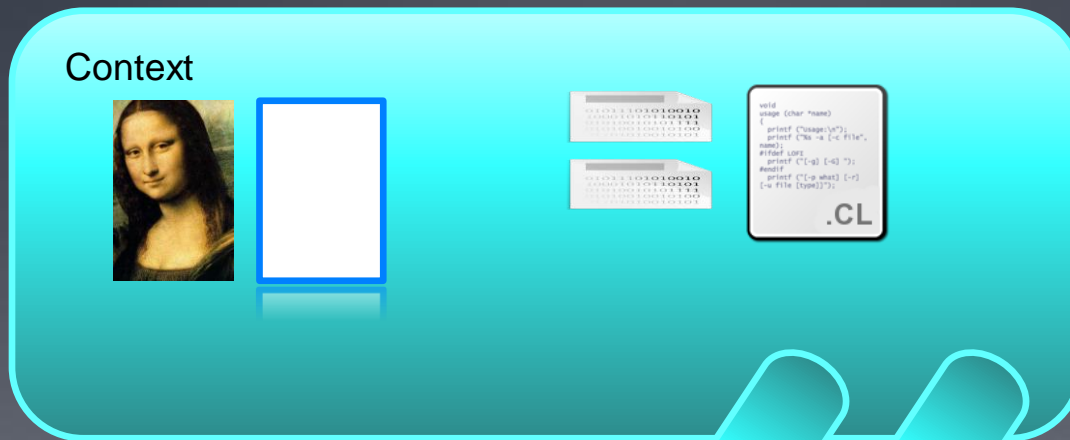
```
__kernel
void vecadd(__global int* A,
            __global int* B,
            __global int* C) {
    int tid = get_global_id(0);
    C[tid] = A[tid] + B[tid];
}
```

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 it 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



An index space of threads is created (dimensions match the data)

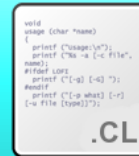


Executing the Kernel

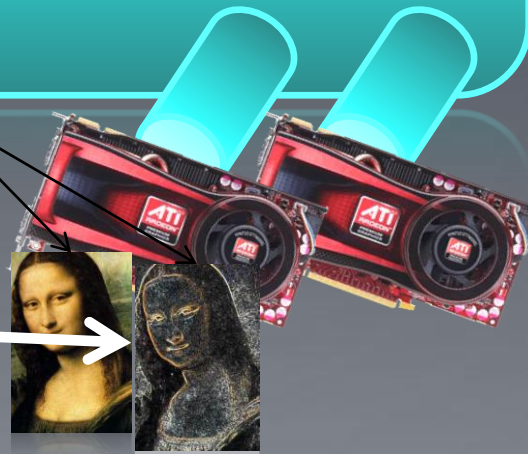
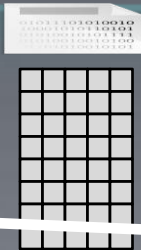
- A thread structure defined by the index-space that is created
 - Each thread executes the same kernel on different data



Context



Each thread
executes
the kernel



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

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



Copied back
from GPU



Releasing Resources

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

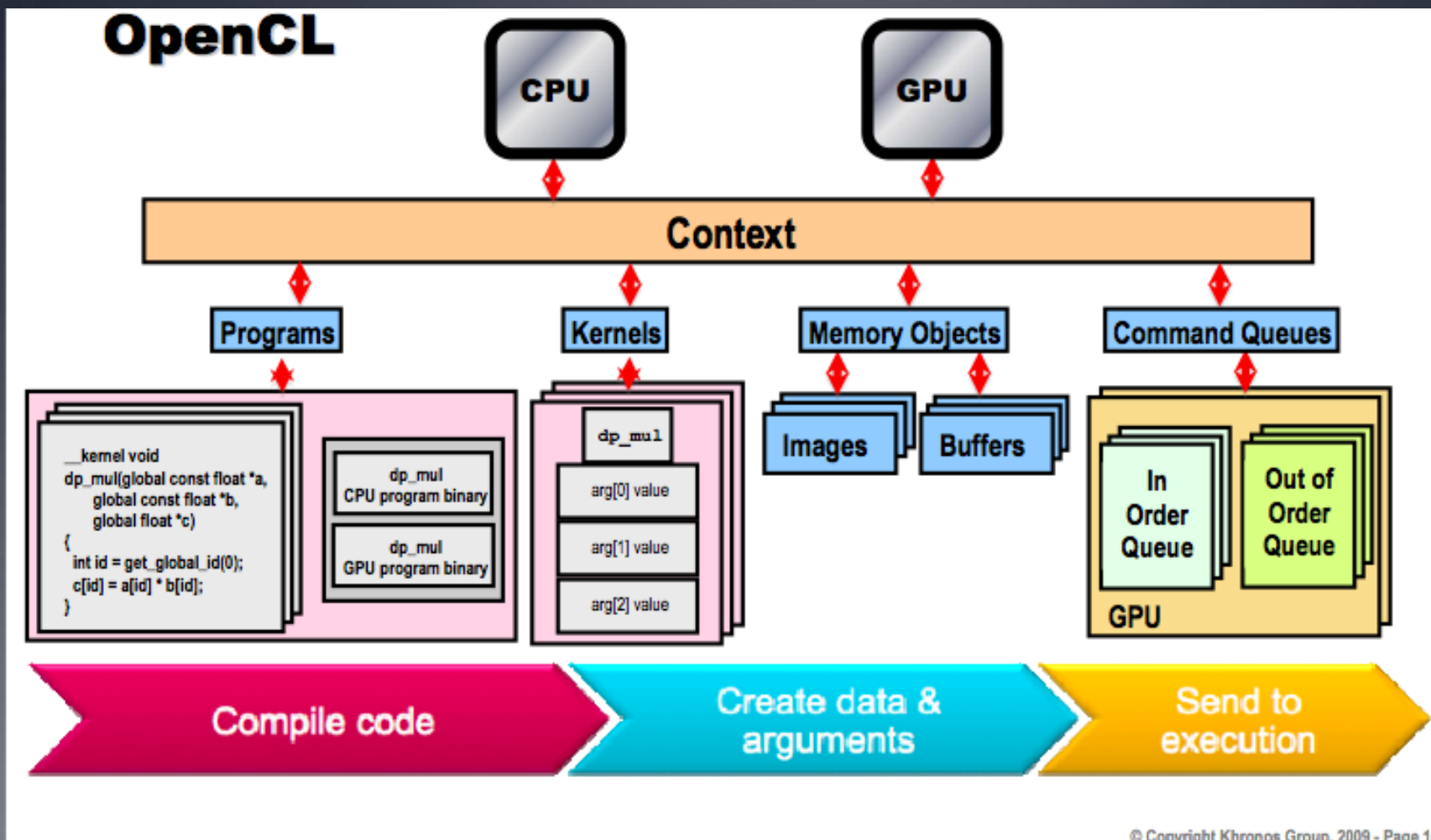
Error Checking

- OpenCL commands 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



Programming Model

- Data parallel
 - One-to-one mapping between work-items and elements in a memory object
 - Work-groups can be defined explicitly (like CUDA) or implicitly (specify the number of work-items and OpenCL creates the work-groups)
- Task parallel
 - Kernel is executed independent of an index space
 - Other ways to express parallelism: enqueueing multiple tasks, using device-specific vector types, etc.
- Synchronization
 - Possible between items in a work-group
 - Possible between commands in a context command queue

Running the Example Code

- A simple vector addition OpenCL program that goes along with this lecture was provided
- Before running, the following should appear in your `.bashrc` file:
 - `export LD_LIBRARY_PATH=<path to stream sdk>/lib/x86_64`
- To compile:
 - Make sure that `vecadd.c` and `vecadd.cl` are in the current working directory
 - `gcc -o vecadd vecadd.c -I<path to stream sdk>/include -L<path to stream sdk>/lib/x86_64 -lOpenCL`

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