OpenCL

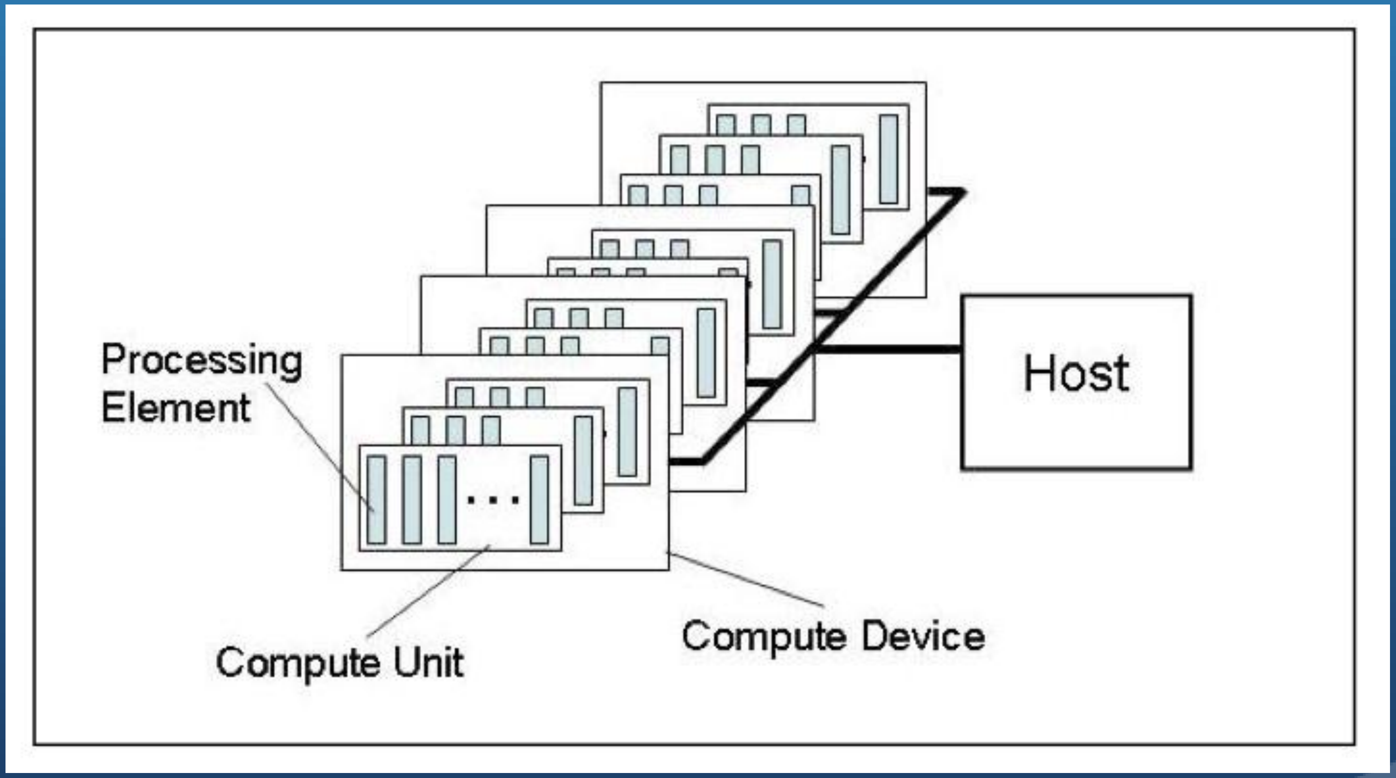
Matt Sellitto
Dana Schaa
Northeastern University
NUCAR

OpenCL Architecture

- Parallel computing for heterogenous devices
  - CPUs, GPUs, other processors (Cell, DSPs, etc)
  - 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)

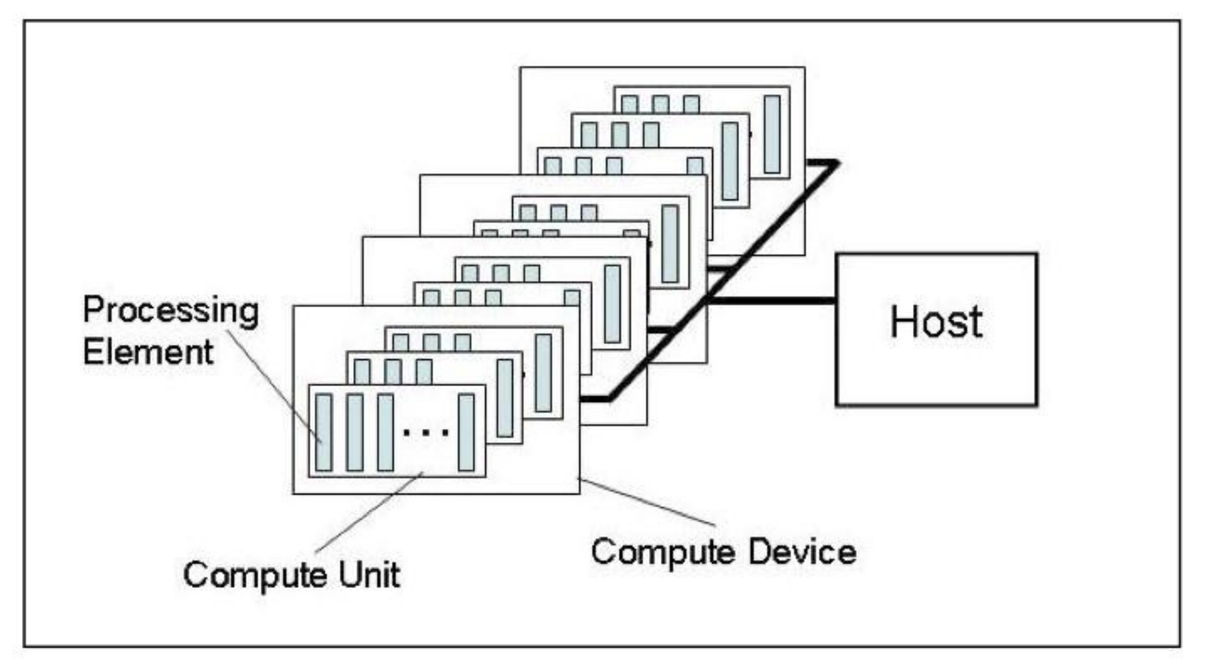
Host-Device Model (Platform Model)

- 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



Host-Device Model

- The host is whatever the OpenCL library runs on
  - Usually x86 CPUs
- Devices are processors that the library can talk to
  - CPUs, GPUs, and other accelerators
- For AMD
  - All CPUs are 1 device (each core is a compute unit and processing element)
  - Each GPU is a separate device



Platforms

- Platform == OpenCL implementation (AMD, NVIDIA, Intel)
- Uses an “Installable Client Driver” model
  - Generic OpenCL library runs and detects platforms
  - The goal is to allow multiple implementations that co-exist
  - However, current GPU driver model does not allow that

Discovering Platforms

```
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

* Step 1

Discovering Platforms

```
// TODO: Use clGetPlatformIDs() to retrieve the number of platforms present
status = clGetPlatformIDs(0, NULL, &numPlatforms);
if(status != CL_SUCCESS) {
    printf("clGetPlatformIDs failed\n");
    exit(-1);
}

// Make sure some platforms were found
if(numPlatforms == 0) {
   printf("No platforms detected.\n");
   exit(-1);
}

// Allocate enough space for each platform
platforms = (cl_platform_id*)malloc(numPlatforms*sizeof(cl_platform_id));
if(platforms == NULL) {
    perror("malloc");
    exit(-1);
}

// TODO: Fill in platforms with clGetPlatformIDs()
clGetPlatformIDs(numPlatforms, platforms, NULL);
if(status != CL_SUCCESS) {
    printf("clGetPlatformIDs failed\n");
    exit(-1);
}
```

Discovering 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

* Step 2

Discovering Devices

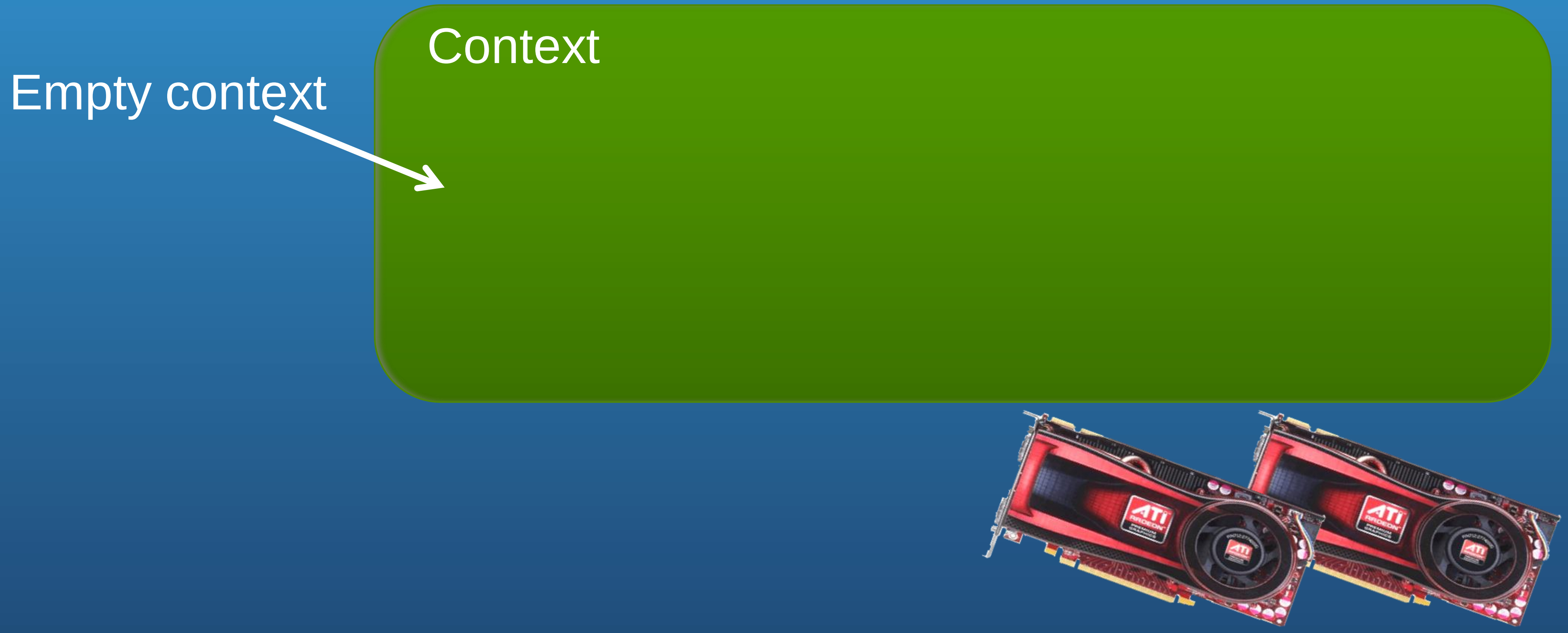
```
// TODO: Use clGetDeviceIDs() to retrive the number of devices present
status = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_GPU, 0, NULL, &numDevices);
if(status != CL_SUCCESS) {
    printf("clGetDeviceIDs failed\n");
    exit(-1);
}

// Make sure some devices were found
if(numDevices == 0) {
   printf("No devices detected.\n");
   exit(-1);
}

// Allocate enough space for each device
devices = (cl_device_id*)malloc(numDevices*sizeof(cl_device_id));
if(devices == NULL) {
    perror("malloc");
    exit(-1);
}

// TODO: Fill in devices with clGetDevicesIDs().
status = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_GPU, numDevices,
                        devices, NULL);
if(status != CL_SUCCESS) {
    printf("clGetDeviceIDs failed\n");
    exit(-1);
}
```

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: coordinators of execution of the kernels on the devices
    - Memory commands (data transfers)
    - 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



Creating a Context

```
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
- The function also provides a callback mechanism for reporting errors to the user

* Step 3

Creating a Context

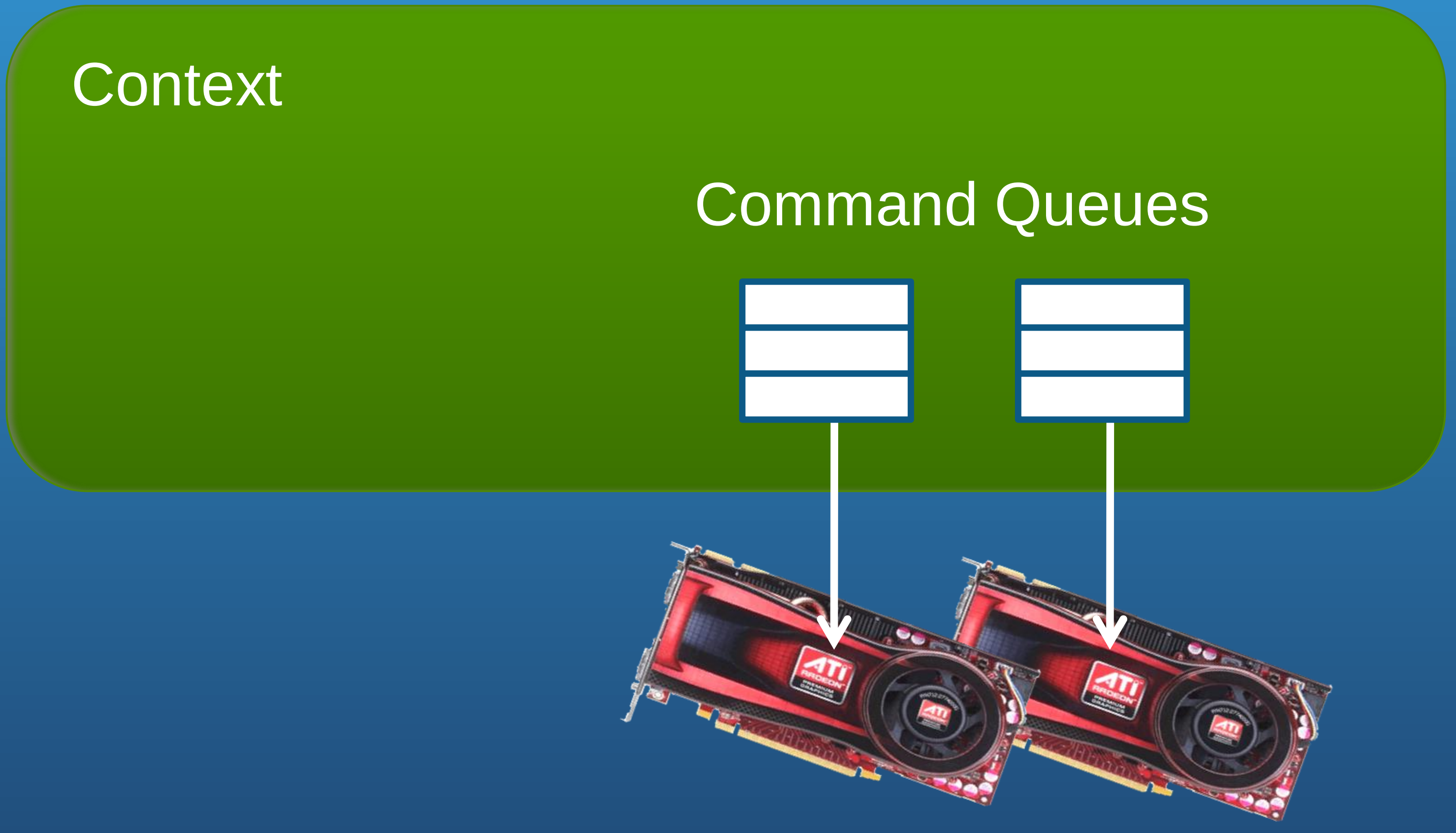
```
// TODO: Create a context using clCreateContext() and
// associate it with the devices
context = clCreateContext(props, numDevices, devices, NULL, NULL, &status);
if(status != CL_SUCCESS || context == NULL) {
    printf("clCreateContext failed\n");
    exit(-1);
}
```

Command Queues

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

Command Queues

- By supplying a command queue as an argument, the device being targeted can be determined



Creating a Command Queue

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

- 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)

* Step 4

Creating a Command Queue

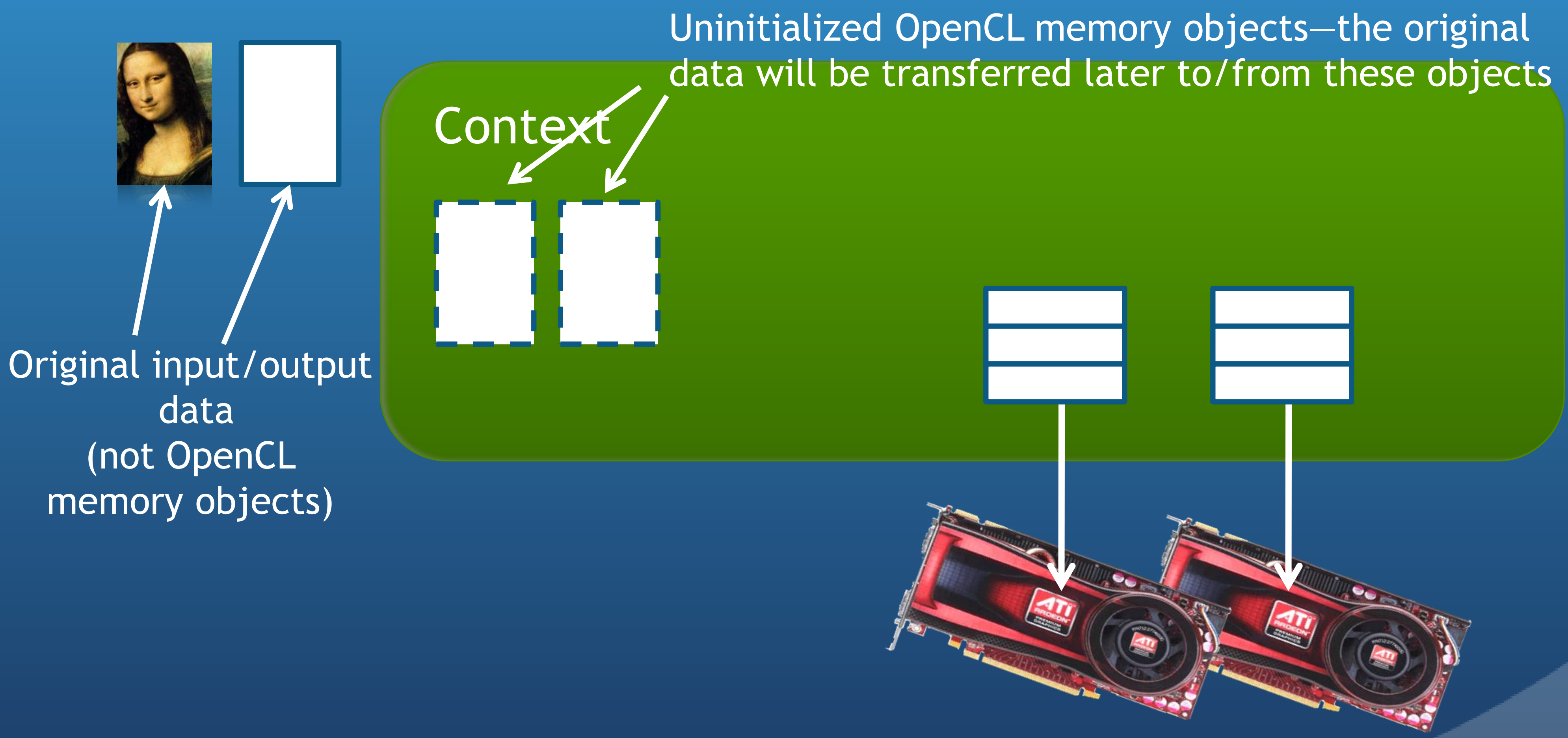
```
// TODO: Create a command queue using clCreateCommandQueue(),
// and associate it with the device you want to execute on
cmdQueue = clCreateCommandQueue(context, devices[0], 0, &status);
if(status != CL_SUCCESS || cmdQueue == NULL) {
    printf("clCreateCommandQueue failed\n");
    exit(-1);
}
```

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

Memory Objects

- Memory objects are associated with a context
  - They must be explicitly copied to a device prior to execution (covered next)



Creating a Buffer

```
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

Creating a Buffer

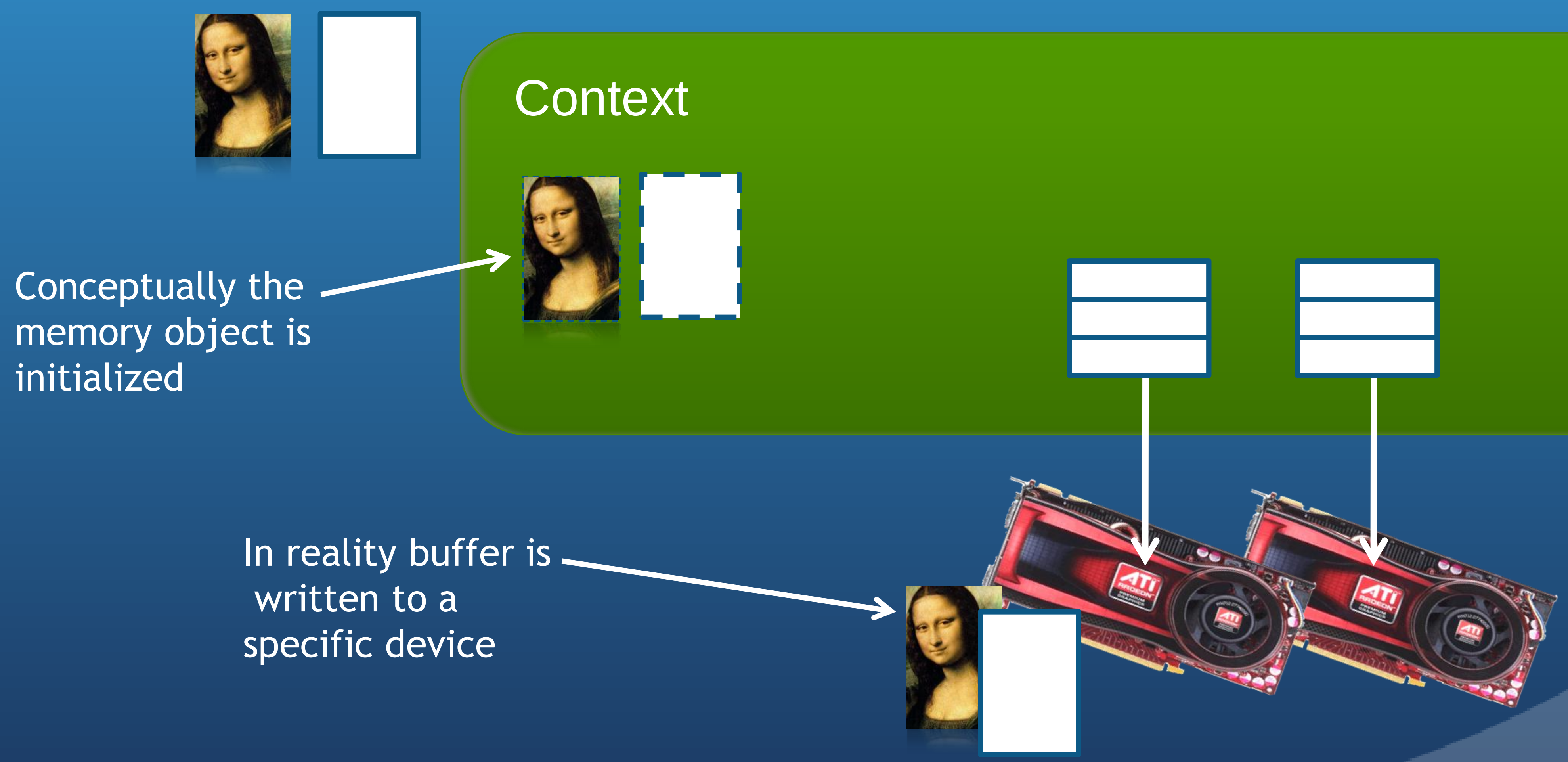
```
// TODO: use clCreateBuffer() to create a buffer object (d_A)
d_A = clCreateBuffer(context, CL_MEM_READ_ONLY, datasize, NULL, &status);
if(status != CL_SUCCESS || d_A == NULL) {
    printf("clCreateBuffer failed\n");
    exit(-1);
}

// TODO: use clCreateBuffer() to create a buffer object (d_B)
d_B = clCreateBuffer(context, CL_MEM_READ_ONLY, datasize, NULL, &status);
if(status != CL_SUCCESS || d_B == NULL) {
    printf("clCreateBuffer failed\n");
    exit(-1);
}

// TODO: use clCreateBuffer() to create a buffer object (d_C)
d_C = clCreateBuffer(context, CL_MEM_READ_WRITE, datasize, NULL, &status);
if(status != CL_SUCCESS || d_C == NULL) {
    printf("clCreateBuffer failed\n");
    exit(-1);
}
```

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)

Transferring Data

- Memory objects are transferred to devices by specifying an action (read or write) and a command queue



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

* Step 6

Transferring Data

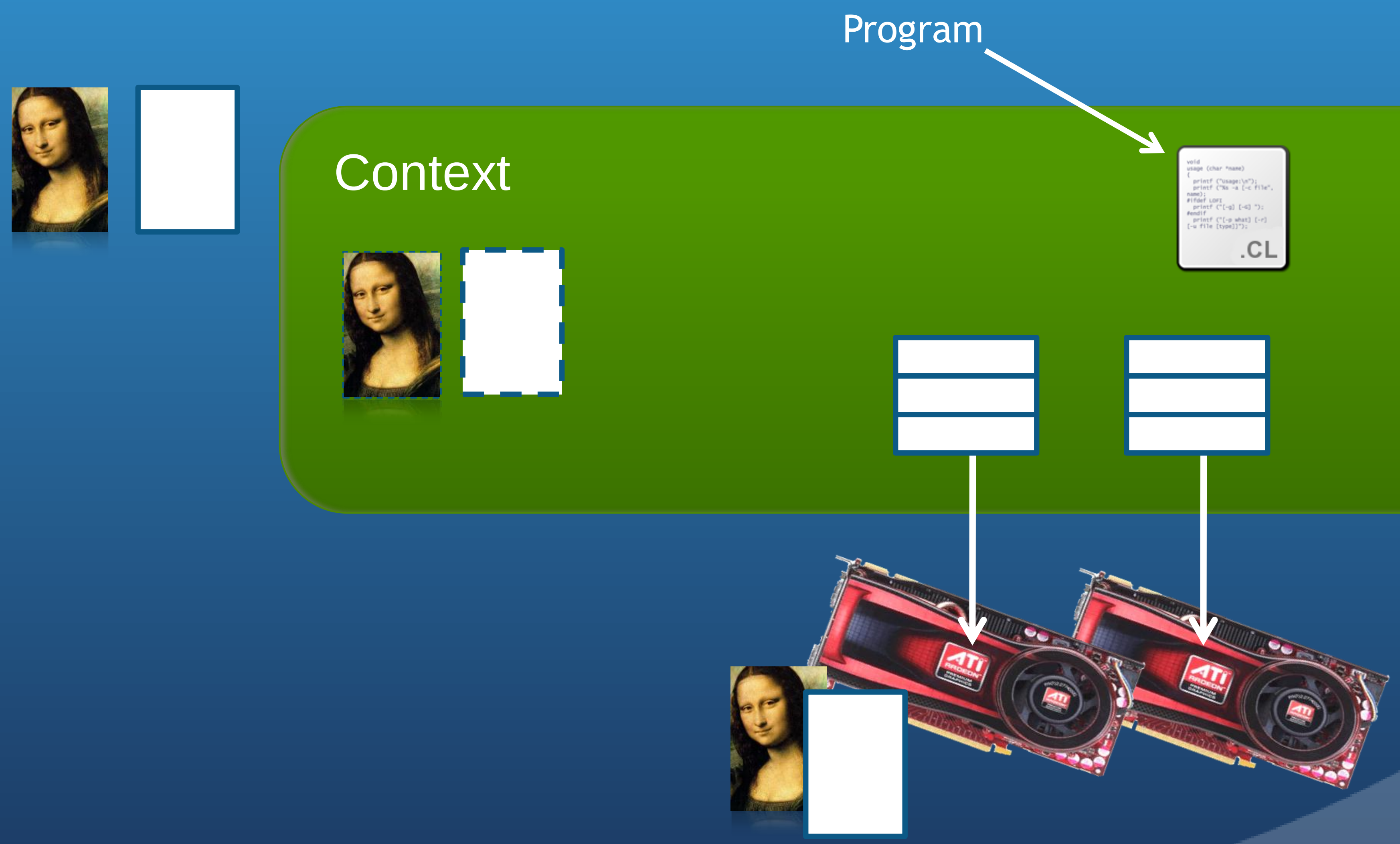
```
// TODO: use clEnqueueWriteBuffer to write d_A to the device
status = clEnqueueWriteBuffer(cmdQueue, d_A, CL_TRUE, 0, datasize, A,
                              0, NULL, NULL);
if(status != CL_SUCCESS) {
   printf("clEnqueueWriteBuffer failed\n");
   exit(-1);
}

// TODO: use clEnqueueWriteBuffer to write d_B to the device
status = clEnqueueWriteBuffer(cmdQueue, d_B, CL_TRUE, 0, datasize, B,
                              0, NULL, NULL);
if(status != CL_SUCCESS) {
   printf("clEnqueueWriteBuffer failed\n");
   exit(-1);
}
```

Programs and Kernels

- 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 a Program

```
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 a Program

```
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

Compiling a Program

- 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

* Step 7

Compiling a Program

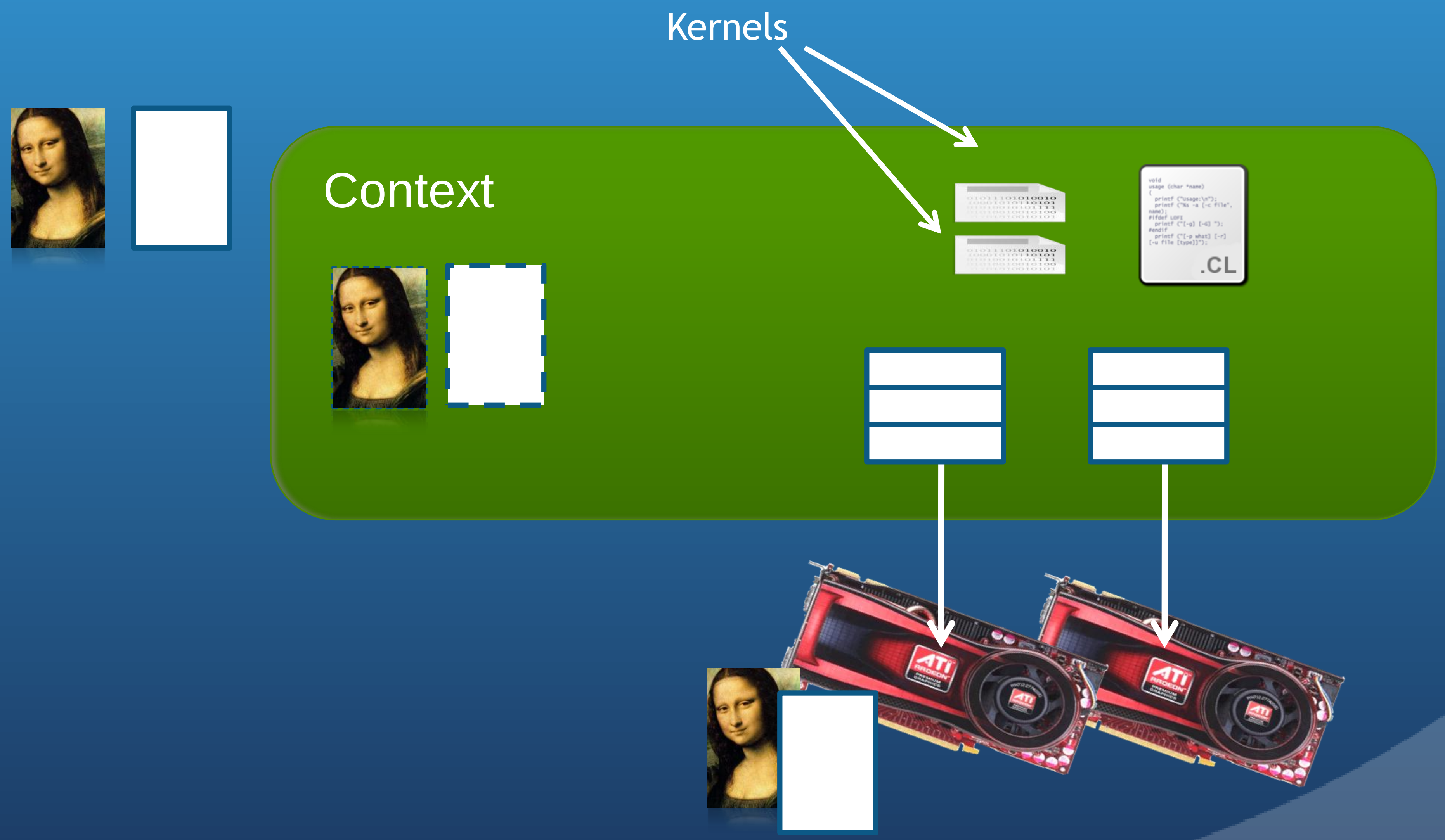
```
// TODO: Create a program using clCreateProgramWithSource()
// The 'source' string is the code from the vectoradd.cl (source) file.
program = clCreateProgramWithSource(context, 1, (const char**)&source,
                                    NULL, &status);
if(status != CL_SUCCESS) {
    printf("clCreateProgramWithSource failed\n");
    exit(-1);
}

cl_int buildErr;
// TODO: Build (compile & link) the program for the devices with
// clBuildProgram(). Save the return value in 'buildErr' (the following
// code will print any compilation errors to the screen)
buildErr = clBuildProgram(program, numDevices, devices, NULL, NULL, NULL);
```

Creating a Kernel

- 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

Creating a Kernel

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



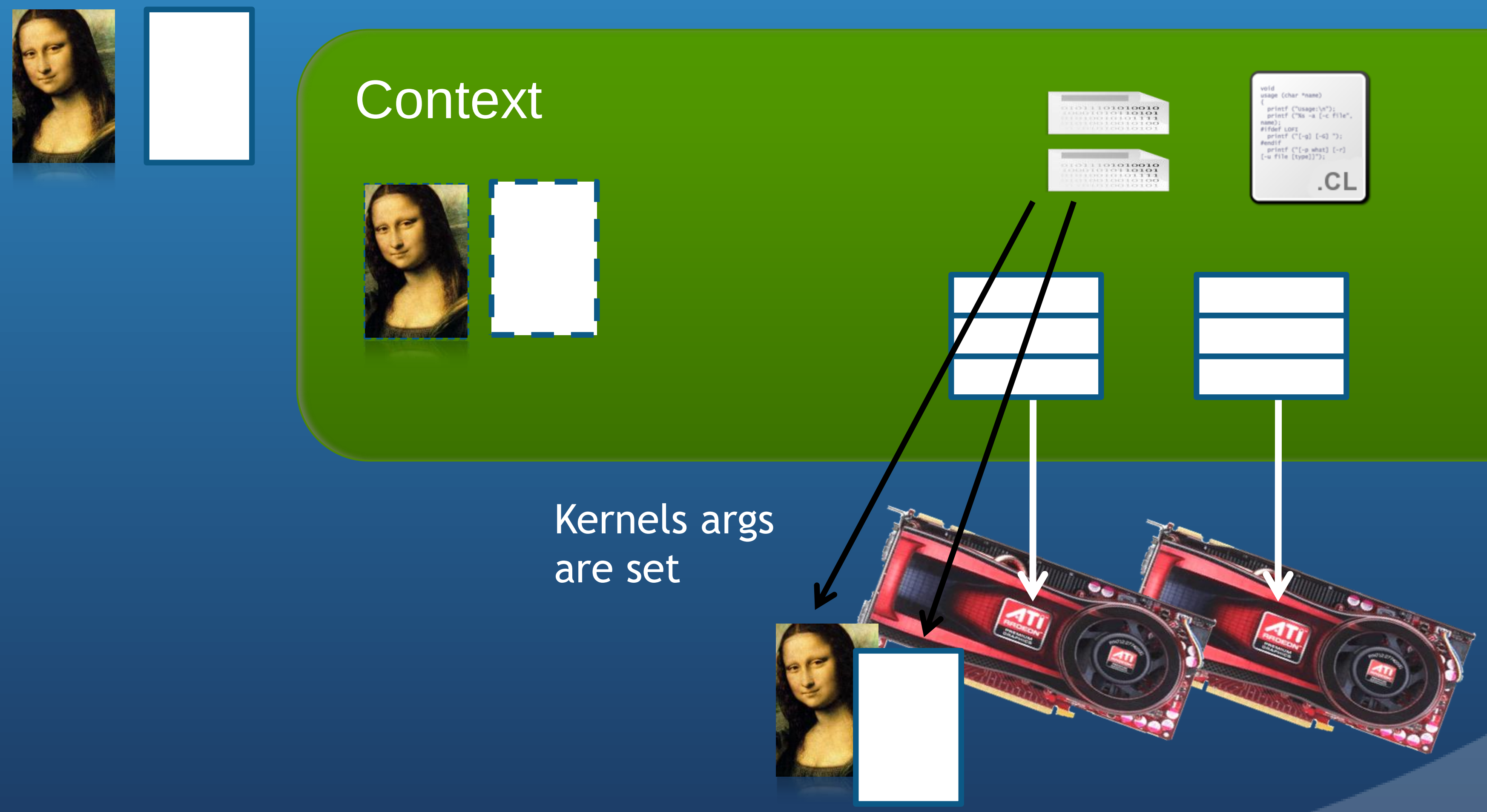
Setting Kernel Arguments

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

- Kernel arguments are set by repeated calls to clSetKernelArgs()
- 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);
- CUDA avoids this by using a preprocessor

Setting Kernel Arguments

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



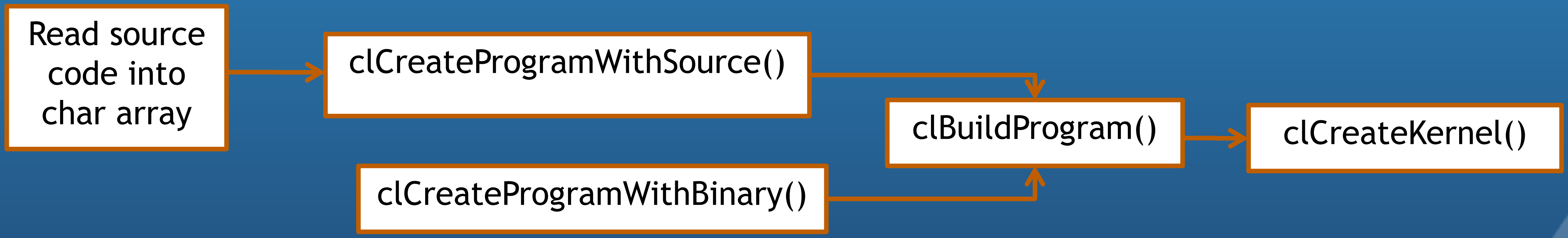
* Step 8

Creating a Kernel

```
// TODO: use clCreateKernel to create a kernel from the vector
// addition function (named "vecadd")
kernel = clCreateKernel(program, "vecadd", &status);
if(status != CL_SUCCESS) {
    printf("clCreateKernel failed\n");
    exit(-1);
}

// TODO: associate the input and output buffers with the kernel
// using clSetKernelArg()
status  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_A);
status |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_B);
status |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_C);
if(status != CL_SUCCESS) {
    printf("clSetKernelArg failed\n");
    exit(-1);
}
```

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

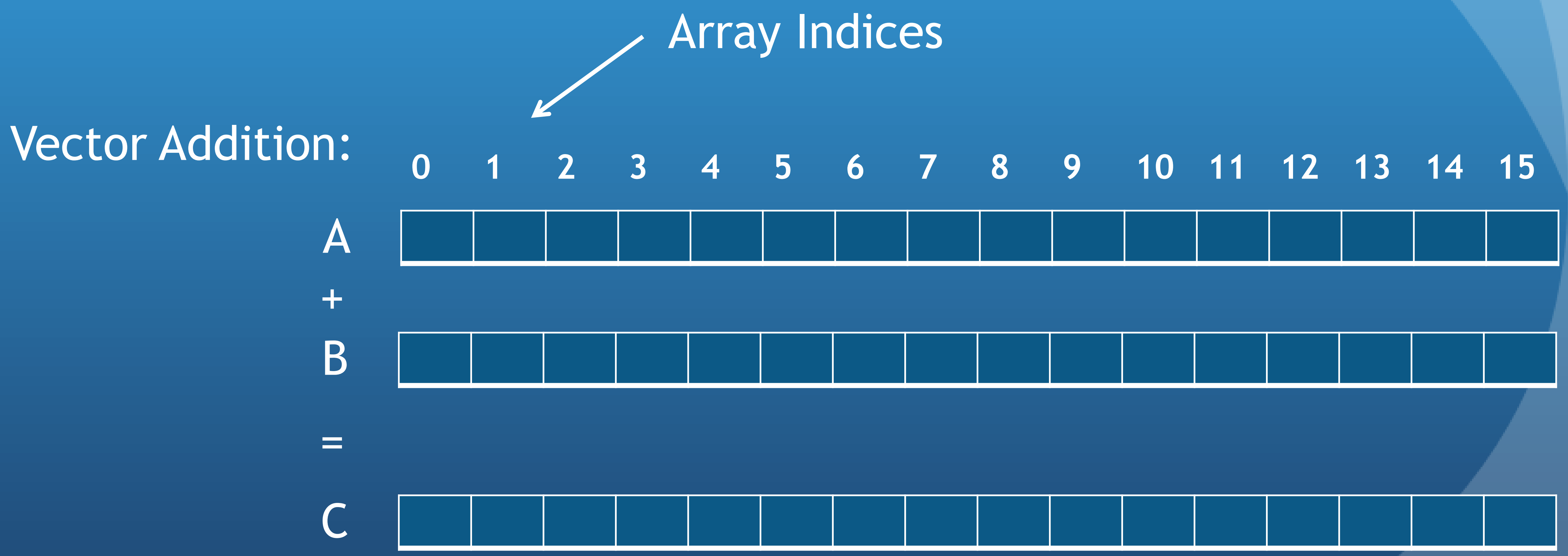


Kernel Threading Model

- 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

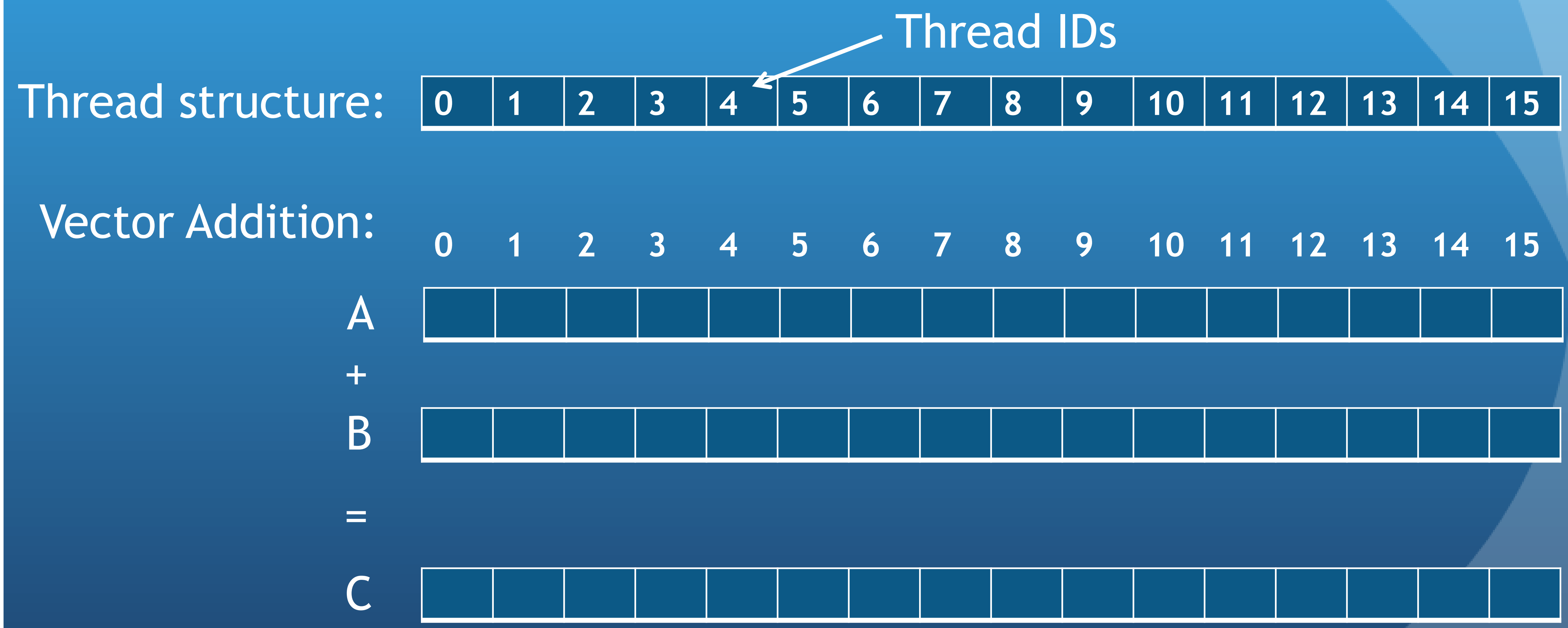
Thread Structure

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



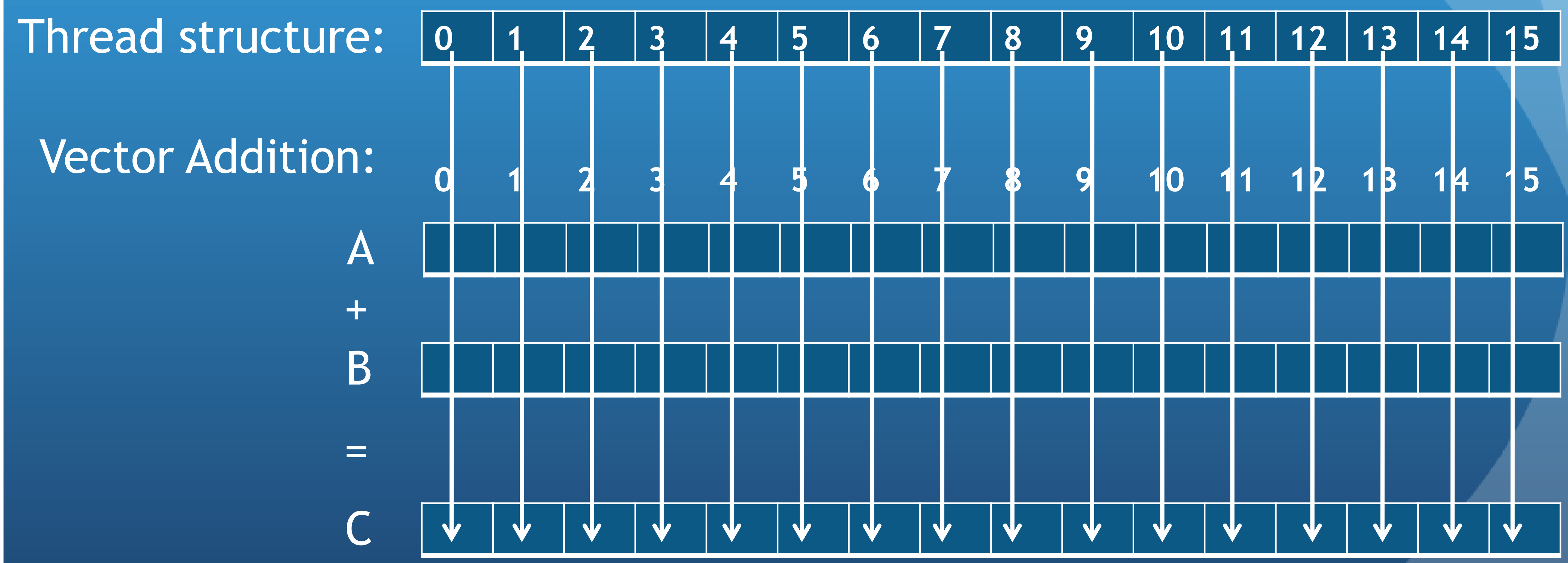
Thread Structure

- 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

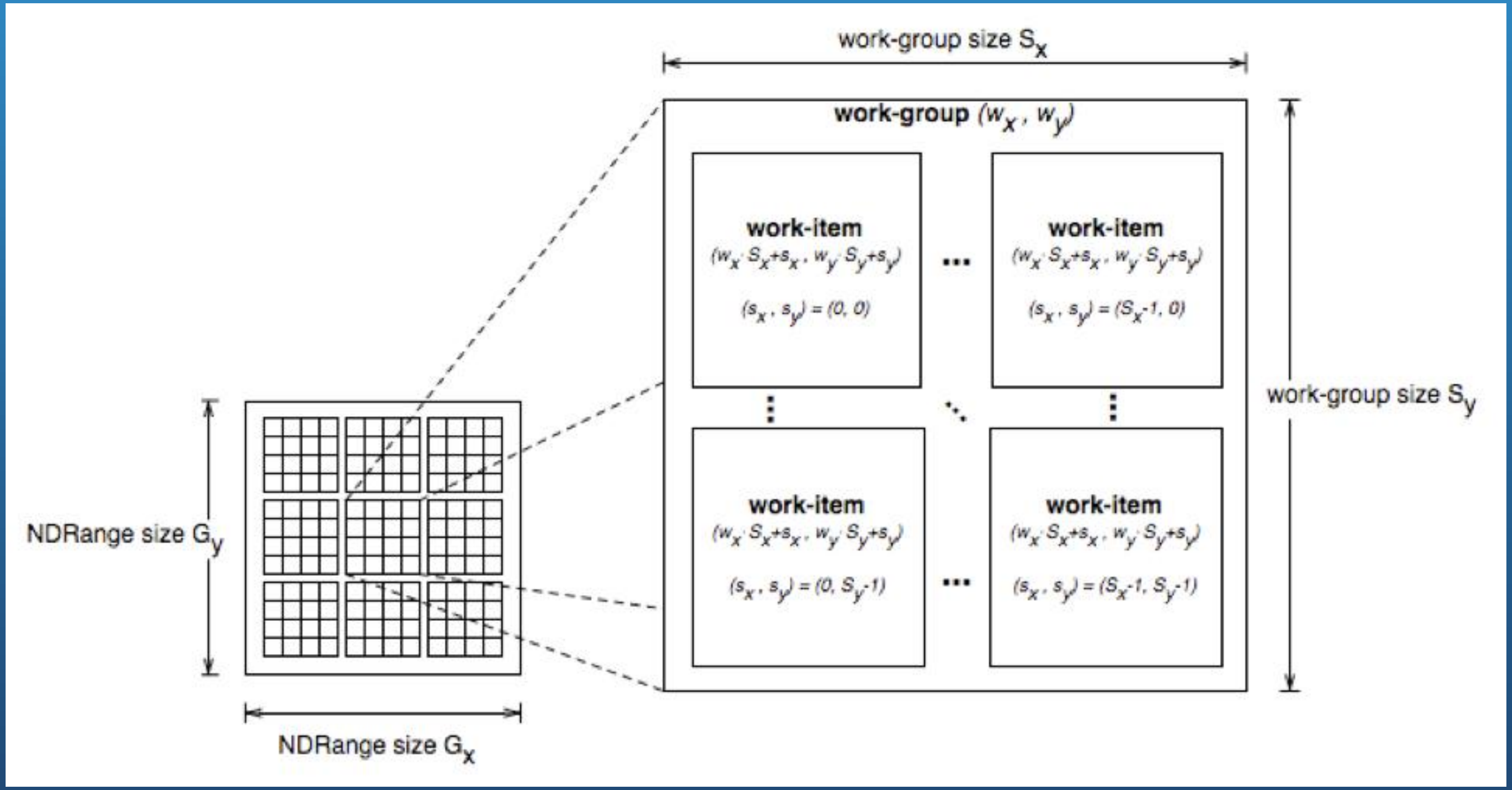


Thread Structure

- 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

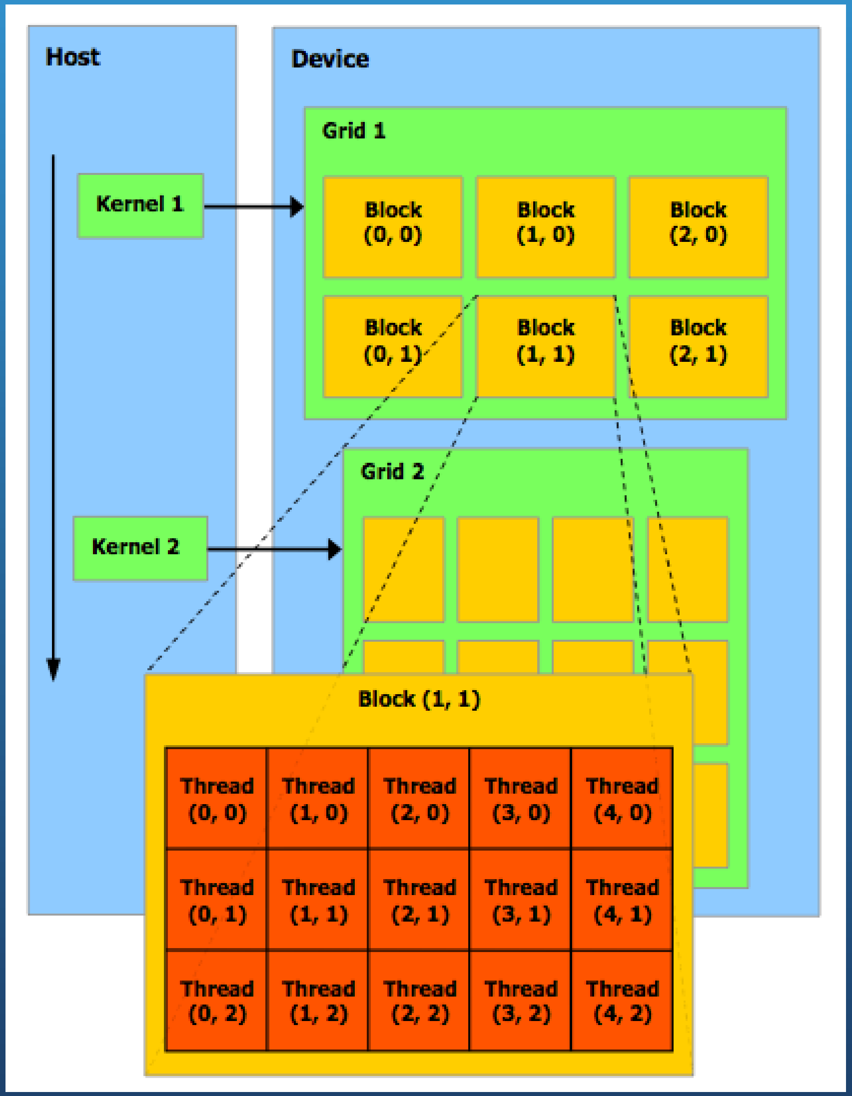
Thread Structure

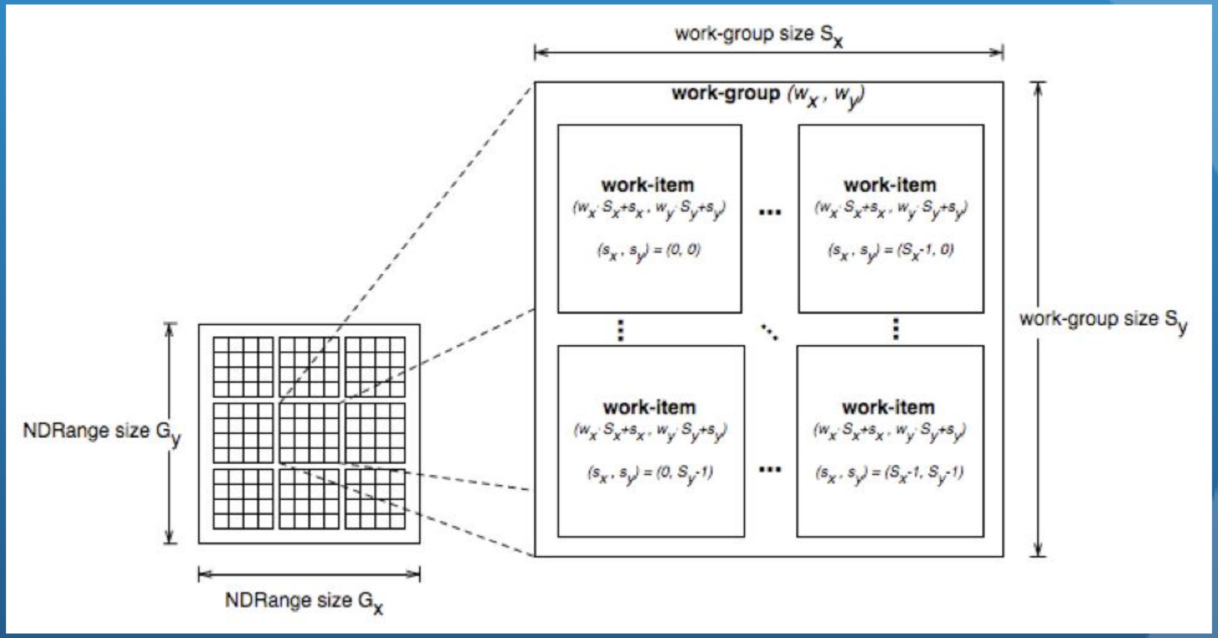
- Work-items can uniquely identify themselves based on:
  - A global id (unique within the index space)
  - A work-group ID and a local ID within the work-group



CUDA Comparison

C for CUDA	OpenCL
Thread	Work Item
Block	Work Group
Grid	Index space/NDRange





Thread Structure

- API calls allow threads to identify themselves
- 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)

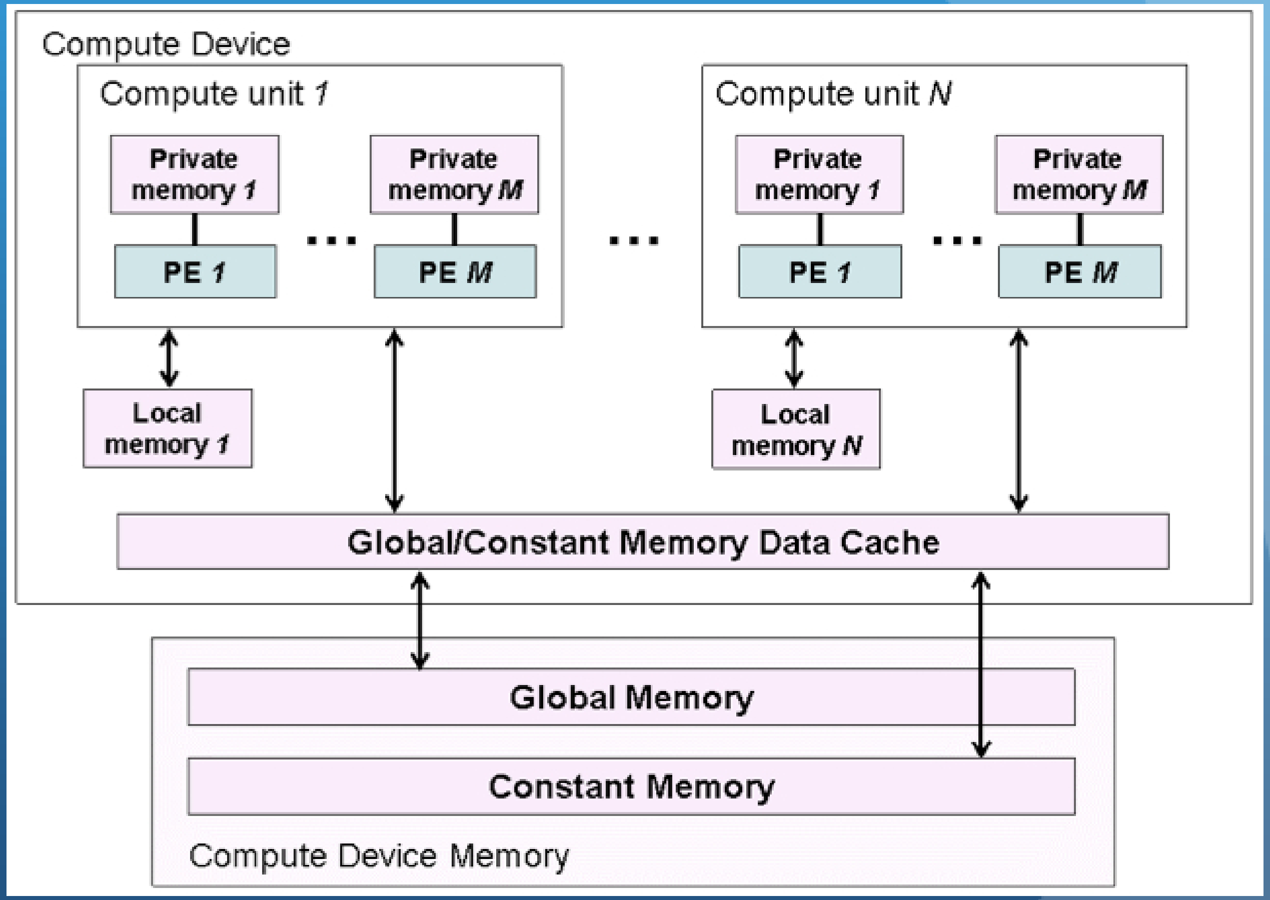
CUDA Comparison

C for CUDA terminology	OpenCL terminology
gridDim	**get_num_groups()**
blockDim	**get_local_size()**
blockIdx	**get_group_id()**
threadIdx	**get_local_id**
No direct equivalent. Combine **blockDim**, **blockIdx**, and **threadIdx** to calculate a global index.	**get_global_id()**
No direct equivalent. Combine **gridDim** and **blockDim** to calculate the global size.	**get_global_size()**

Memory Model

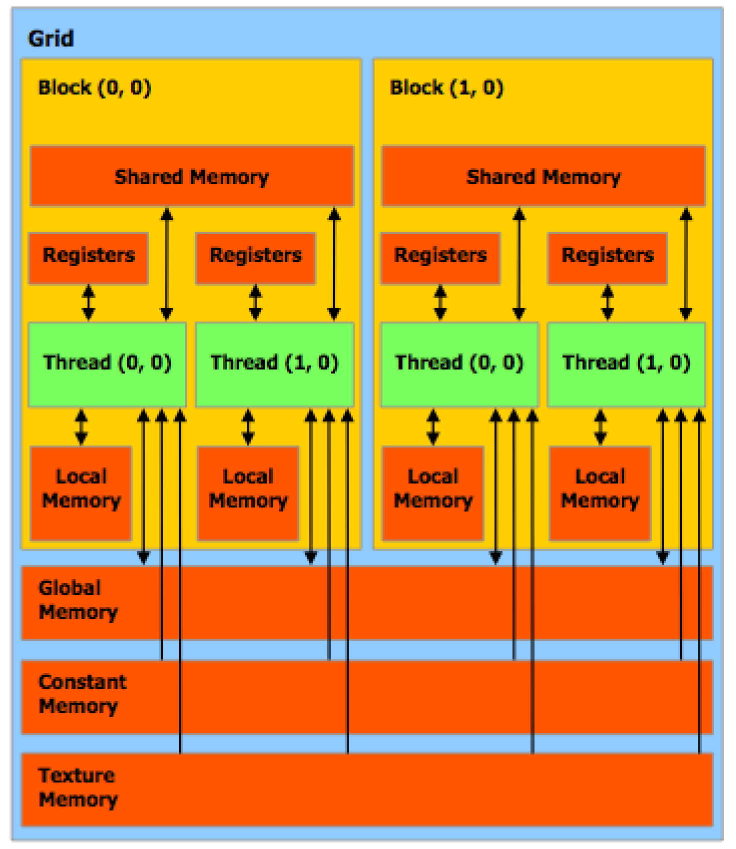
- The OpenCL memory model defines the various types of memories (closely related to GPU memory hierarchy)

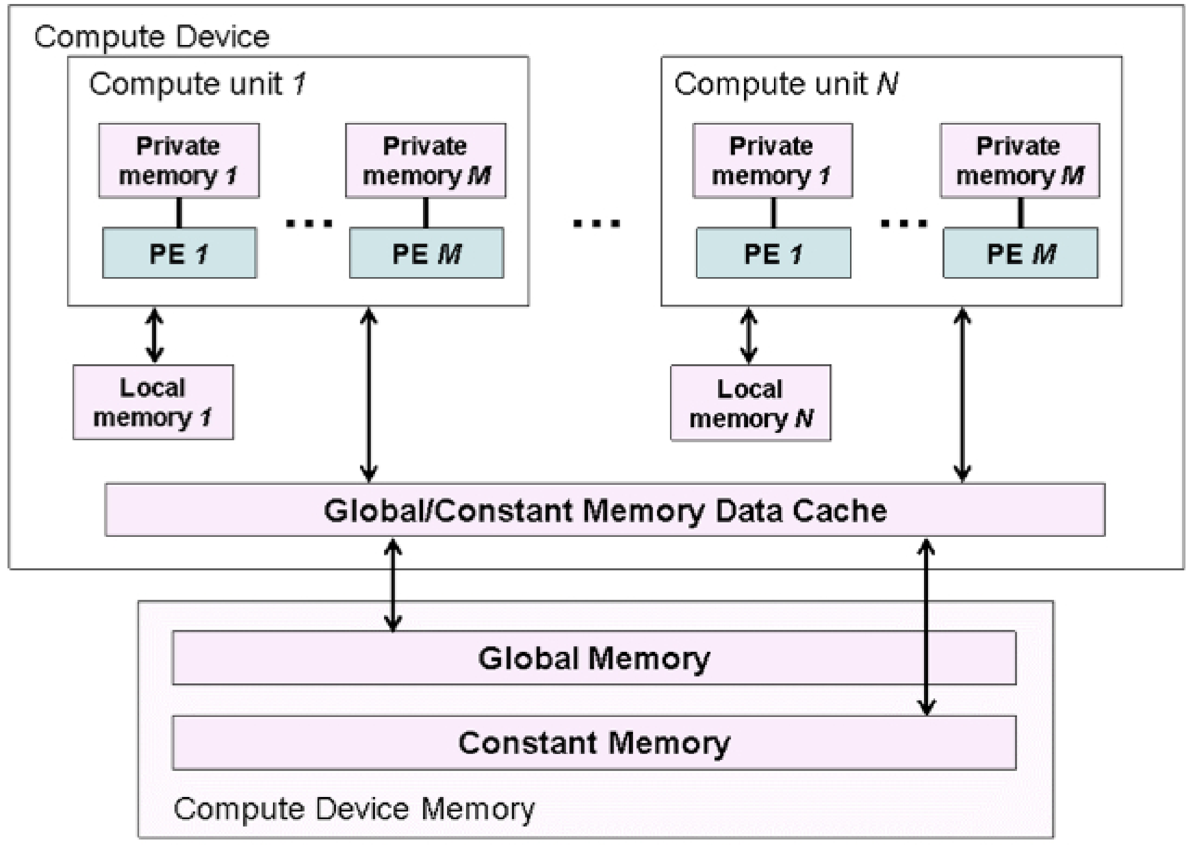
Memory	Description
Global	Accessible by all work-items
Constant	Read-only, global
Local	Local to a work-group
Private	Private to a work-item



CUDA Comparison

C for CUDA terminology	OpenCL terminology
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

CUDA Comparison

C for CUDA terminology	OpenCL terminology
__global__ function (callable from host, not callable from device)	**__kernel** function (callable fromdevice, includingCPU device)
__device__ function (not callable from host)	No annotation necessary
__constant__ variable declaration	**__constant** variable declaration
__device__ variable declaration	**__global** variable declaration
__shared__ variable declaration	**__local** variable declaration

Example Kernel

- Simple kernel to copy data from input to output buffer
  - Input and output data live in global memory
  - get_global_id(0) returns the thread ID in the X direction
    - Since the data is treated as an array, the thread structure will only be in one dimension

```
__kernel void
Copy1(__global const float * input, __global float * output)
{
  uint gid = get_global_id(0);
  output[gid] = input[gid];
  return;
}
```

* Write kernel

Writing a Kernel

```
__kernel
void vecadd(__global int *A,
            __global int *B,
            __global int *C) {


   // TODO: Fill in the kernel so that a work-item
   // adds the corresponding locations of 'A' and 'B', and
   // stores the result in 'C'.
   int idx = get_global_id(0);

   C[idx] = A[idx] + B[idx];

}
```

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

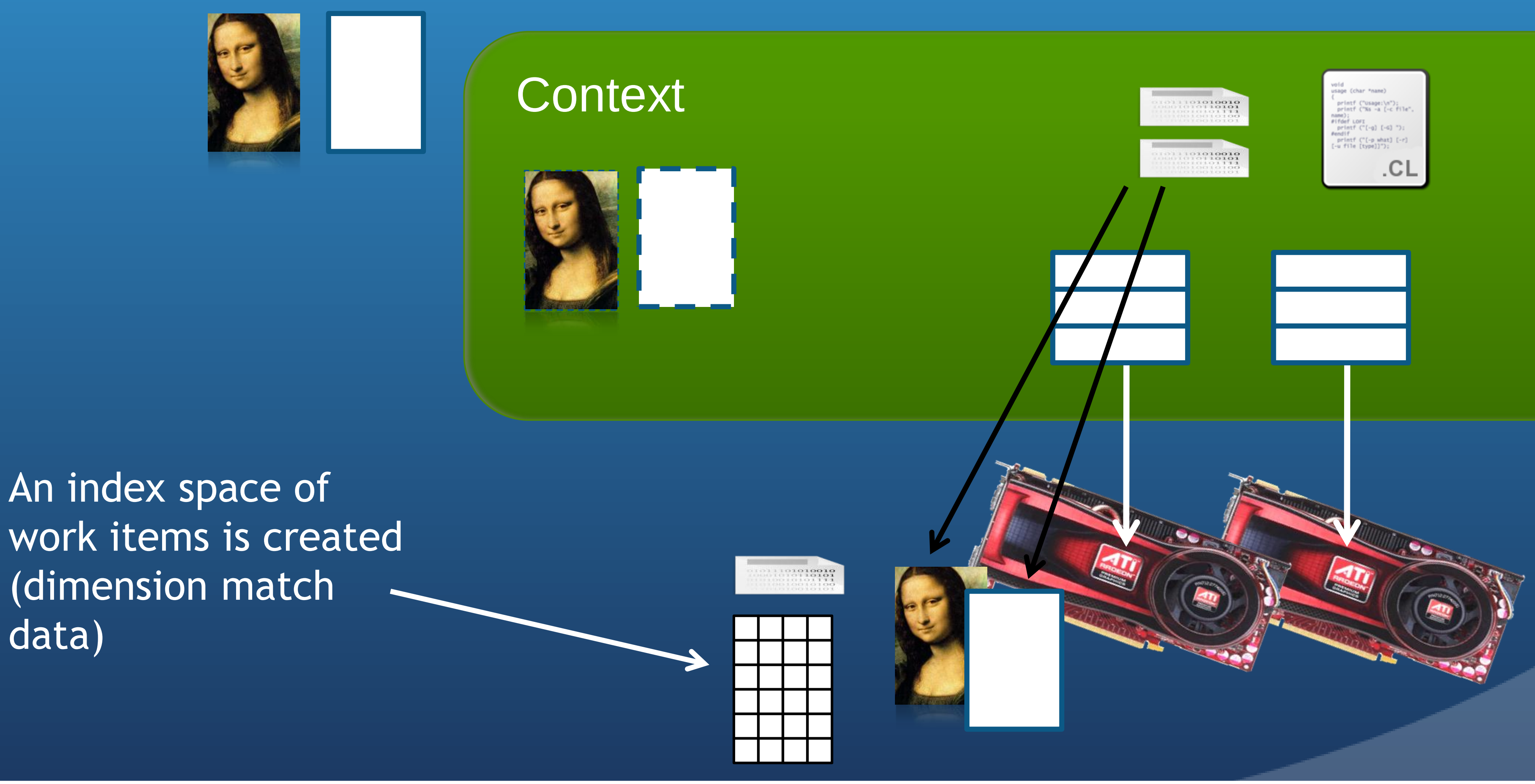
Executing the Kernel

```
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

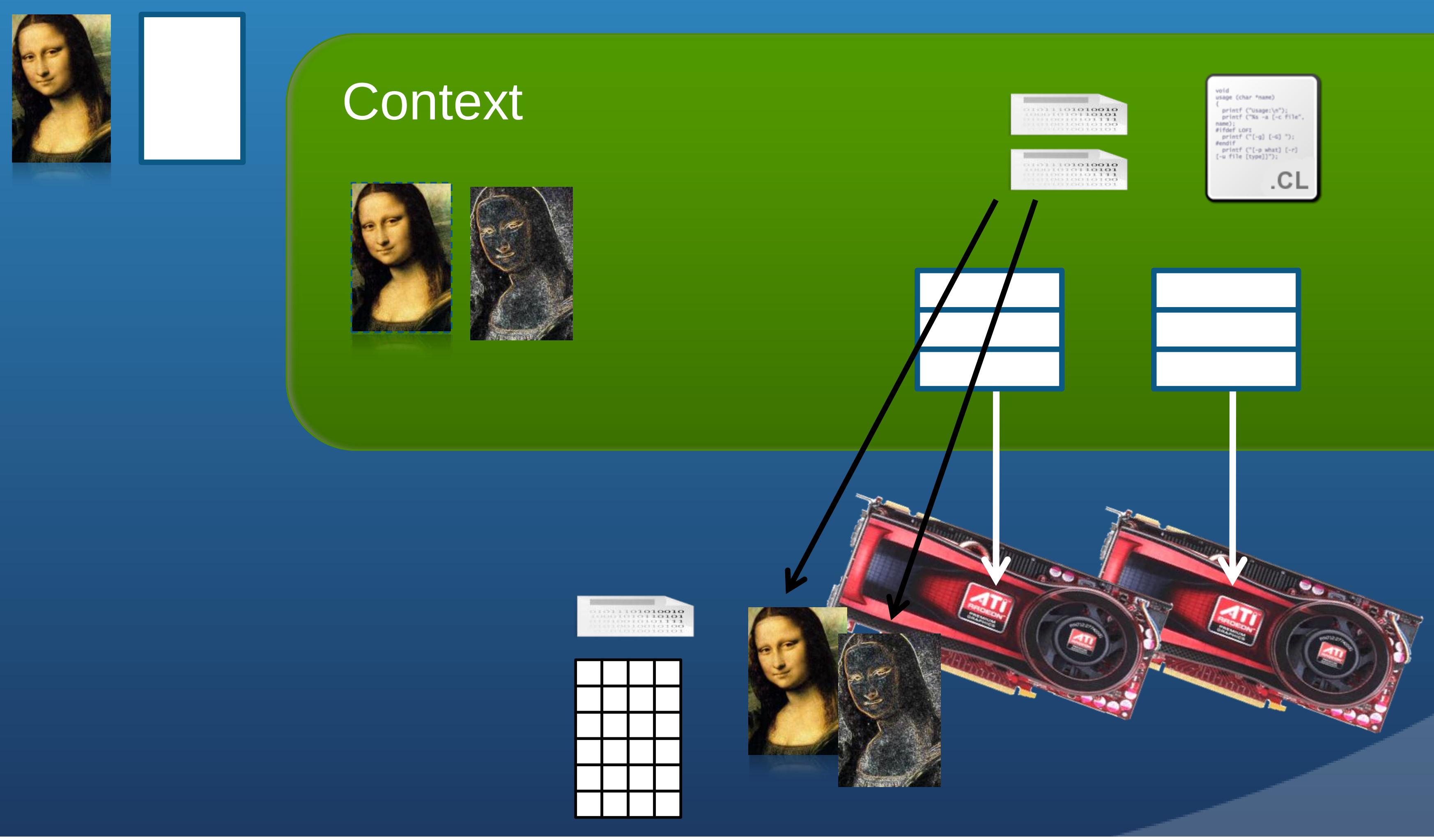
Executing the Kernel

- 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

* Step 9

Executing the Kernel

```
// TODO: Execute the kernel by using clEnqueueNDRangeKernel().
// 'globalWorkSize' is the 1D dimension of the work-items
status = clEnqueueNDRangeKernel(cmdQueue, kernel, 1, NULL, globalWorkSize,
                                NULL, 0, NULL, NULL);
if(status != CL_SUCCESS) {
    printf("clEnqueueNDRangeKernel failed\n");
    exit(-1);
}
```

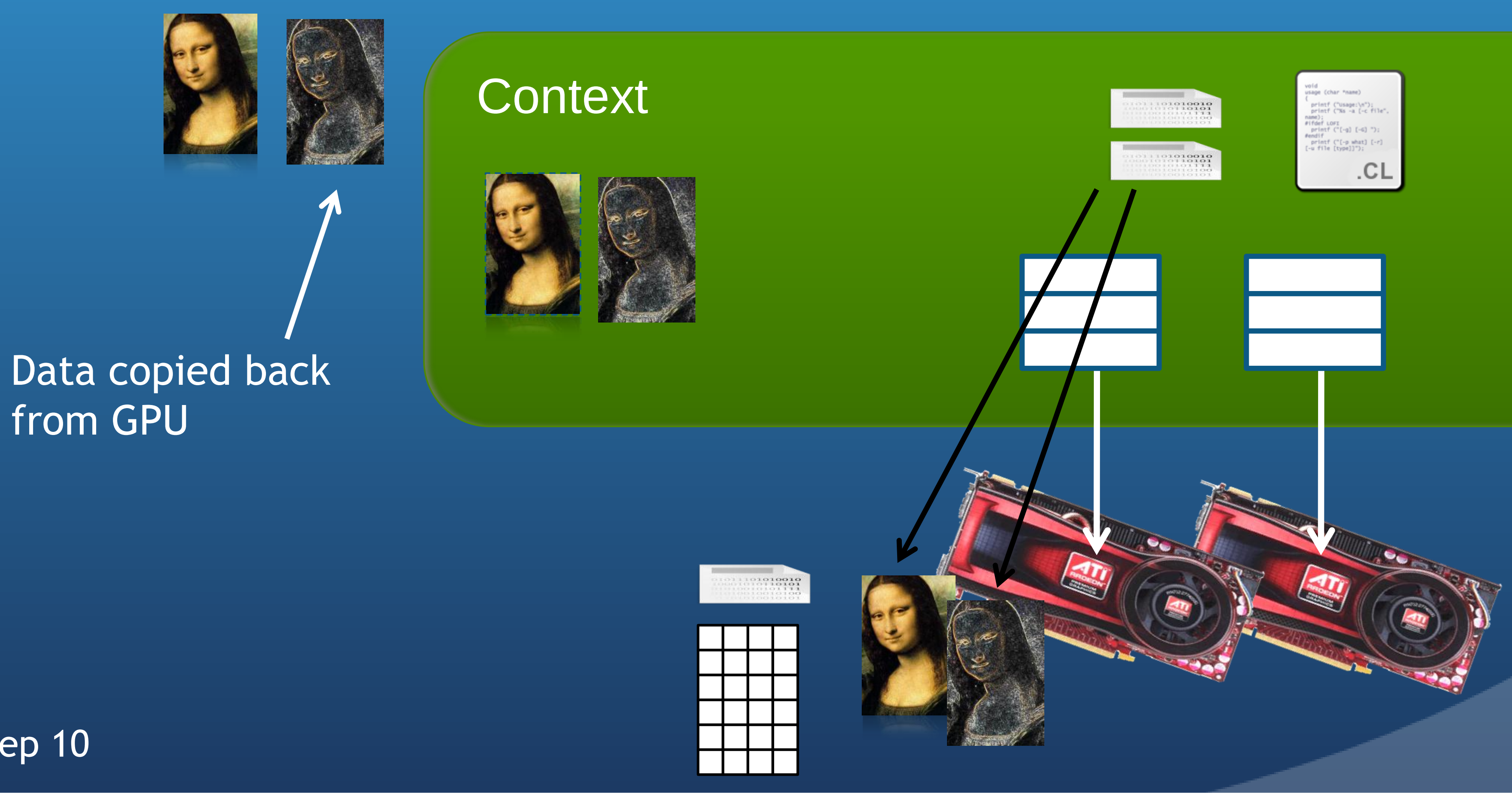
Copying Data Back

- 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

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

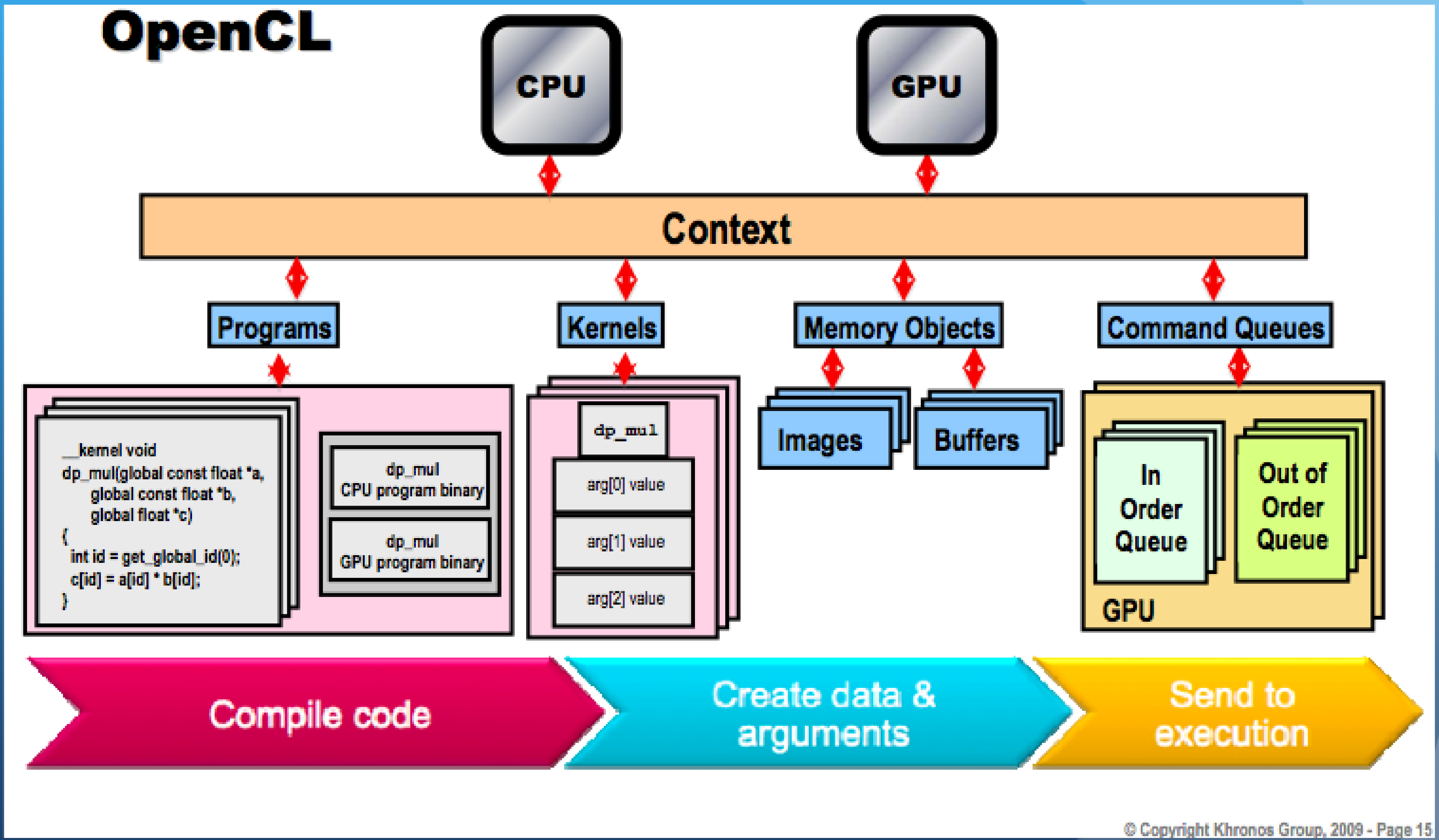


* Step 10

Copying Data Back

```
// TODO: Use clEnqueueReadBuffer() read the OpenCL output buffer (d_C)
// to the host output array (C)
clEnqueueReadBuffer(cmdQueue, d_C, CL_TRUE, 0, datasize, C,
                    0, NULL, NULL);
```

Big Picture



OpenCL
CPU
GPU
Context
Programs
Kernels
Memory Objects
Command Queues
__kernel void
dp_mul(global const float *a,
global const float *b,
global float *c)
{
int id = get_global_id(0);
c[id] = a[id] * b[id];
}
dp_mul
CPU program binary
dp_mul
GPU program binary
dp_mul
arg[0] value
arg[1] value
arg[2] value
Images
Buffers
In Order Queue
Out of Order Queue
GPU
Compile code
Create data & arguments
Send to execution
© Copyright Khronos Group, 2009 - Page 15

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 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

OpenCL vs. CUDA (runtime)

OpenCL	CUDA
clGetPlatformIDs clGetDeviceIDs clCreateContext clCreateCommandQueue	<not needed>
clCreateBuffer	cudaMalloc
clEnqueueWriteBuffer	cudaMemcpy
clCreateProgramWithSource clBuildProgram clCreateKernel clSetKernelArg clEnqueueNDRangeKernel	kernel <<< dims >>> args
clEnqueueReadBuffer	cudaMemcpy
clRelease*	cudaFree