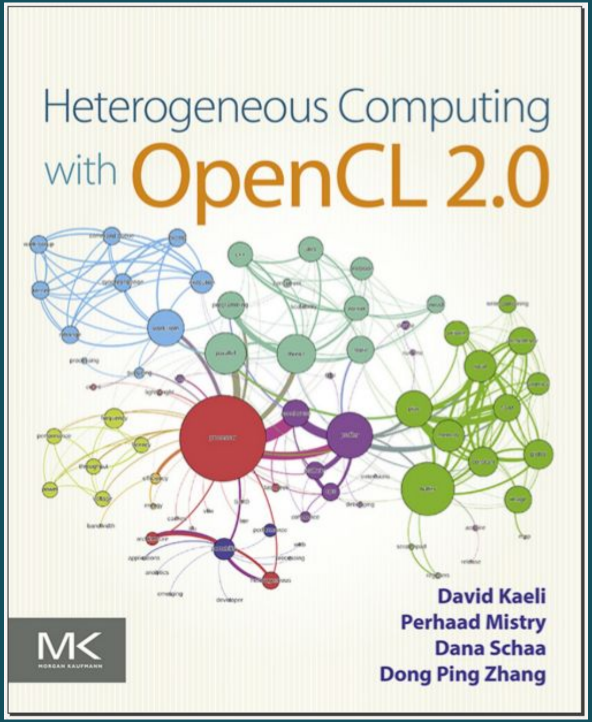
Introduction to OpenCL

AdC - 2025

Bibliografia



Heterogeneous Computing with OpenCL 2.0

Editorial:	Morgan Kaufmann
Autores:	David Kaeli
	Perhaad Mistry
	Dana Schaa
	Dong Ping Zhang

Heterogeneous Systems

No single architecture is best for running all classes of workloads, and most applications possess a mix of the workload characteristics.

Control-intensive applications tend to run faster on superscalar CPUs, where significant hardware has been devoted to branch prediction mechanisms.

Data-intensive applications tend to run fast on vector architectures, where the same operation is applied to multiple data items concurrently.

Heterogeneous Computing with OpenCL

About OpenCL

- OpenCL allows parallel computing on heterogeneous devices
  - CPUs (x86, ARM, and PowerPC both in embebbed and HPC systems), GPUs (AMD, NVIDIA), other processors (FPGAs, DSPs, etc)
  - Support efficient design of complicated concurrent programs (both task-based and data-based parallelism) with low overhead.

- *"Most important, OpenCL's cross-platform, industrywide support makes it an excellent programming model for developers to learn and use, with the confidence that it will continue to be widely available for years to come with ever-increasing scope and applicability."*

OpenCL Architecture

- **<u>Host</u>** code typically written in C (OpenCL Specification is based on this language). However, official wrappers exists for C++, and private porting to Java, Python, .NET and others are also available.
- **<u>Device</u>** codes (kernels) are written in OpenCL C language.

OpenCL Model

OpenCL model is defined in four parts

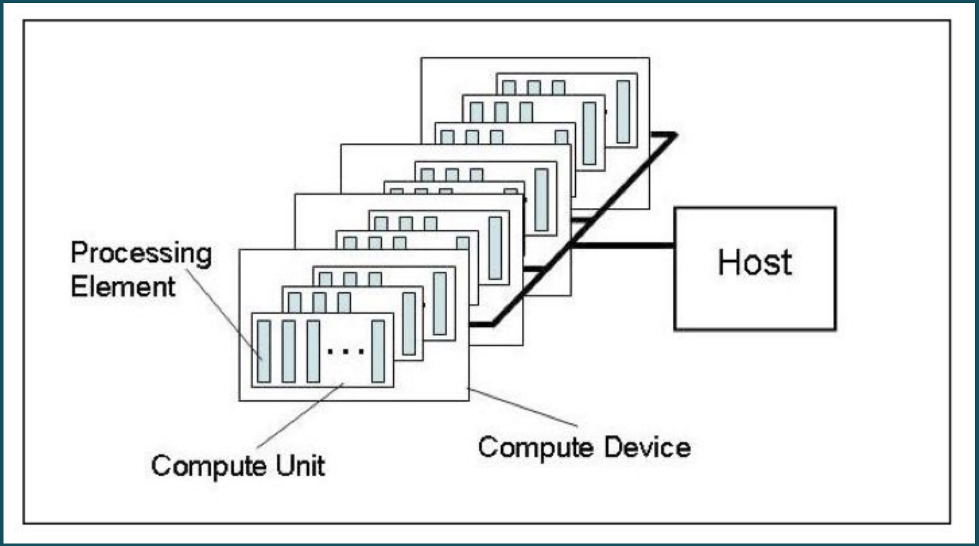
- Platform Model: Specifies that there is one processor coordinating execution (the host) and one or more processors capable of executing OpenCL C code (the devices). It defines an abstract hardware model that is used by programmers when writing OpenCL C functions (called kernels) that execute on the devices.
- Execution Model: Defines how the OpenCL environment is configured on the host and how kernels are executed on the device. This includes setting up an OpenCL context on the host, providing mechanisms for host–device interaction, and defining a concurrency model used for kernel execution on devices.

OpenCL Model

- Memory Model: Defines the abstract memory hierarchy that kernels use, regardless of the actual underlying memory architecture. The memory model closely resembles current GPU memory hierarchies, although this has not limited adoptability by other accelerators.
- Programming Model: Defines how the workload is divided in the

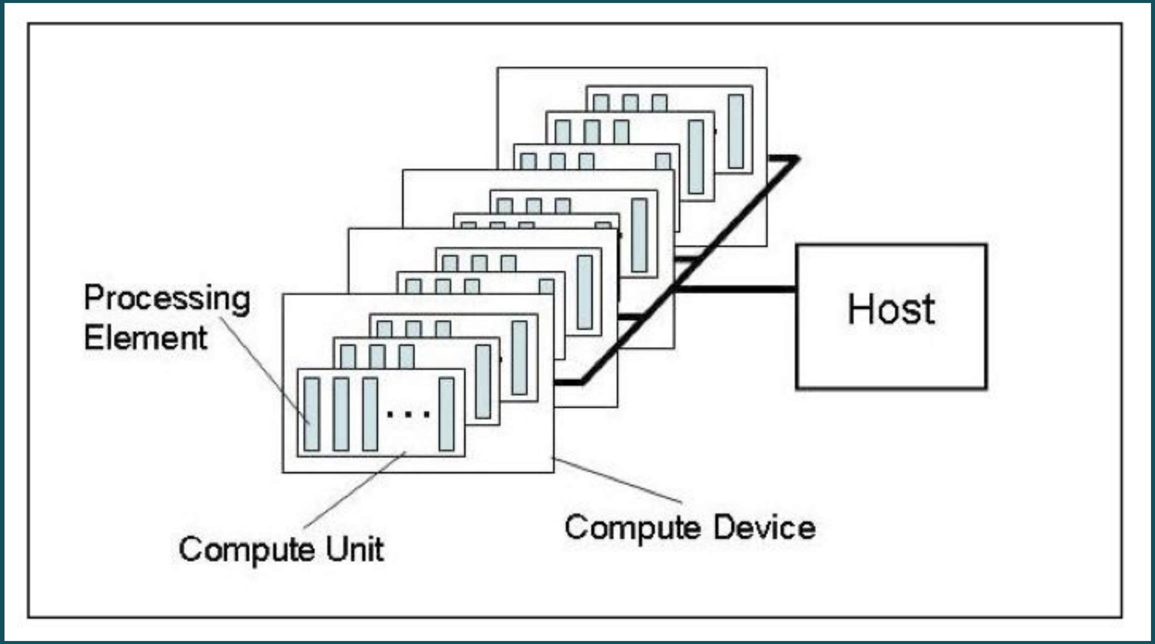
Platform Model

- 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

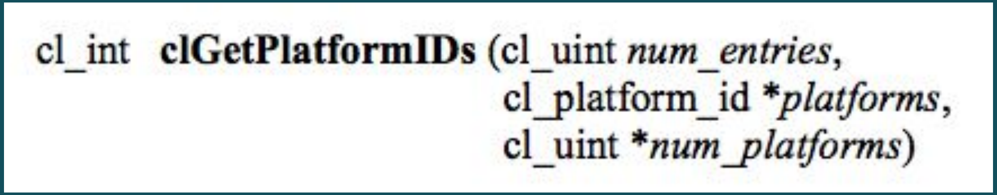


OpenCL WorkFlow

- We are going to create a vector addition program (A[i] + B[i] = C[i] forall i from 0 to 2048).

- Headers:
  ```
  // OpenCL includes
  #include <CL/cl.h>
  ```

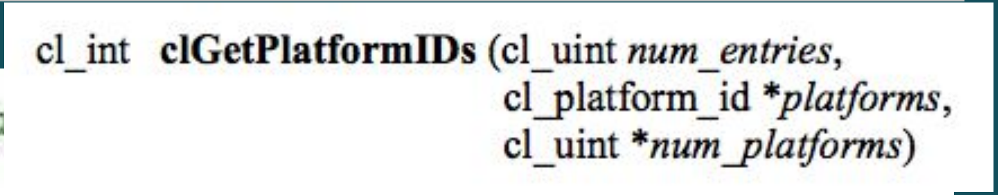
Selecting a Platform



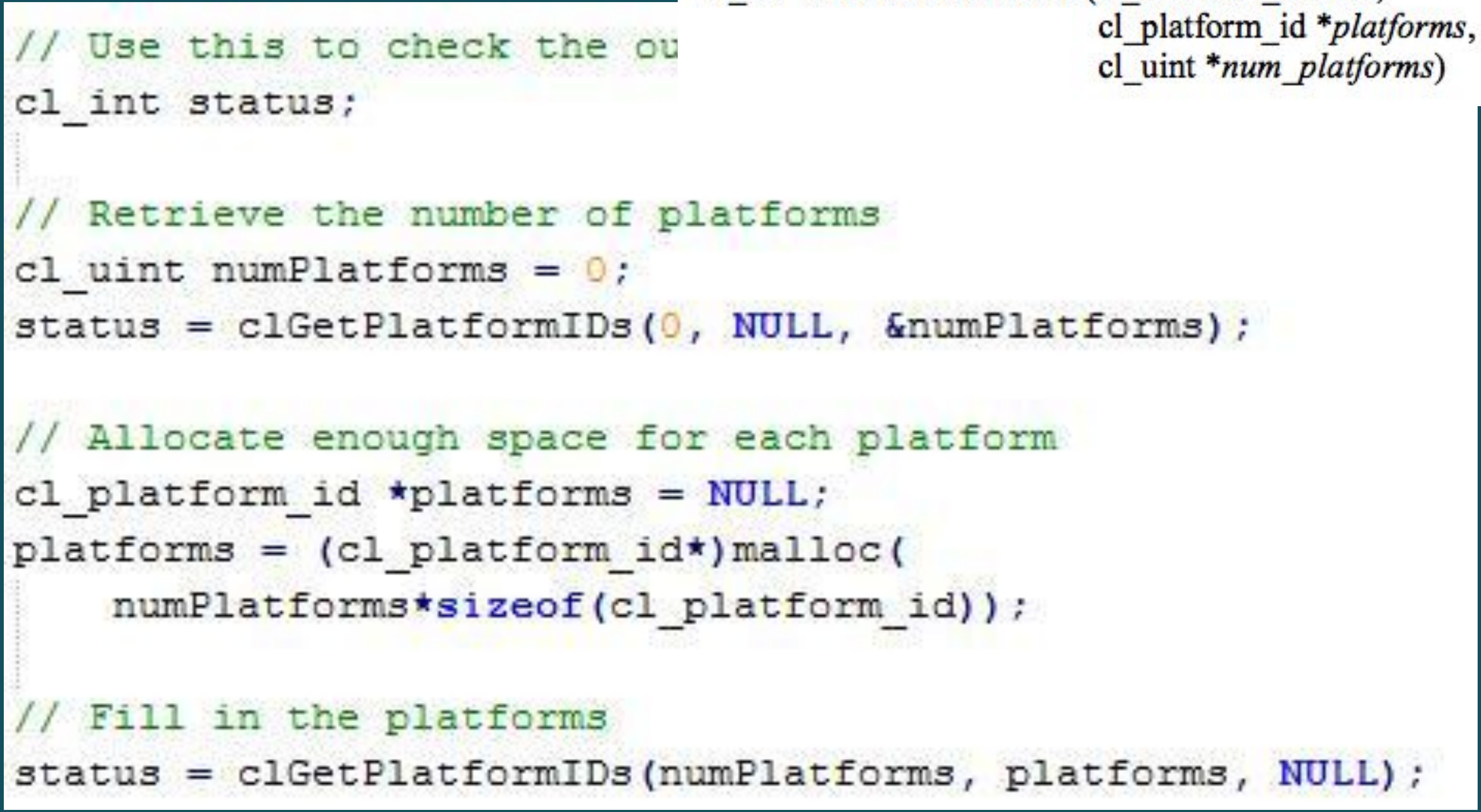
```
cl_int  clGetPlatformIDs (cl_uint num_entries,
                          cl_platform_id *platforms,
                          cl_uint *num_platforms)
```

- 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 a Platform



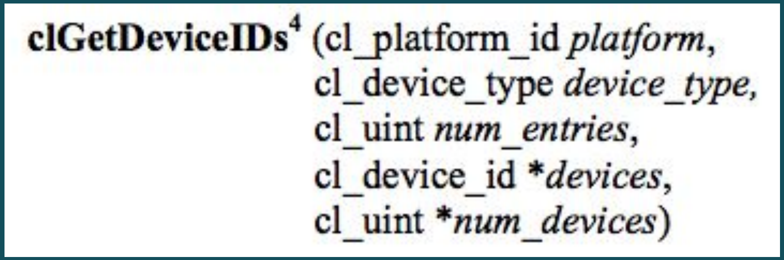
```
cl_int  clGetPlatformIDs (cl_uint num_entries,
                          cl_platform_id *platforms,
                          cl_uint *num_platforms)
```



```
// Use this to check the ou
cl_int status;

// Retrieve the number of platforms
cl_uint numPlatforms = 0;
status = clGetPlatformIDs(0, NULL, &numPlatforms);

// Allocate enough space for each platform
cl_platform_id *platforms = NULL;
platforms = (cl_platform_id*)malloc(
    numPlatforms*sizeof(cl_platform_id));

// Fill in the platforms
status = clGetPlatformIDs(numPlatforms, platforms, NULL);
```

Selecting Devices

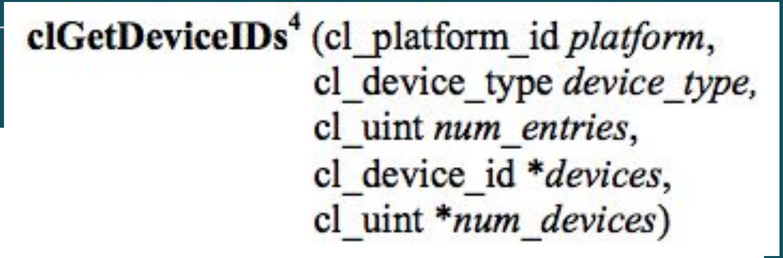
- Once a platform is selected, we can then query for the devices that it knows how to interact with



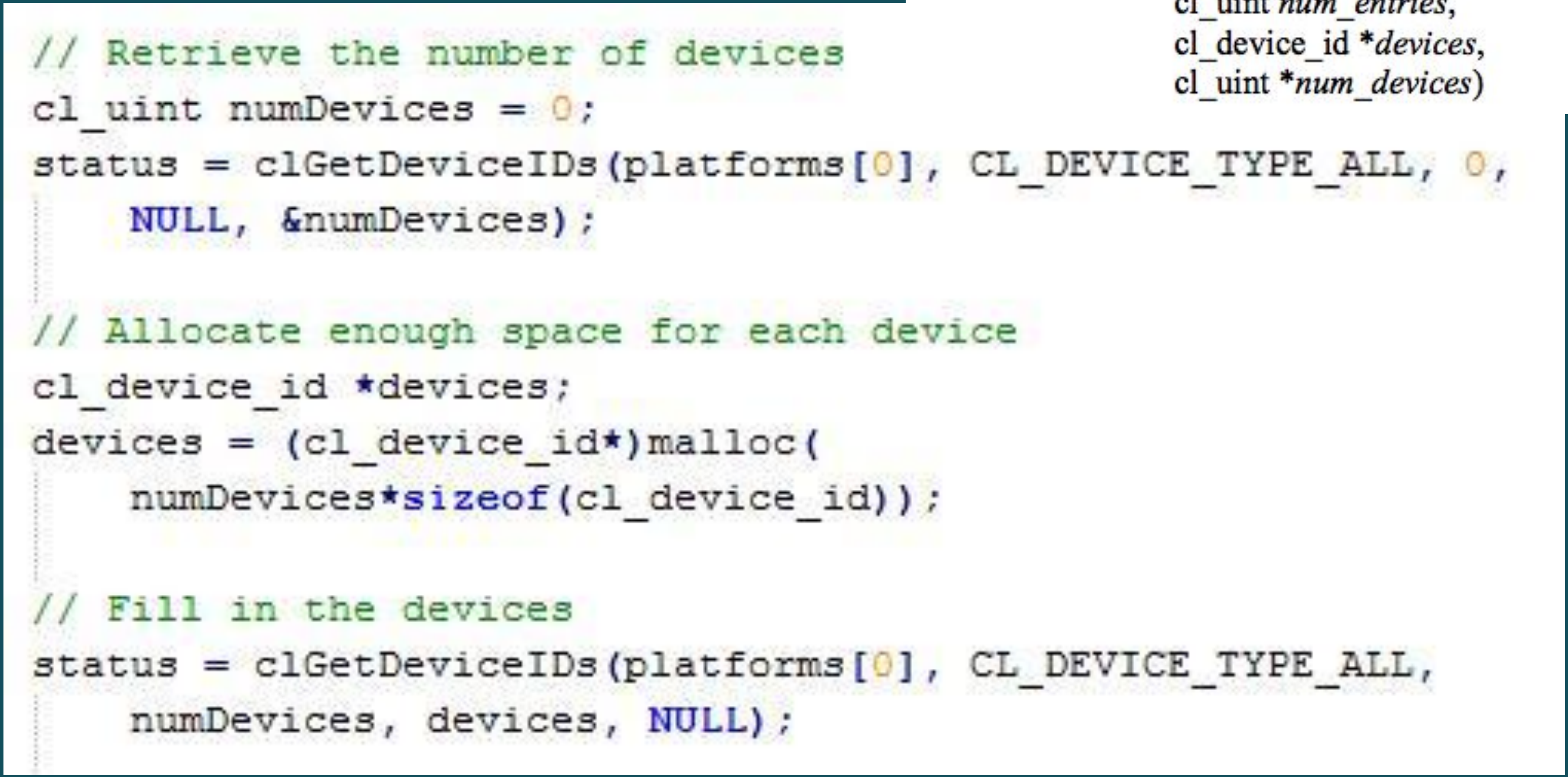
```
clGetDeviceIDs[4] (cl_platform_id platform,
                   cl_device_type device_type,
                   cl_uint num_entries,
                   cl_device_id *devices,
                   cl_uint *num_devices)
```

- 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

Selecting Devices



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



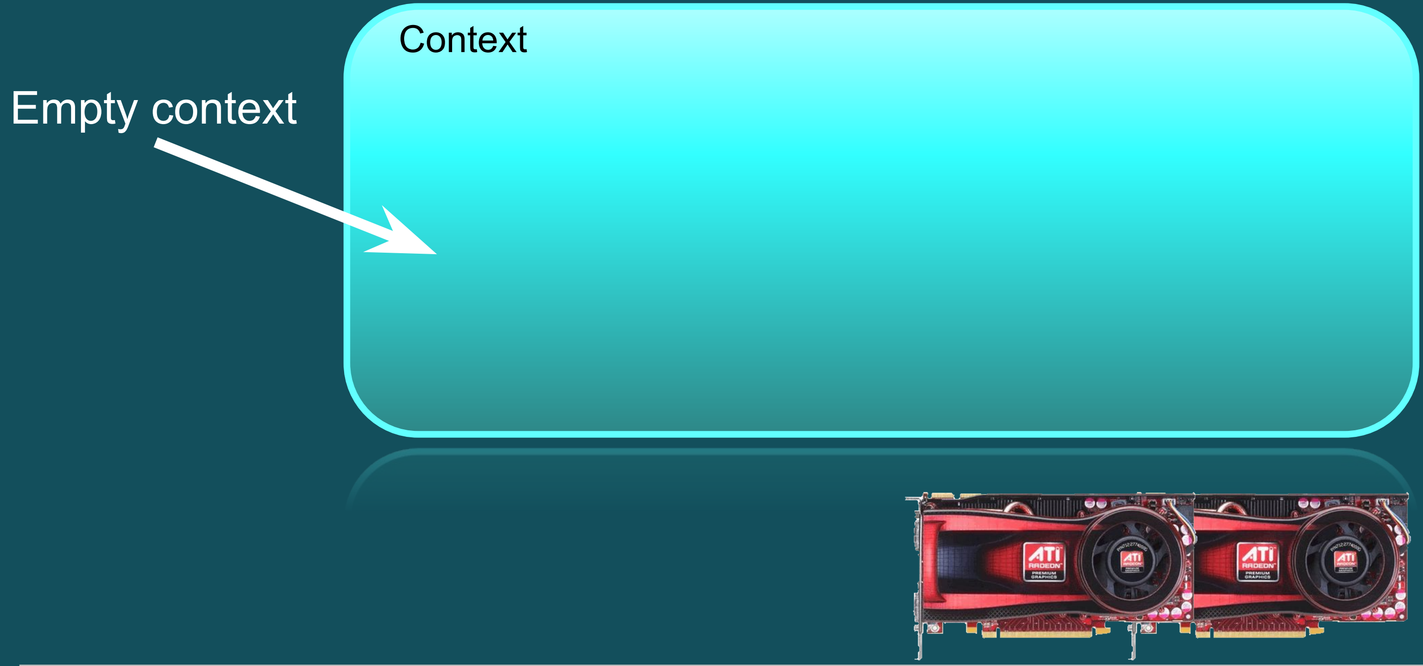
```
// Retrieve the number of devices
cl_uint numDevices = 0;
status = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL, 0,
    NULL, &numDevices);

// Allocate enough space for each device
cl_device_id *devices;
devices = (cl_device_id*)malloc(
    numDevices*sizeof(cl_device_id));

// Fill in the devices
status = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL,
    numDevices, devices, NULL);
```

Contexts

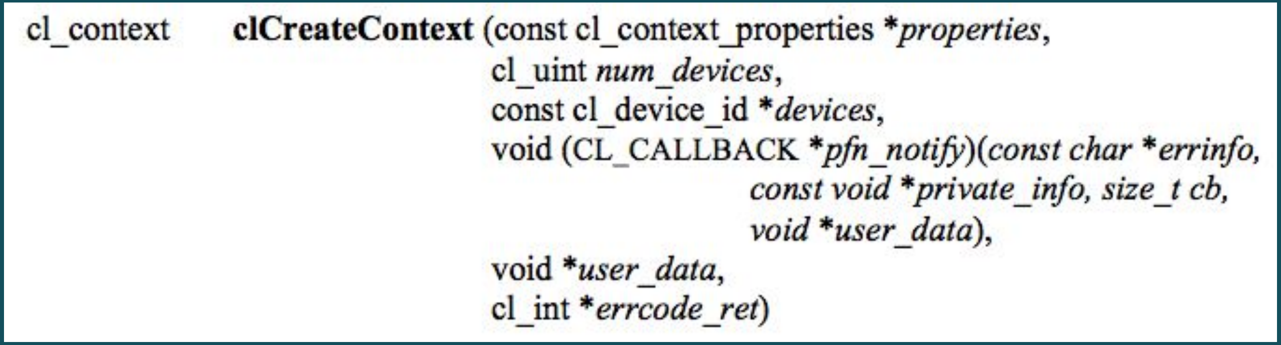
- A context refers to the environment for managing OpenCL objects and resources
- To manage OpenCL programs, the following are associated with a context
  - Devices: the things doing the execution
  - Program objects: the program source that implements the kernels
  - Kernels: functions that run on OpenCL 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



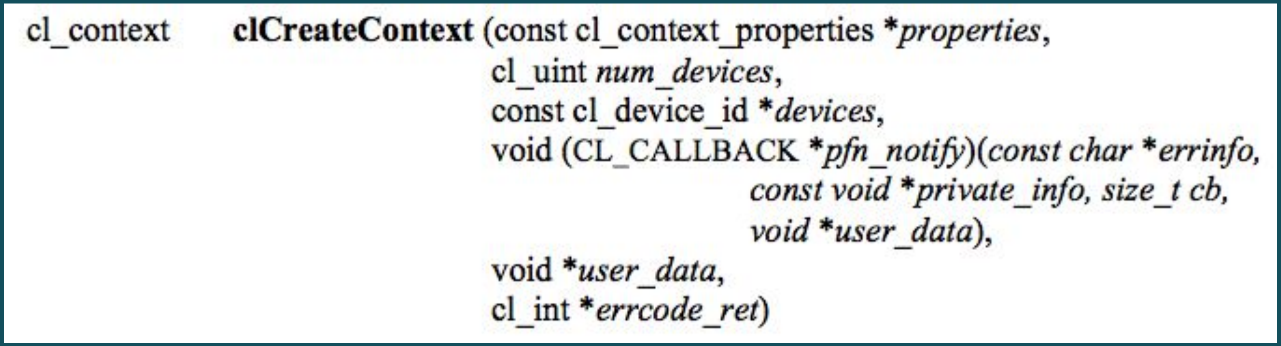
Contexts



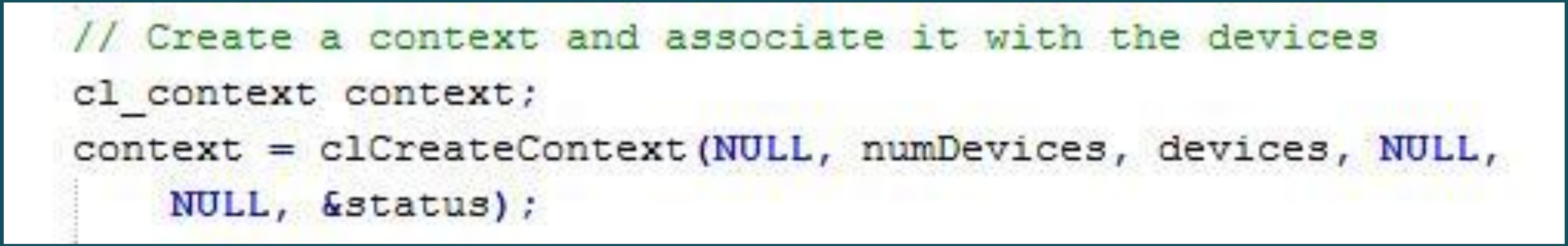
```
cl_context    clCreateContext (const cl_context_properties *properties,
                              cl_uint num_devices,
                              const cl_device_id *devices,
                              void (CL_CALLBACK *pfn_notify)(const char *errinfo,
                                             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

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)
```

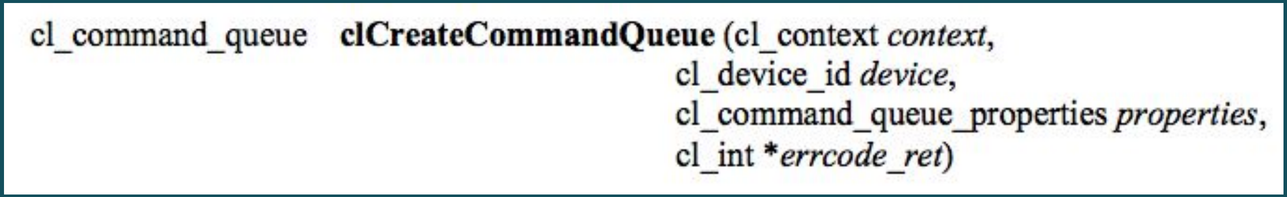


```
// Create a context and associate it with the devices
cl_context context;
context = clCreateContext(NULL, numDevices, devices, NULL,
    NULL, &status);
```

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

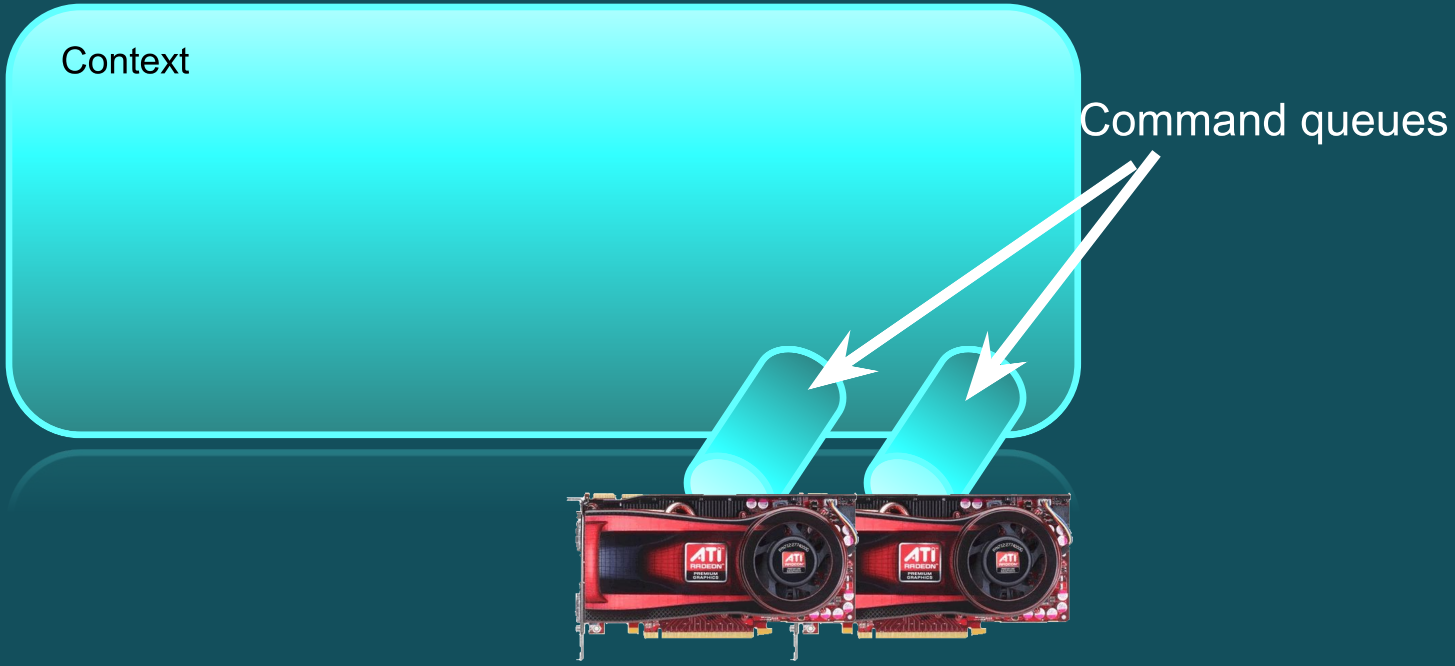


```
cl_command_queue   clCreateCommandQueue (cl_context context,
                                         cl_device_id device,
                                         cl_command_queue_properties properties,
                                         cl_int *errcode_ret)
```

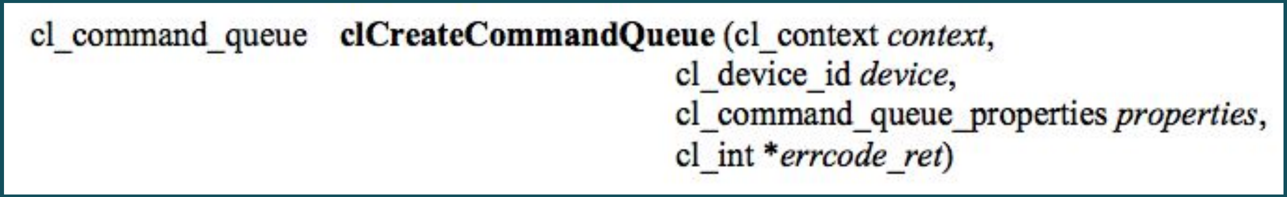
- A command queue establishes a relationship between a context and a device
- The command queue properties specify:
  - If out-of-order execution of commands is allowed
  - If profiling is enabled
    - Profiling is done using *events* (discussed in a later lecture) and will create some overhead

Command Queues

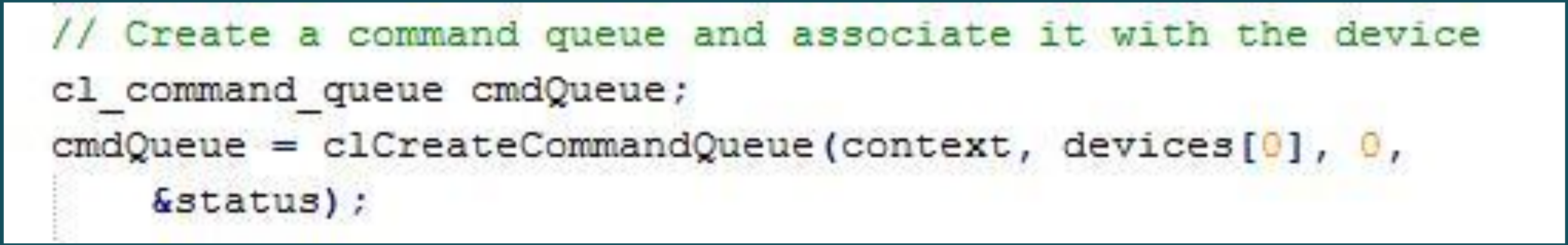
- Command queues associate a context with a device
  - Despite the figure below, they are not a physical connection



Command Queues



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



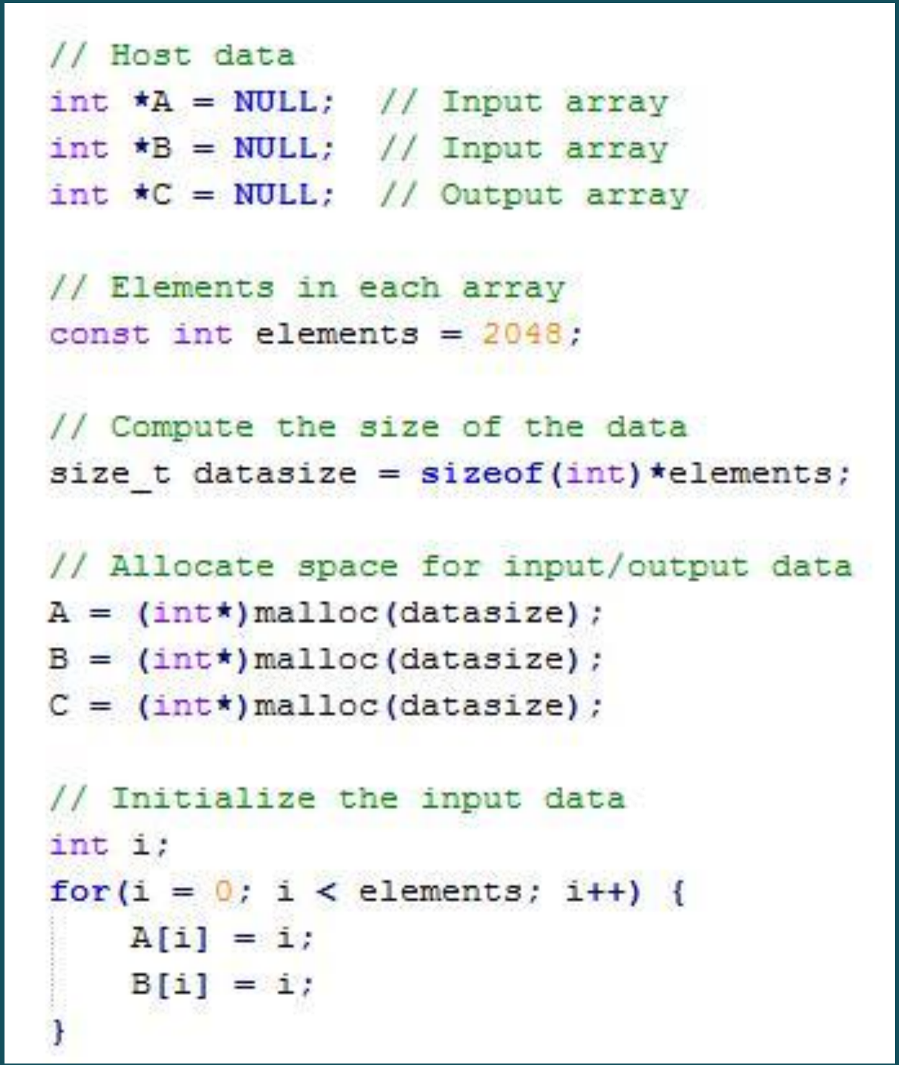
```
// Create a command queue and associate it with the device
cl_command_queue cmdQueue;
cmdQueue = clCreateCommandQueue(context, devices[0], 0,
    &status);
```

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

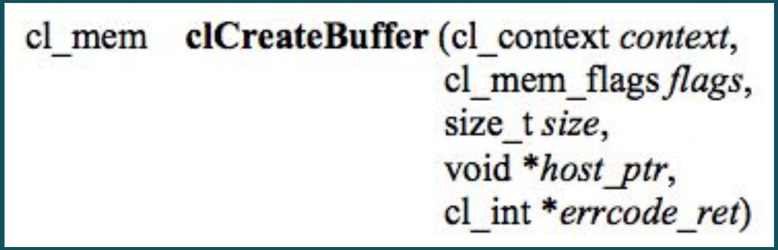
Memory Objects

- Data Must be created on the host side first!



```
// Host data
int *A = NULL;  // Input array
int *B = NULL;  // Input array
int *C = NULL;  // Output array

// Elements in each array
const int elements = 2048;

// Compute the size of the data
size_t datasize = sizeof(int)*elements;

// Allocate space for input/output data
A = (int*)malloc(datasize);
B = (int*)malloc(datasize);
C = (int*)malloc(datasize);

// Initialize the input data
int i;
for(i = 0; i < elements; i++) {
    A[i] = i;
    B[i] = i;
}
```

Creating buffers

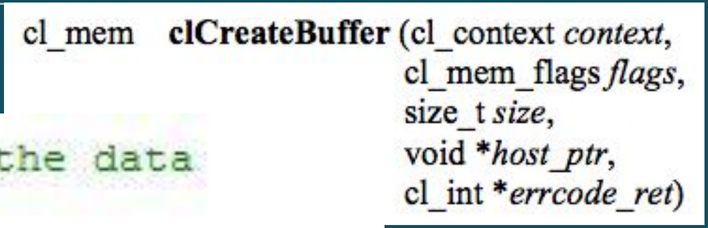


```
cl_mem    clCreateBuffer (cl_context context,
                          cl_mem_flags flags,
                          size_t size,
                          void *host_ptr,
                          cl_int *errcode_ret)
```

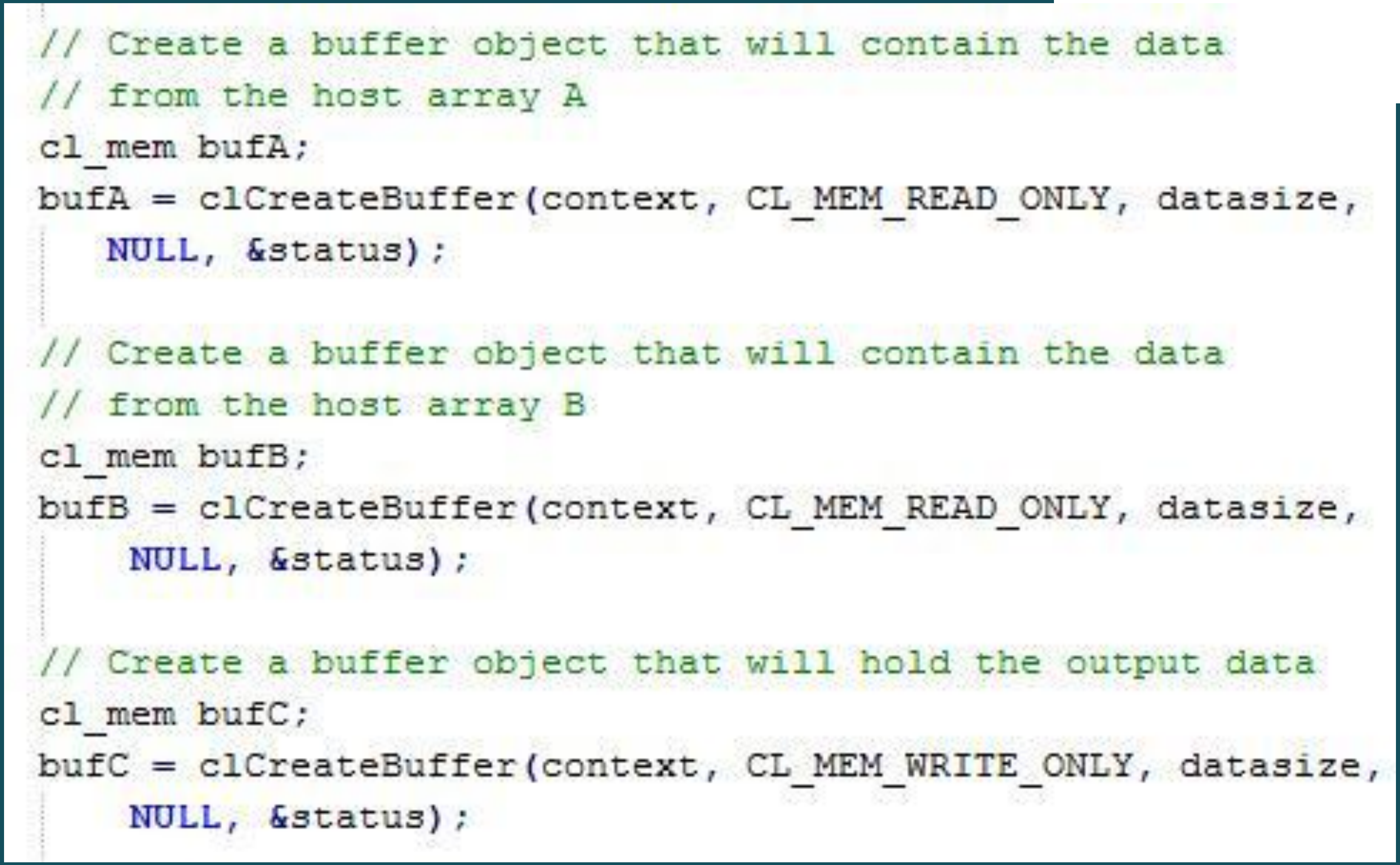
- This function creates a buffer (cl_mem object) for the given context
  - Images are more complex and will be covered in a later lecture

- 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

Creating buffers



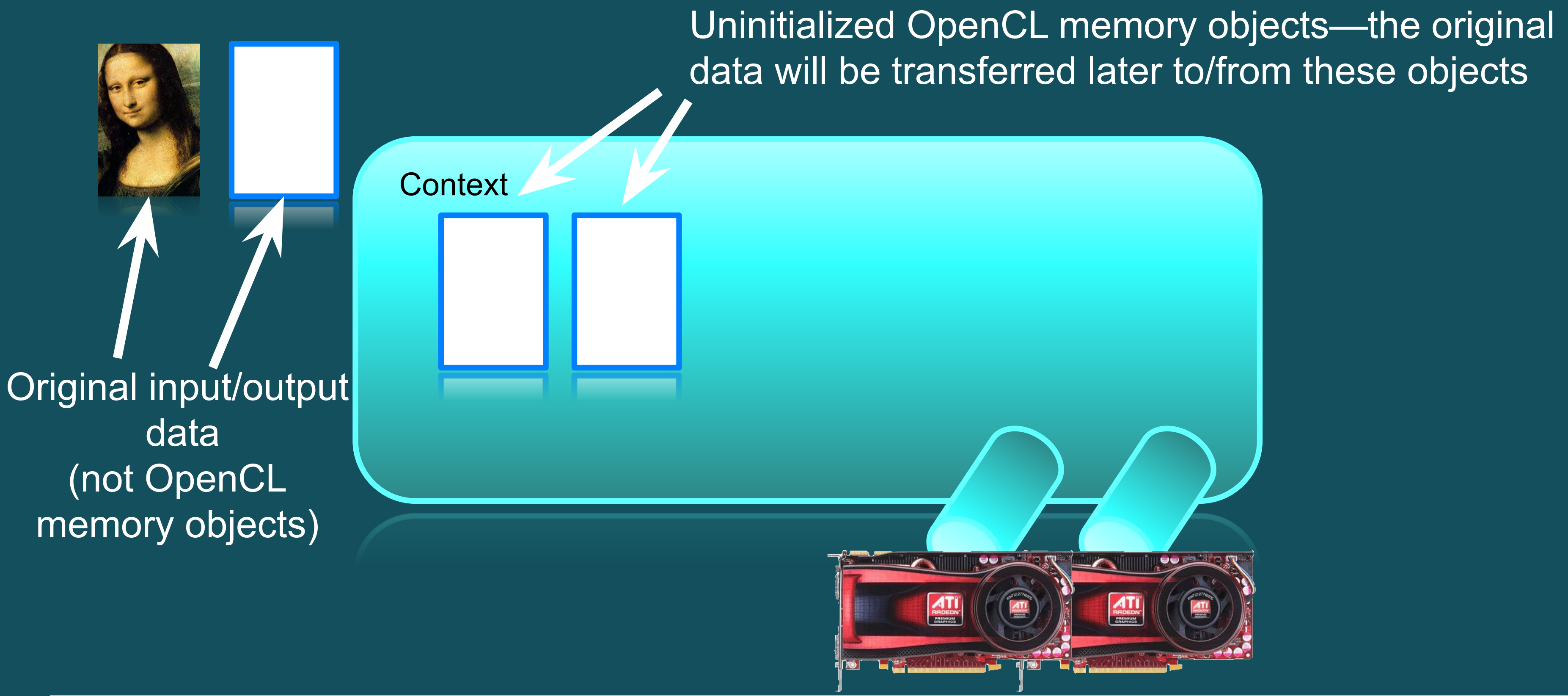
```
cl_mem    clCreateBuffer (cl_context context,
                          cl_mem_flags flags,
                          size_t size,
                          void *host_ptr,
                          cl_int *errcode_ret)
```



```
// Create a buffer object that will contain the data
// from the host array A
cl_mem bufA;
bufA = clCreateBuffer(context, CL_MEM_READ_ONLY, datasize,
   NULL, &status);

// Create a buffer object that will contain the data
// from the host array B
cl_mem bufB;
bufB = clCreateBuffer(context, CL_MEM_READ_ONLY, datasize,
    NULL, &status);

// Create a buffer object that will hold the output data
cl_mem bufC;
bufC = clCreateBuffer(context, CL_MEM_WRITE_ONLY, datasize,
    NULL, &status);
```

Memory Objects

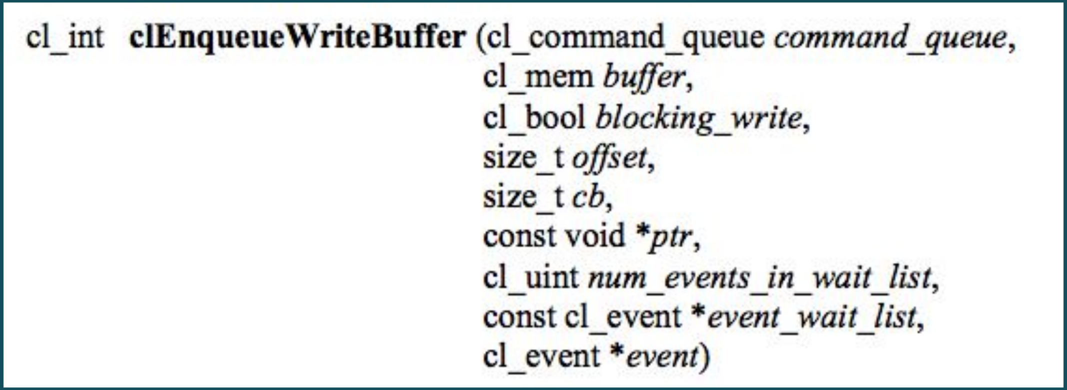
- Memory objects are associated with a context
  - They must be explicitly transferred to devices prior to execution (covered later)


Uninitialized OpenCL memory objects—the original data will be transferred later to/from these objects
Context
Original input/output data (not OpenCL memory objects)

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

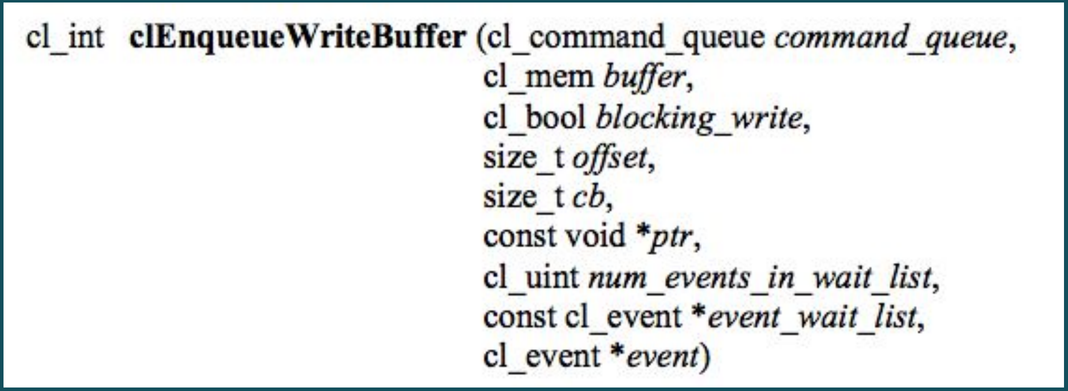
Transferring Data



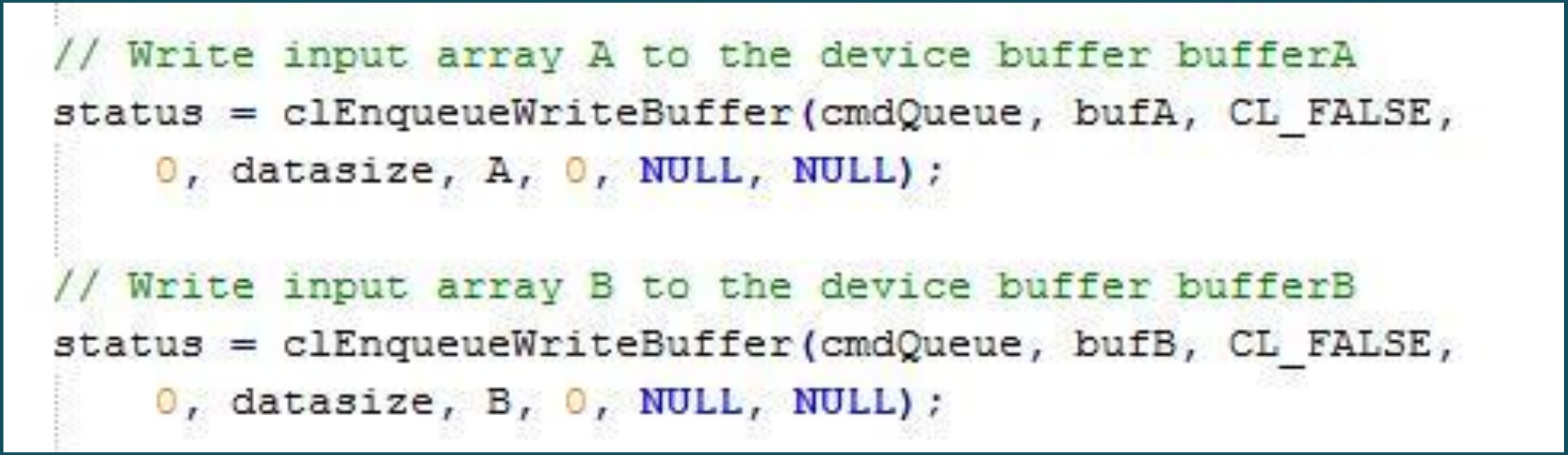
```
cl_int  clEnqueueWriteBuffer (cl_command_queue command_queue,
                              cl_mem buffer,
                              cl_bool blocking_write,
                              size_t offset,
                              size_t cb,
                              const void *ptr,
                              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



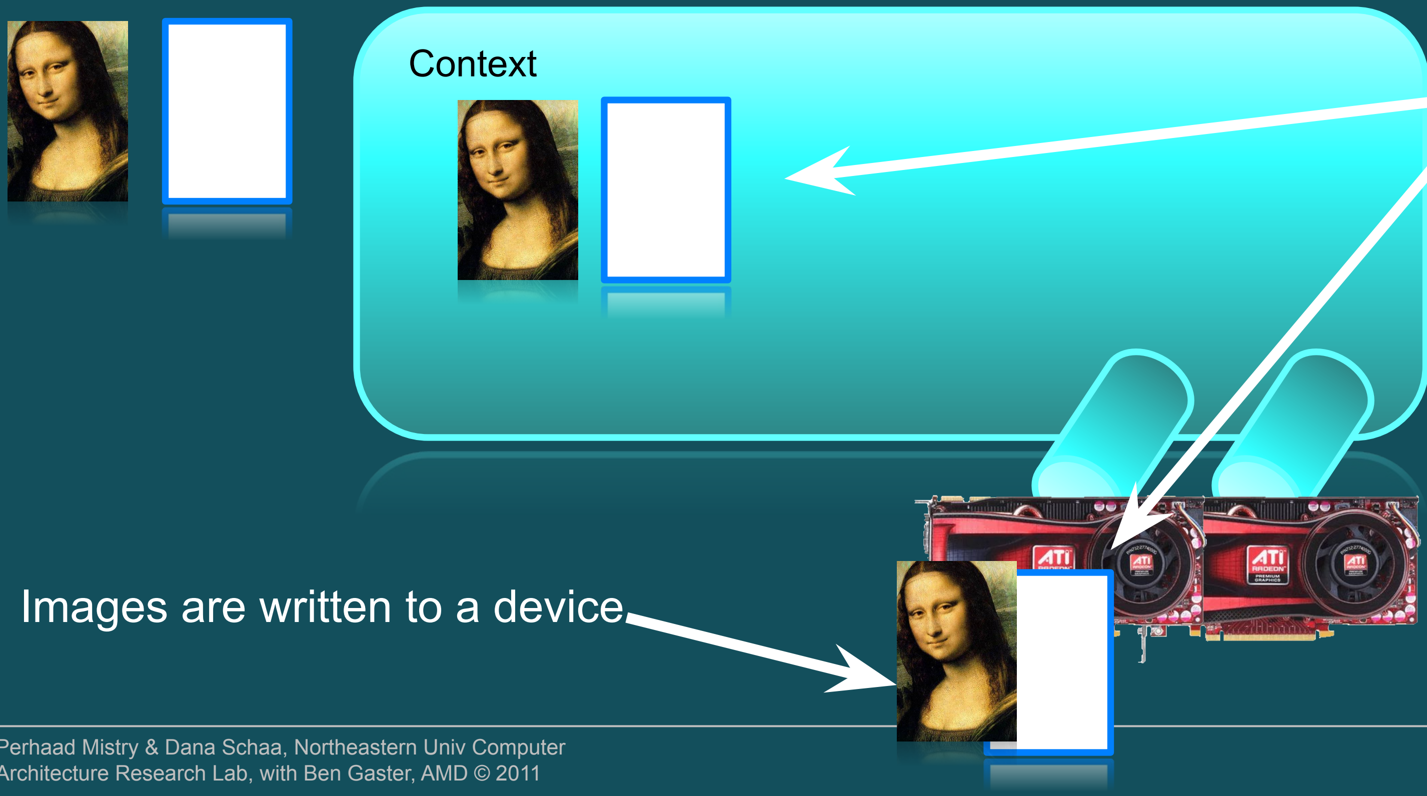
```
cl_int  clEnqueueWriteBuffer (cl_command_queue command_queue,
                              cl_mem buffer,
                              cl_bool blocking_write,
                              size_t offset,
                              size_t cb,
                              const void *ptr,
  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



```
// Write input array A to the device buffer bufferA
status = clEnqueueWriteBuffer(cmdQueue, bufA, CL_FALSE,
    0, datasize, A, 0, NULL, NULL);

// Write input array B to the device buffer bufferB
status = clEnqueueWriteBuffer(cmdQueue, bufB, CL_FALSE,
    0, datasize, B, 0, NULL, NULL);
```

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

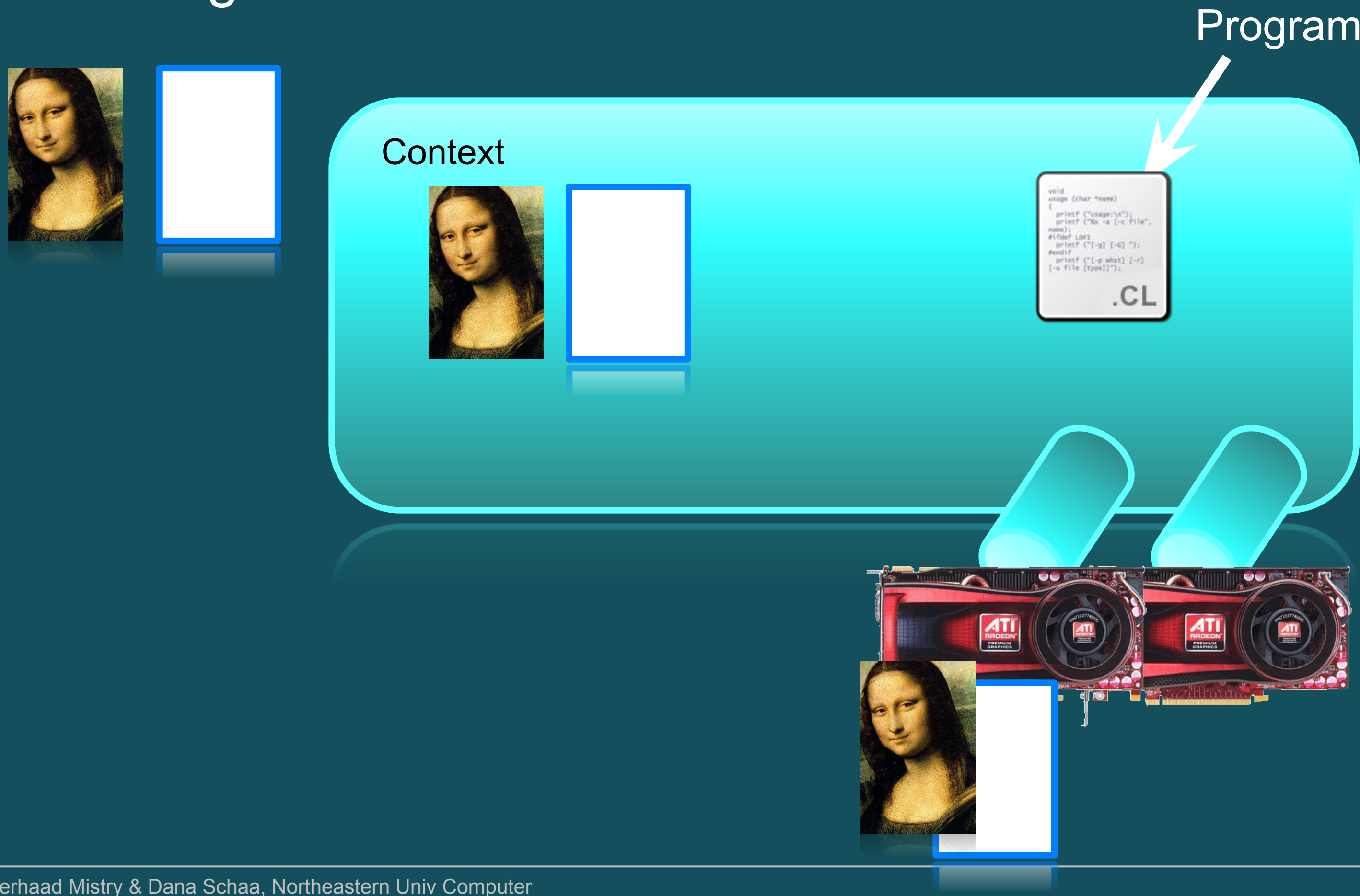
Images are written to a device

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)

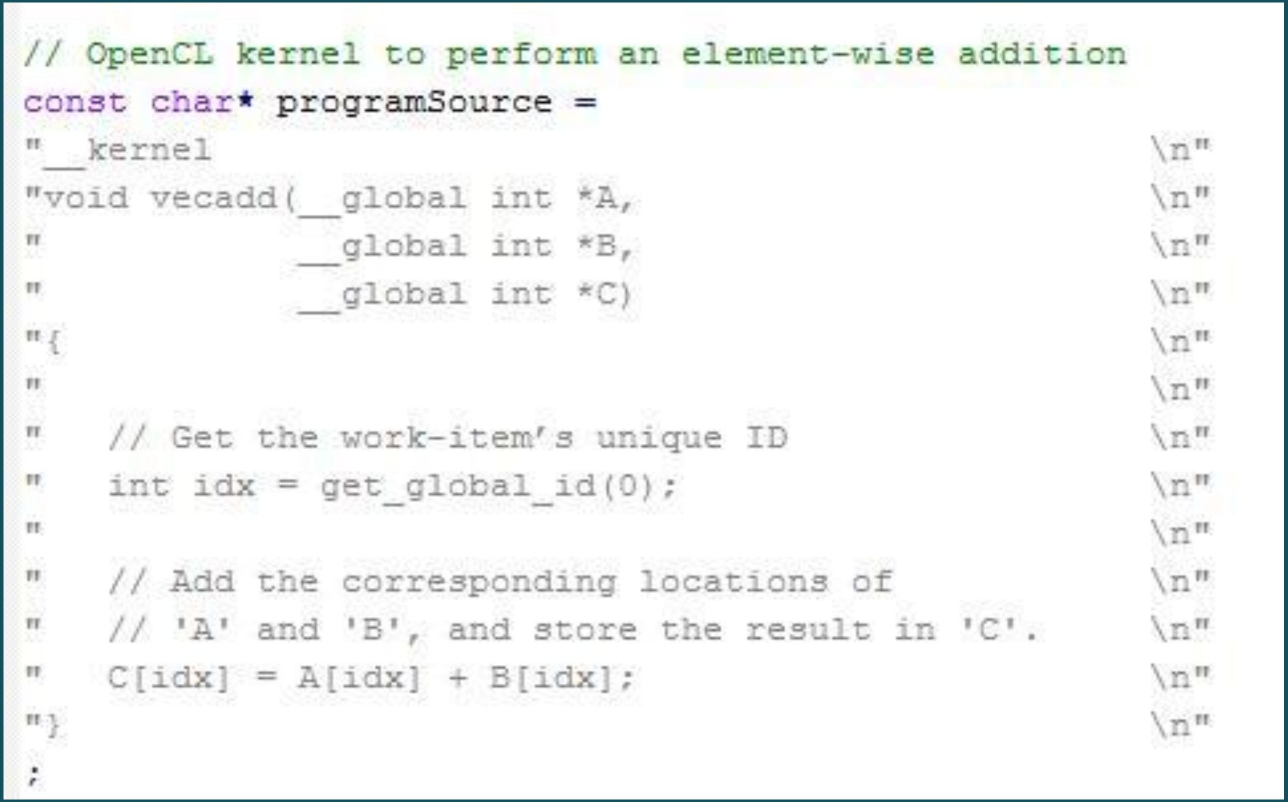
Programs

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



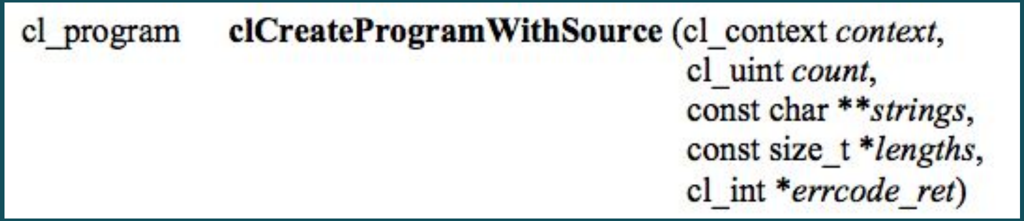
Creating Programs

- Programs are defined as source code within the host code:



```
// OpenCL kernel to perform an element-wise addition
const char* programSource =
"__kernel                                                  \n"
"void vecadd(__global int *A,                              \n"
"            __global int *B,                              \n"
"            __global int *C)                              \n"
"{                                                         \n"
"                                                          \n"
"   // Get the work-item's unique ID                       \n"
"   int idx = get_global_id(0);                            \n"
"                                                          \n"
"   // Add the corresponding locations of                  \n"
"   // 'A' and 'B', and store the result in 'C'.           \n"
"   C[idx] = A[idx] + B[idx];                              \n"
"}                                                         \n"
;
```

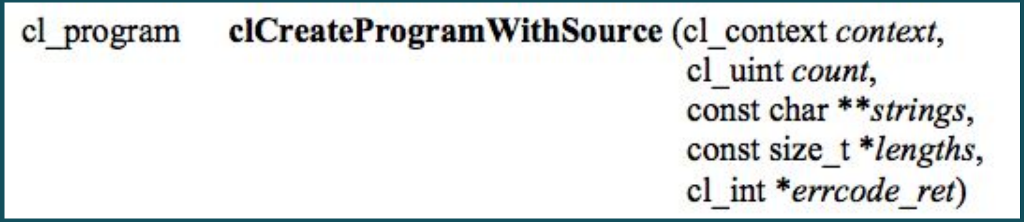
Creating Programs



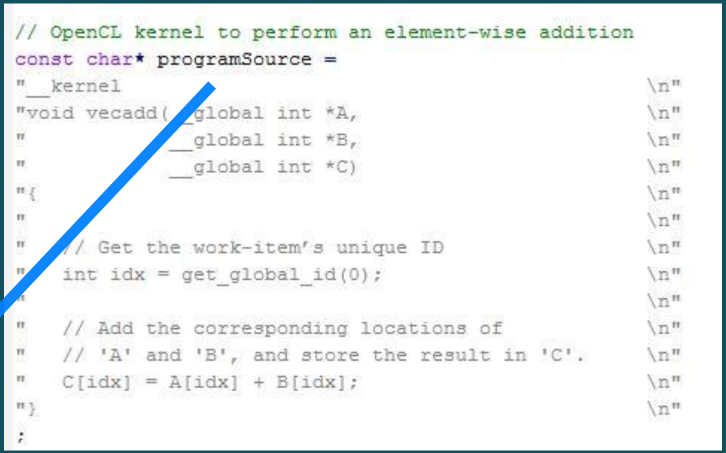
```
cl_program    clCreateProgramWithSource (cl_context context,
                                         cl_uint count,
                                         const char **strings,
                                         const size_t *lengths,
                                         cl_int *errcode_ret)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

- 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

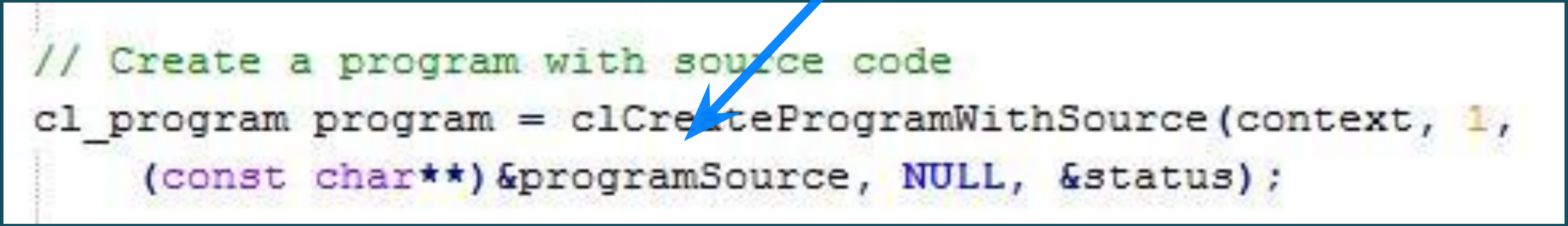
Creating Programs



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

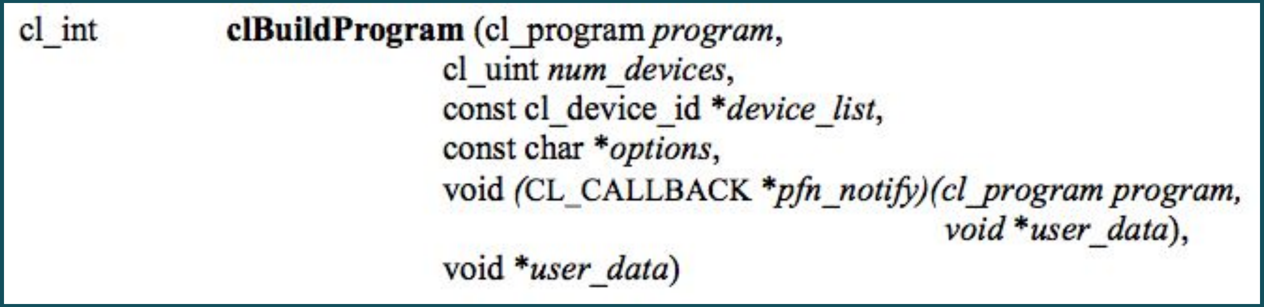


```
// OpenCL kernel to perform an element-wise addition
const char* programSource =
"__kernel                                                       \n"
"void vecadd(__global int *A,                                   \n"
"            __global int *B,                                   \n"
"            __global int *C)                                   \n"
"{                                                              \n"
"                                                               \n"
"   // Get the work-item's unique ID                            \n"
"   int idx = get_global_id(0);                                 \n"
"                                                               \n"
"   // Add the corresponding locations of                       \n"
"   // 'A' and 'B', and store the result in 'C'.                \n"
"   C[idx] = A[idx] + B[idx];                                   \n"
"}                                                              \n"
;
```



```
// Create a program with source code
cl_program program = clCreateProgramWithSource(context, 1,
    (const char**)&programSource, NULL, &status);
```

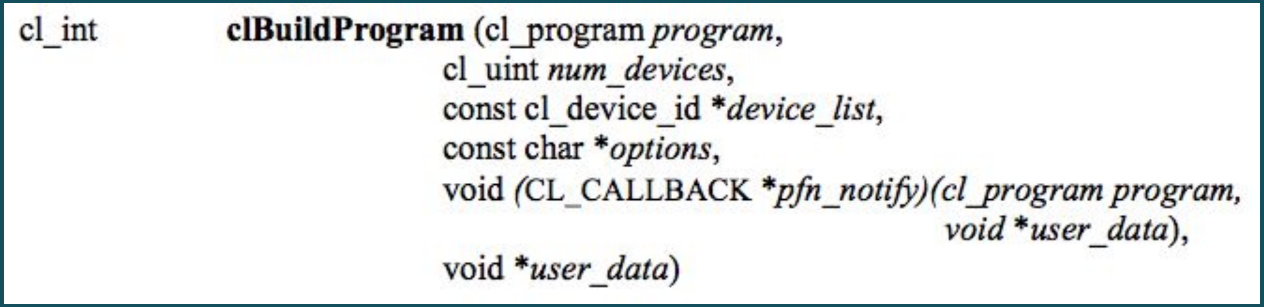
Compiling Programs



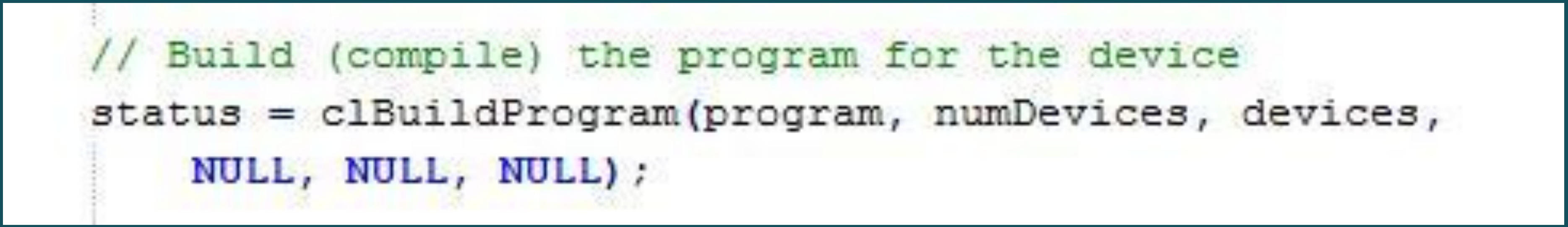
```
cl_int        clBuildProgram (cl_program program,
                              cl_uint num_devices,
                              const cl_device_id *device_list,
                              const char *options,
                              void (CL_CALLBACK *pfn_notify)(cl_program program,
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

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)
```



```
// Build (compile) the program for the device
status = clBuildProgram(program, numDevices, devices,
    NULL, NULL, NULL);
```

Reporting Compile Errors

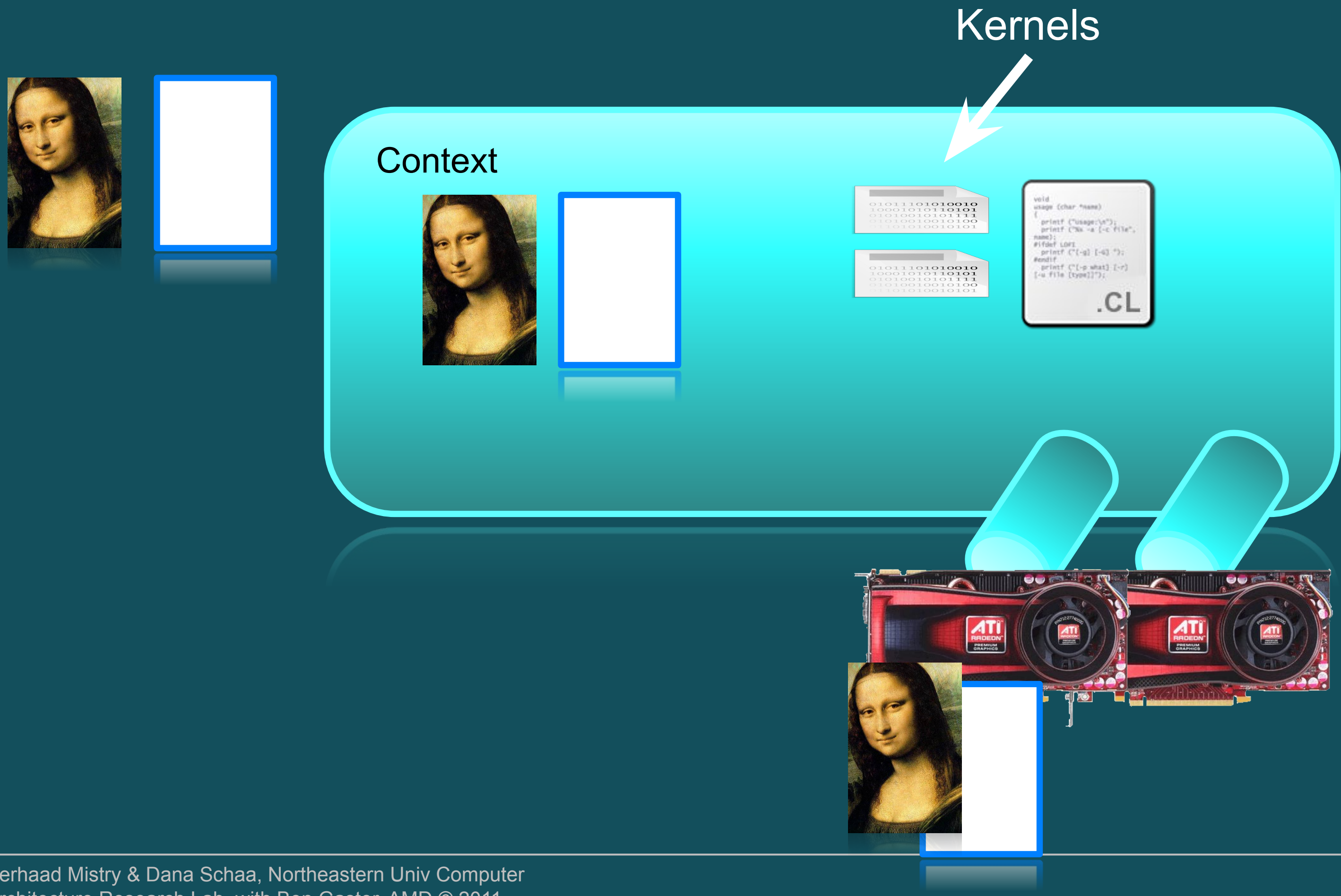
- If a program fails to compile, OpenCL requires the programmer to explicitly ask for compiler output
  - A compilation failure is determined by an error value returned from clBuildProgram()
  - Calling clGetProgramBuildInfo() with the program object and the parameter CL_PROGRAM_BUILD_STATUS returns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

Kernels

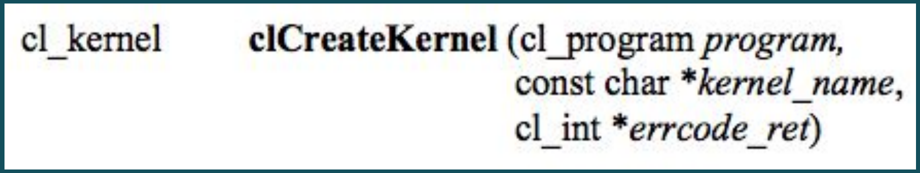
- A kernel is a function declared in a program that is executed on an OpenCL device
  - A kernel object is a kernel function along with its associated arguments
- A kernel object is created from a compiled program
- Must explicitly associate arguments (memory objects, primitives, etc) with the kernel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

Kernels

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



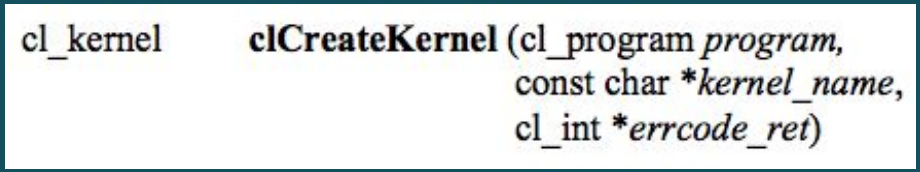
Kernels



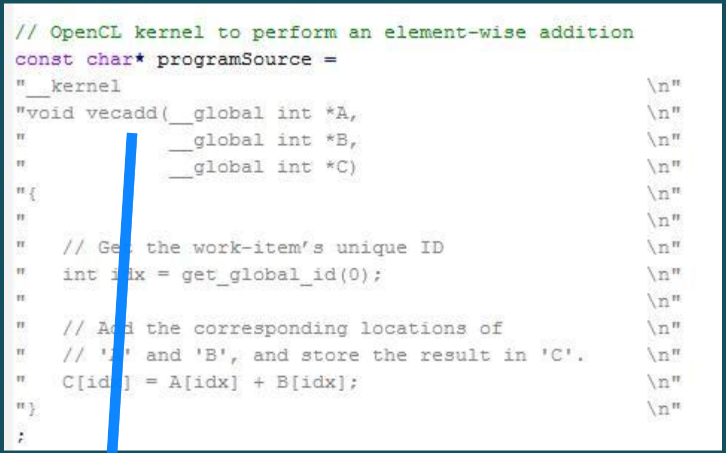
```
cl_kernel      clCreateKernel (cl_program program,
                               const char *kernel_name,
                               cl_int *errcode_ret)
```

- 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

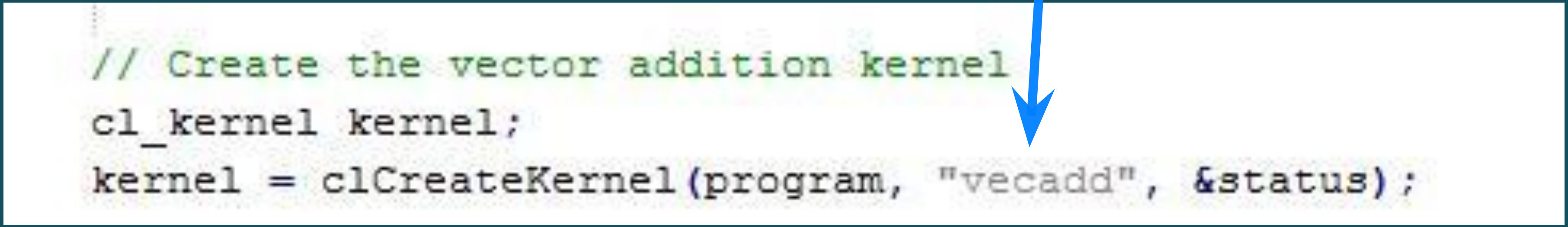
Kernels



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



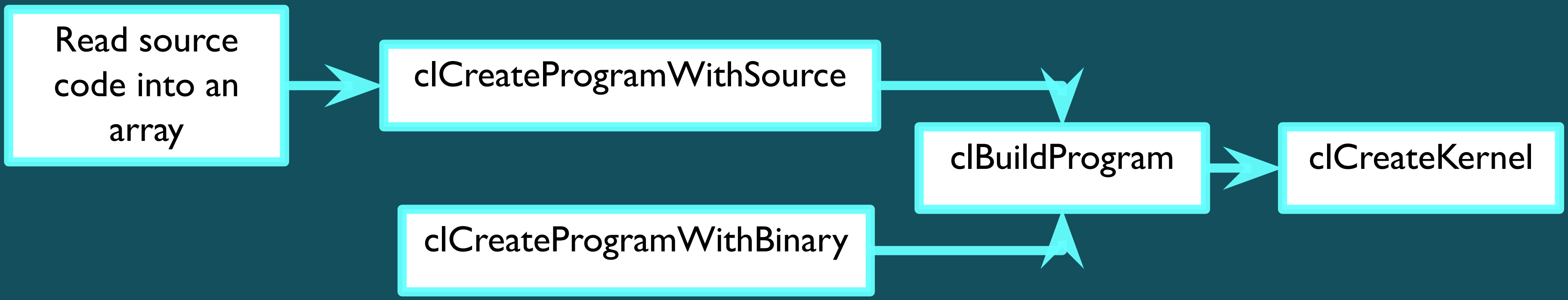
```
// OpenCL kernel to perform an element-wise addition
const char* programSource =
"__kernel                                              \n"
"void vecadd(__global int *A,                          \n"
"            __global int *B,                          \n"
"            __global int *C)                          \n"
"{                                                     \n"
"                                                      \n"
"   // Get the work-item's unique ID                   \n"
"   int idx = get_global_id(0);                        \n"
"                                                      \n"
"   // Add the corresponding locations of              \n"
"   // 'A' and 'B', and store the result in 'C'.       \n"
"   C[idx] = A[idx] + B[idx];                          \n"
"}                                                     \n"
;
```



```
// Create the vector addition kernel
cl_kernel kernel;
kernel = clCreateKernel(program, "vecadd", &status);
```

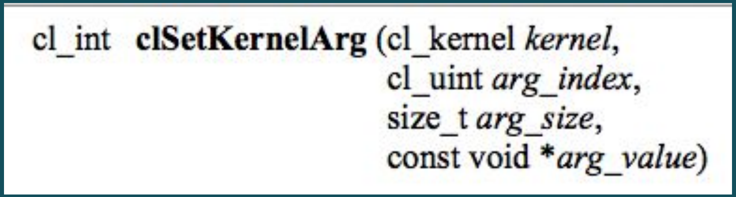
Runtime Compilation

- 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


Read source code into an array
clCreateProgramWithSource
clCreateProgramWithBinary
clBuildProgram
clCreateKernel

Setting Kernel Arguments

- Kernel arguments are set by repeated calls to clSetKernelArgs

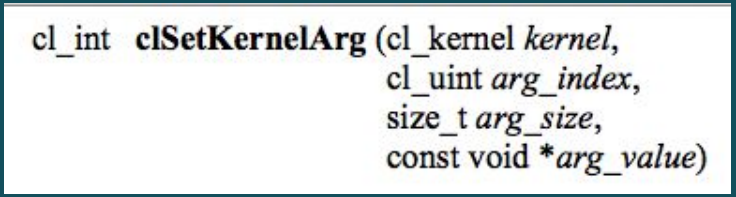


```
cl_int  clSetKernelArg (cl_kernel kernel,
                        cl_uint arg_index,
                        size_t arg_size,
                        const void *arg_value)
```

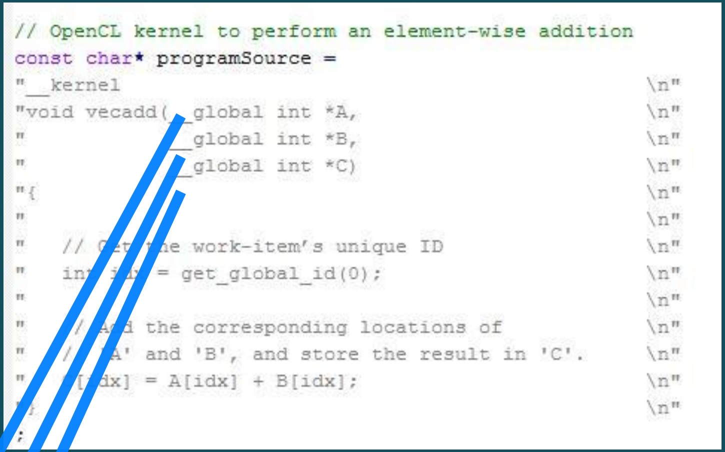
- Each call must specify:
  - The index of the argument as it appears in the function signature, the size, and a pointer to the data
- Examples:
  - `clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*)&d_iImage);`
  - `clSetKernelArg(kernel, 1, sizeof(int), (void*)&a);`

Setting Kernel Arguments

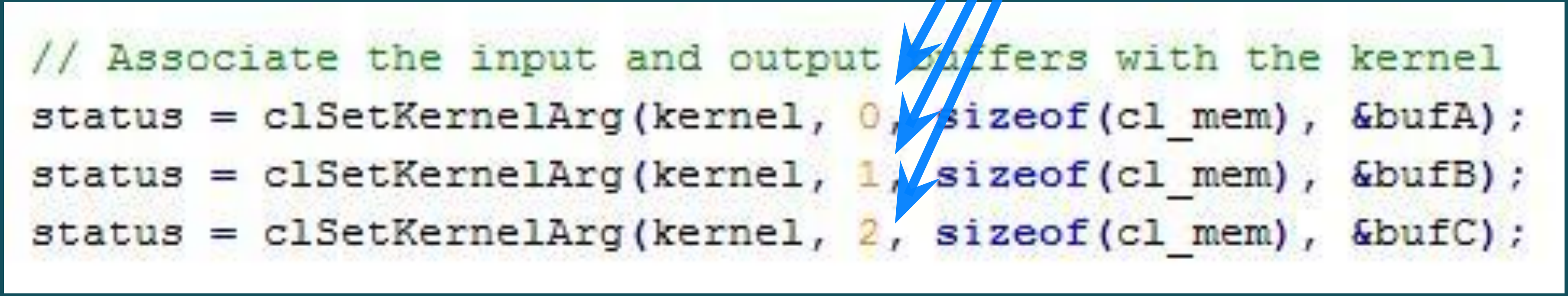
- Kernel arguments are set by repeated calls to clSetKernelArgs



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



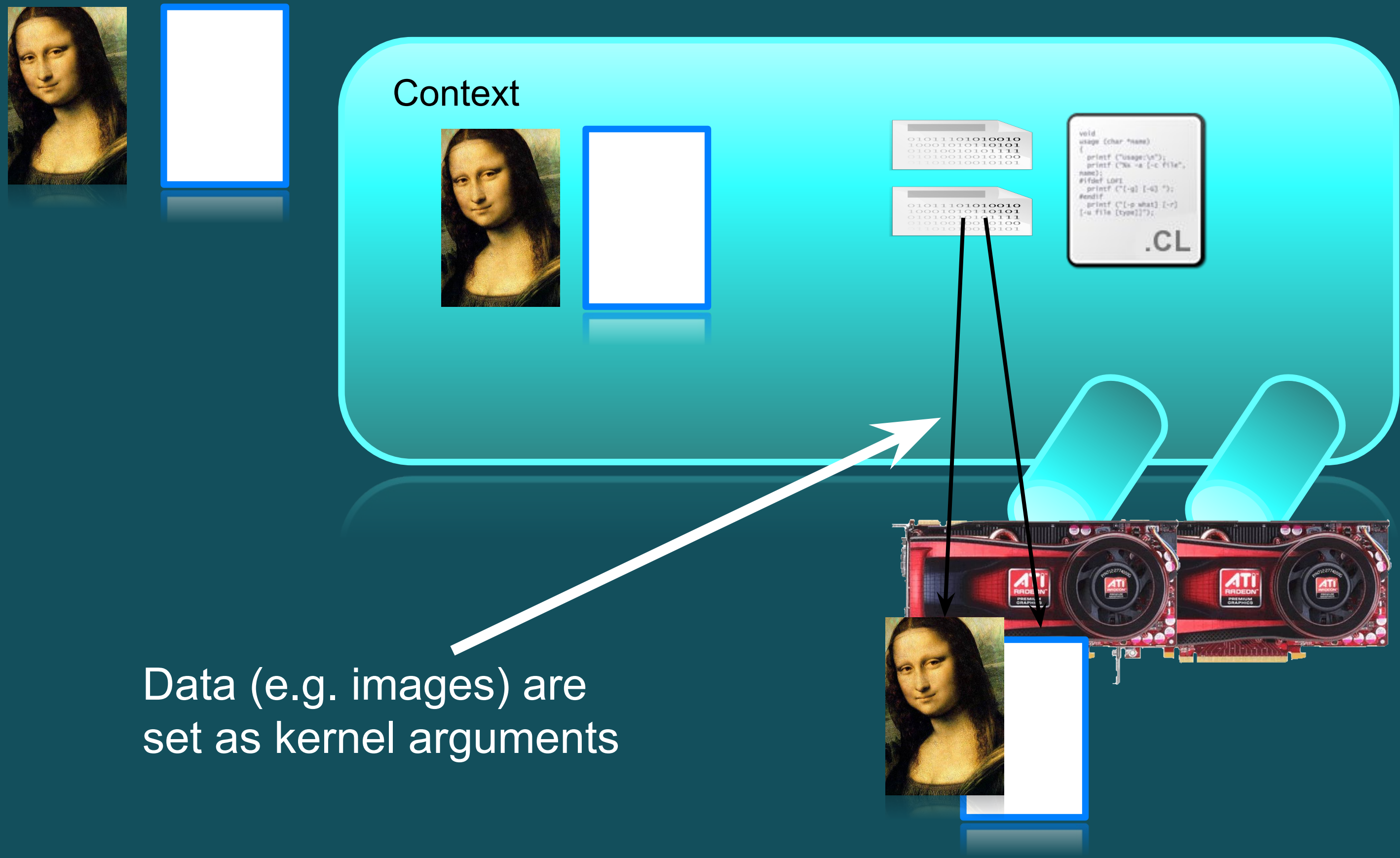
```
// OpenCL kernel to perform an element-wise addition
const char* programSource =
"__kernel                                              \n"
"void vecadd(__global int *A,                          \n"
"            __global int *B,                          \n"
"            __global int *C)                          \n"
"{                                                     \n"
"                                                      \n"
"   // Get the work-item's unique ID                   \n"
"   int idx = get_global_id(0);                        \n"
"                                                      \n"
"   // Add the corresponding locations of              \n"
"   // 'A' and 'B', and store the result in 'C'.       \n"
"   C[idx] = A[idx] + B[idx];                          \n"
"}                                                     \n"
;
```



```
// Associate the input and output buffers with the kernel
status = clSetKernelArg(kernel, 0, sizeof(cl_mem), &bufA);
status = clSetKernelArg(kernel, 1, sizeof(cl_mem), &bufB);
status = clSetKernelArg(kernel, 2, sizeof(cl_mem), &bufC);
```

Kernel Arguments

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



OpenCL WorkFlow

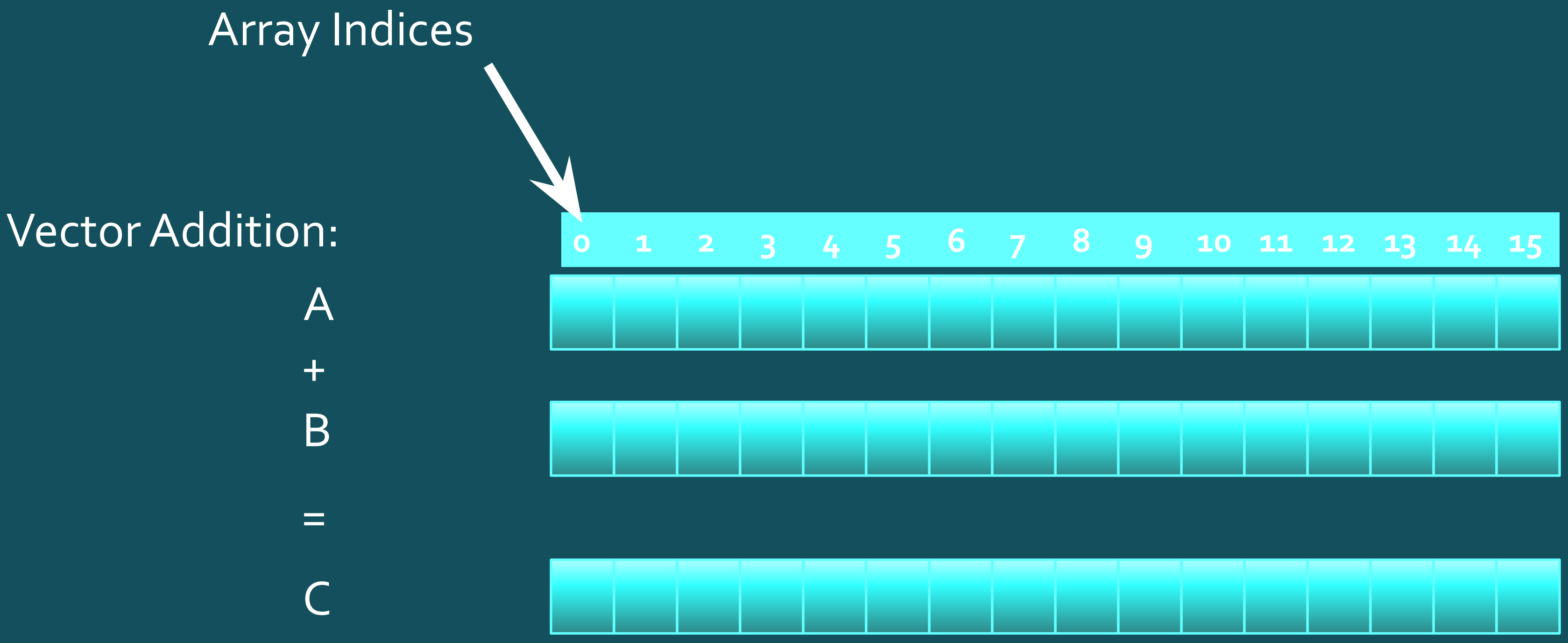
- Up to here… the workflow is straightforward…

Thread Structure

- Massively parallel programs are usually written so that each thread computes one part of a problem (*Data-Parallelism is very common*)
  - 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

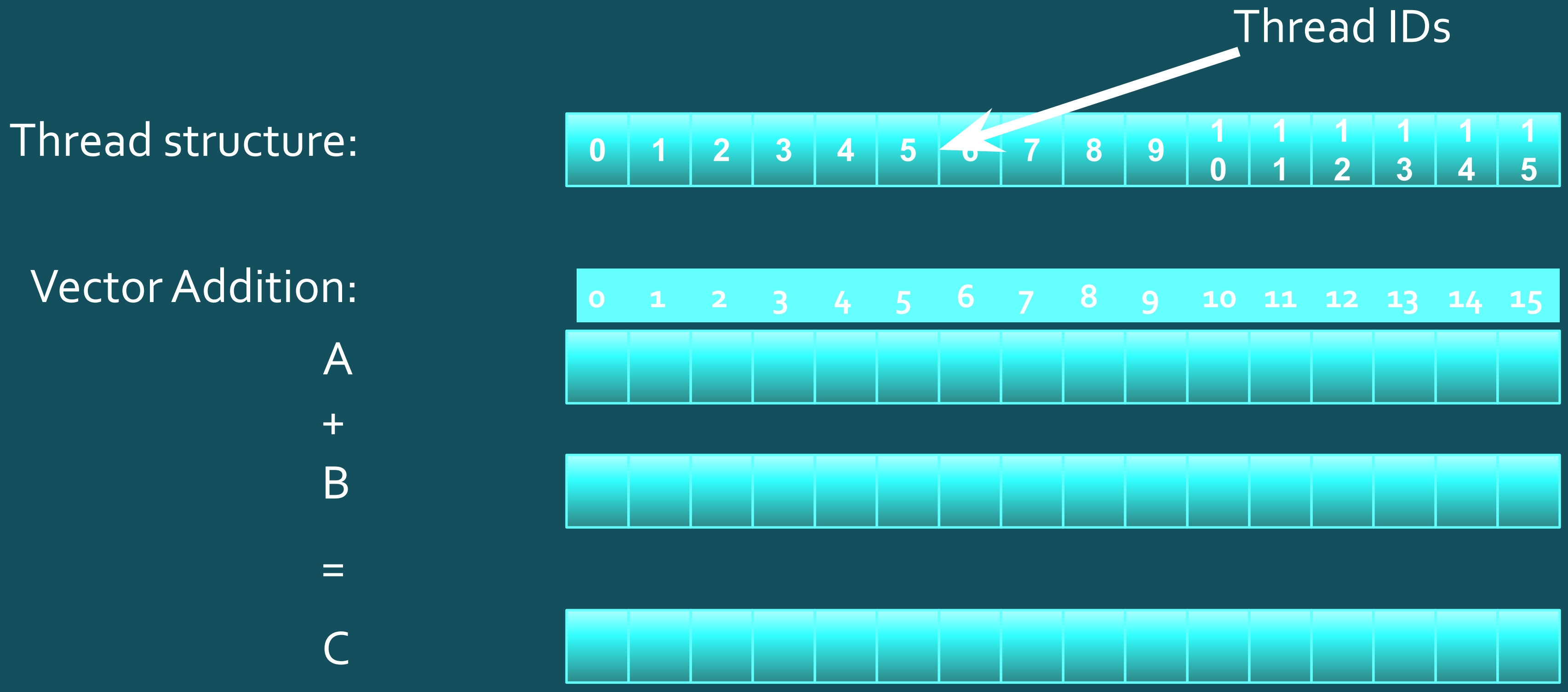
Thread Structure

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



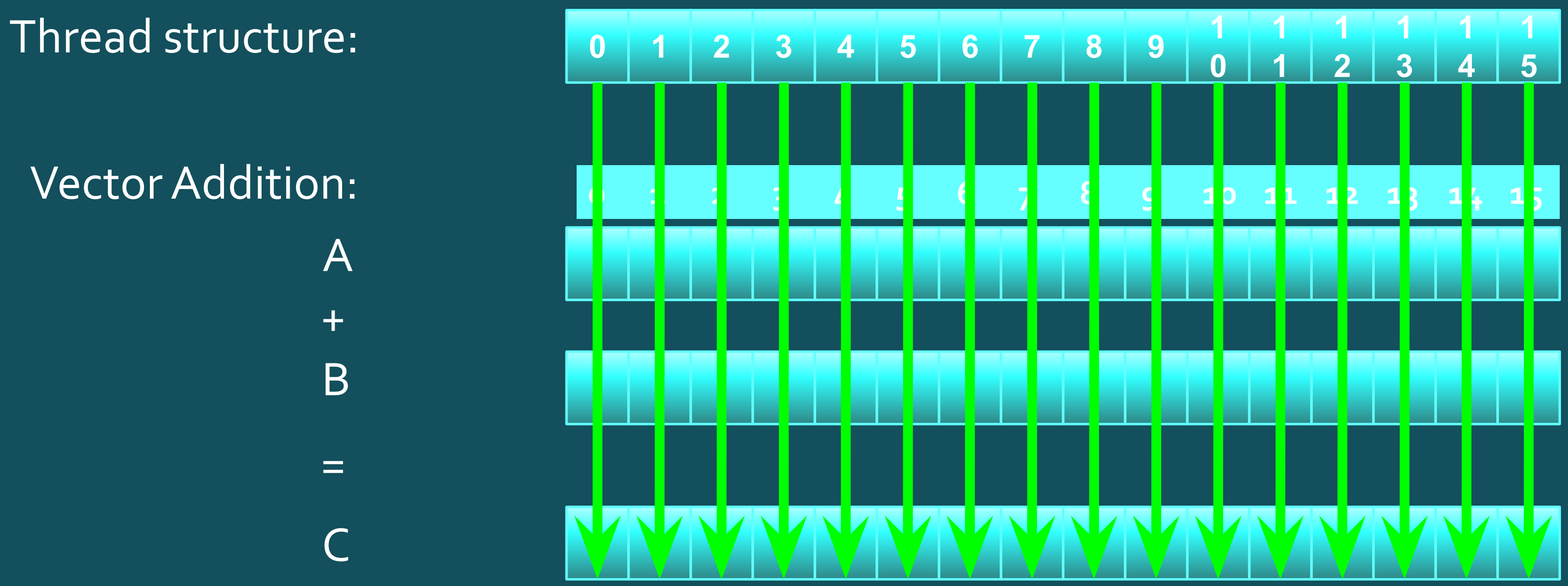
Thread Structure

- 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



Work Items

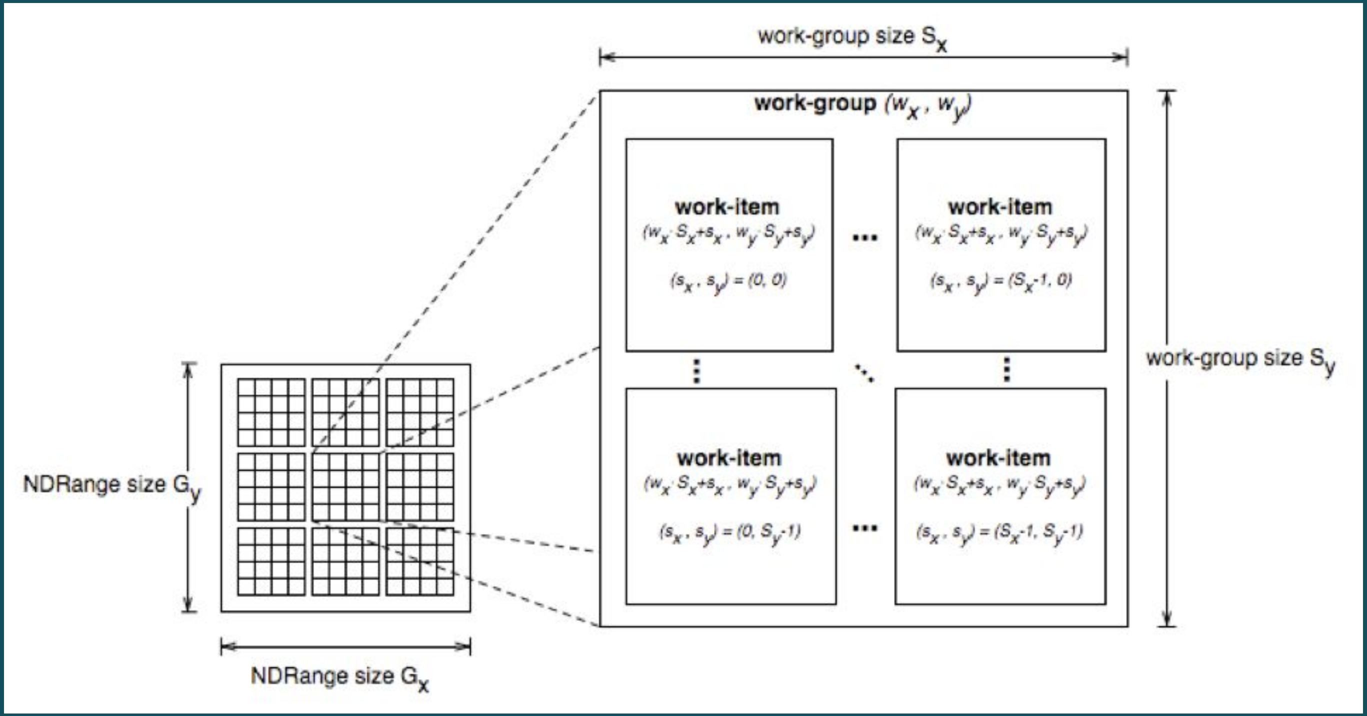
- The unit of concurrent execution in OpenCL C is a work-item. Each **work-item** executes the **kernel** function body
- Each instance of a kernel is called a work-item (though “thread” is commonly used as well)
- Instead of manually **strip mining the loop**, we will often map a single iteration of the loop to a work-item.
- An n-dimensional range (**NDRange**) index space defines a hierarchy of work-groups and work-items.

Work Groups

- Achieving scalability comes from dividing the work-items of an NDRange into smaller, equally sized workgroups.
  - Work-groups are independent from one-another (this is where scalability comes from)
- Work items within a workgroup have a special relationship with one another: They can perform barrier operations to synchronize and they have access to a shared memory address space.
- Because workgroup sizes are fixed, this communication does not have a need to scale and hence does not affect scalability of a large concurrent dispatch.

Work Items and Groups

- 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


work-group size S_x
work-group (w_x, w_y)
work-item
$(w_x \cdot S_x + s_x, w_y \cdot S_y + s_y)$
$(s_x, s_y) = (0, 0)$
work-item
$(w_x \cdot S_x + s_x, w_y \cdot S_y + s_y)$
$(s_x, s_y) = (S_x - 1, 0)$
work-item
$(w_x \cdot S_x + s_x, w_y \cdot S_y + s_y)$
$(s_x, s_y) = (0, S_y - 1)$
work-item
$(w_x \cdot S_x + s_x, w_y \cdot S_y + s_y)$
$(s_x, s_y) = (S_x - 1, S_y - 1)$
work-group size S_y
NDRange size G_y
NDRange size G_x

OpenCL requires that the index space sizes are evenly divisible by the workgroup sizes in each dimension.

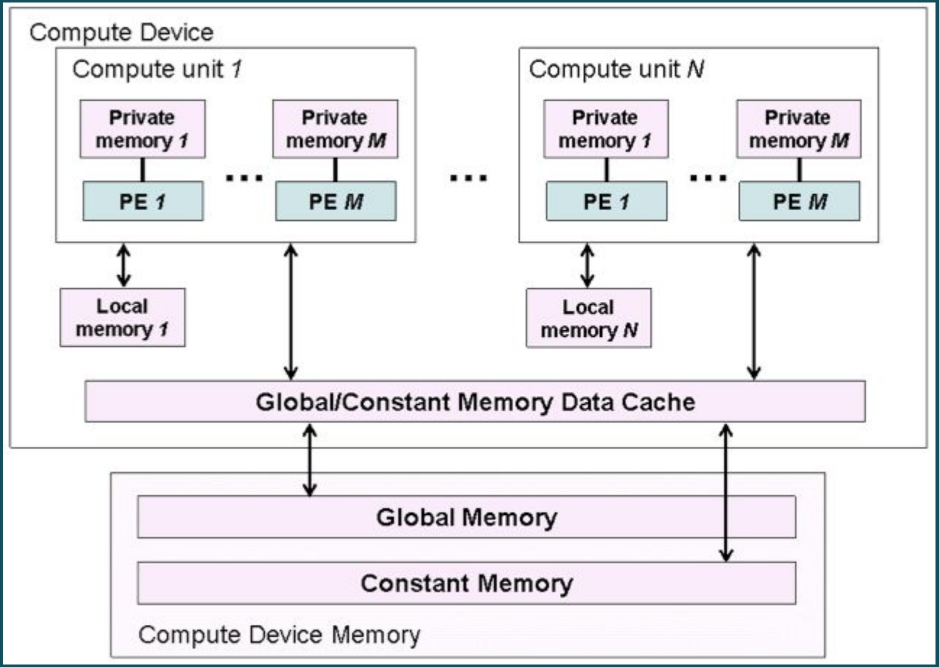
Work Items and Groups

- 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

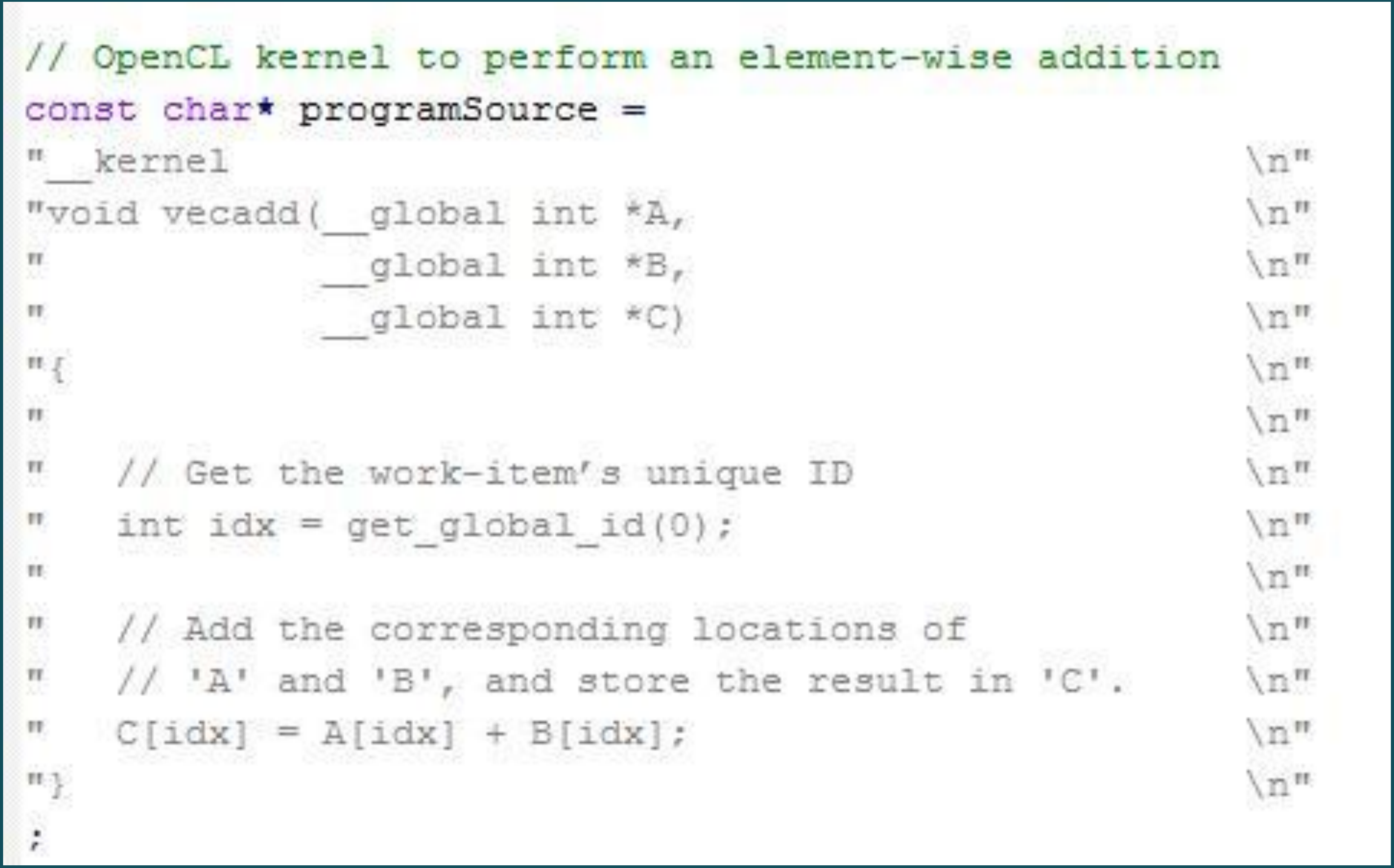
Address Space Identifiers

- `__global` – memory allocated from global address space to memory objects (buffers or images).
- `__constant` – a special type of read-only memory
- `__local` – memory shared by a work-group
- `__private` – private per work-item memory

- By default, kernel arguments are considered `__private`. If they are pointers or arrays, they can point to `__global`, `__local` or `__constant`.

Example Kernel

- Simple vector addition kernel:



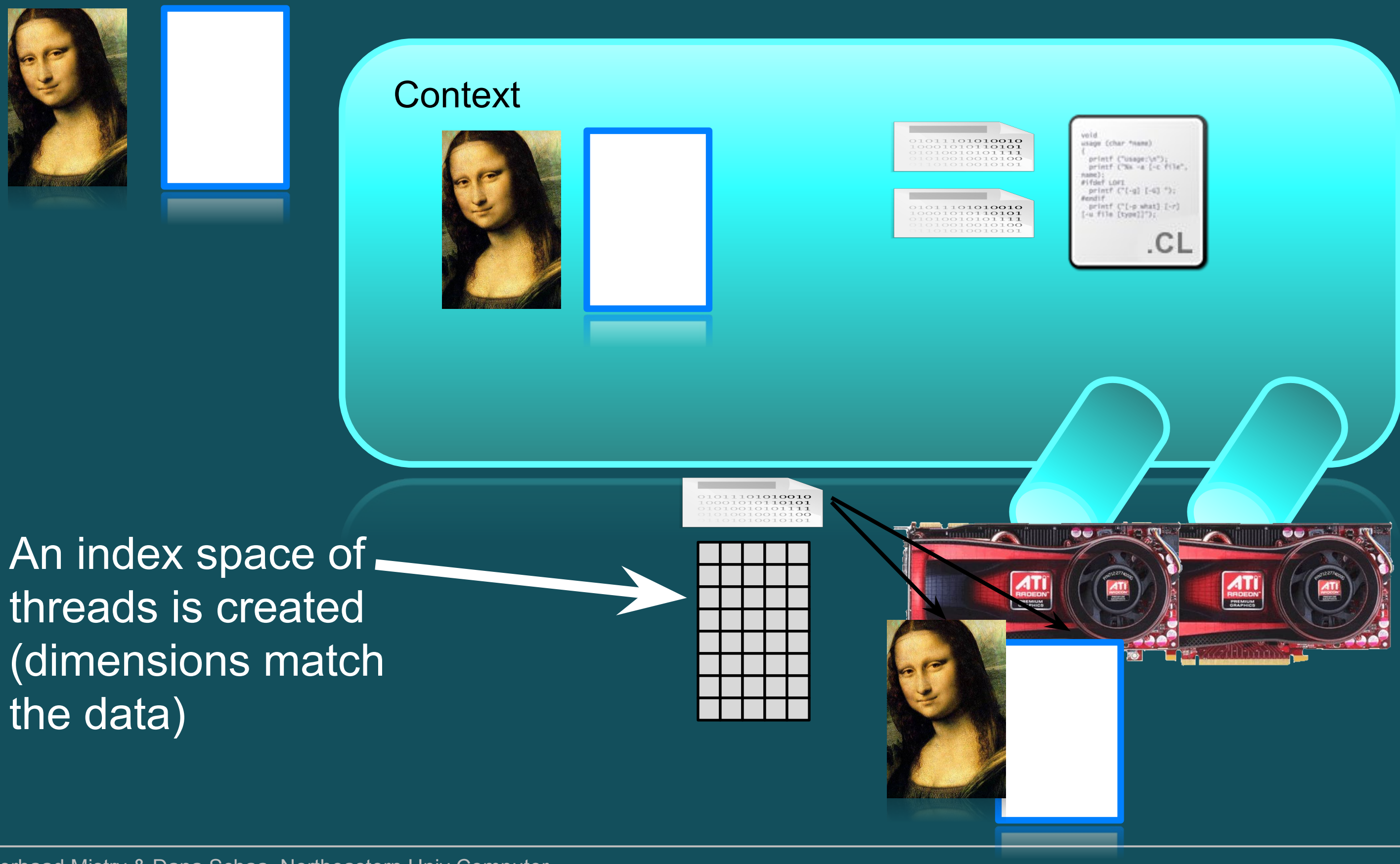
```
// OpenCL kernel to perform an element-wise addition
const char* programSource =
"__kernel                                                   \n"
"void vecadd(__global int *A,                               \n"
"            __global int *B,                               \n"
"            __global int *C)                               \n"
"{                                                          \n"
"                                                           \n"
"   // Get the work-item's unique ID                        \n"
"   int idx = get_global_id(0);                             \n"
"                                                           \n"
"   // Add the corresponding locations of                   \n"
"   // 'A' and 'B', and store the result in 'C'.            \n"
"   C[idx] = A[idx] + B[idx];                               \n"
"}                                                          \n"
;
```

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

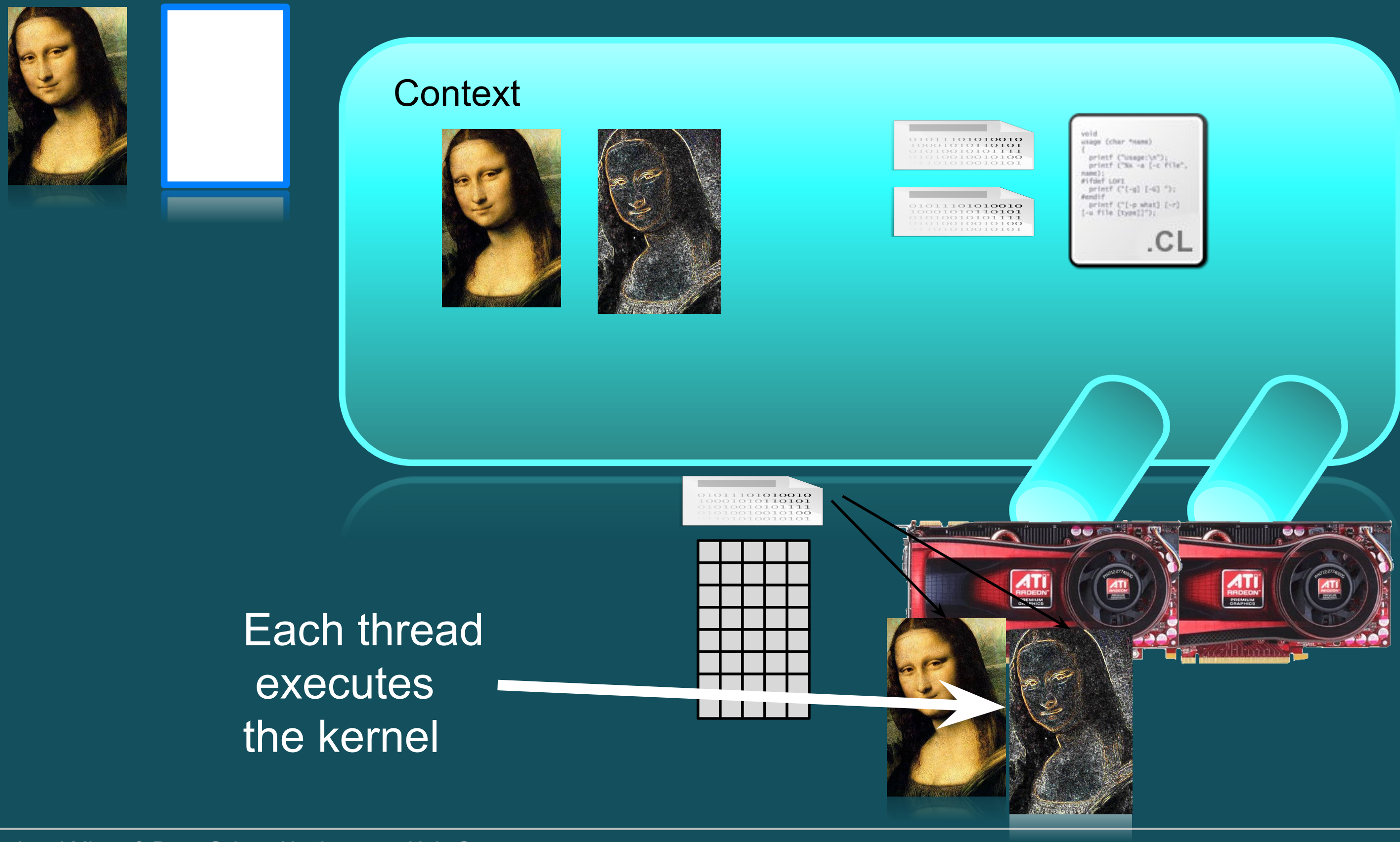
Executing the Kernel

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

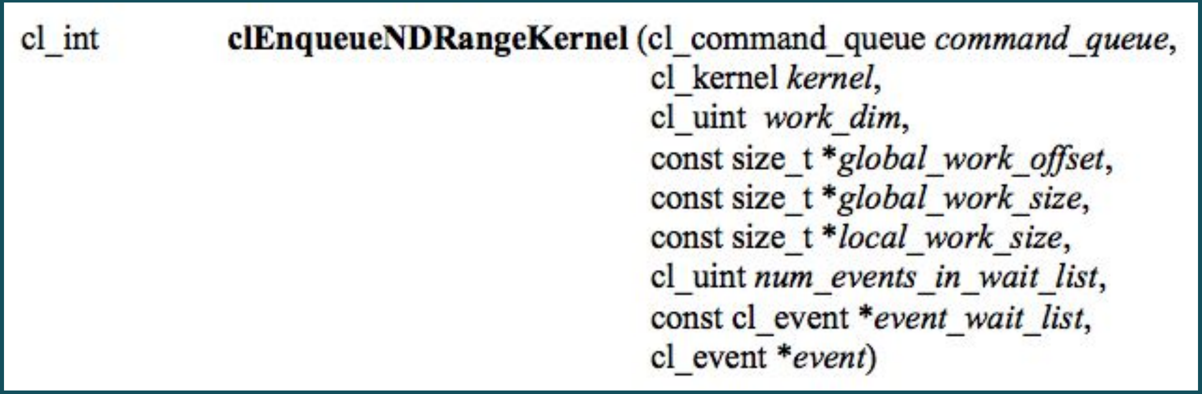

Context
.CL
An index space of threads is created (dimensions match the data)

Executing the Kernel

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



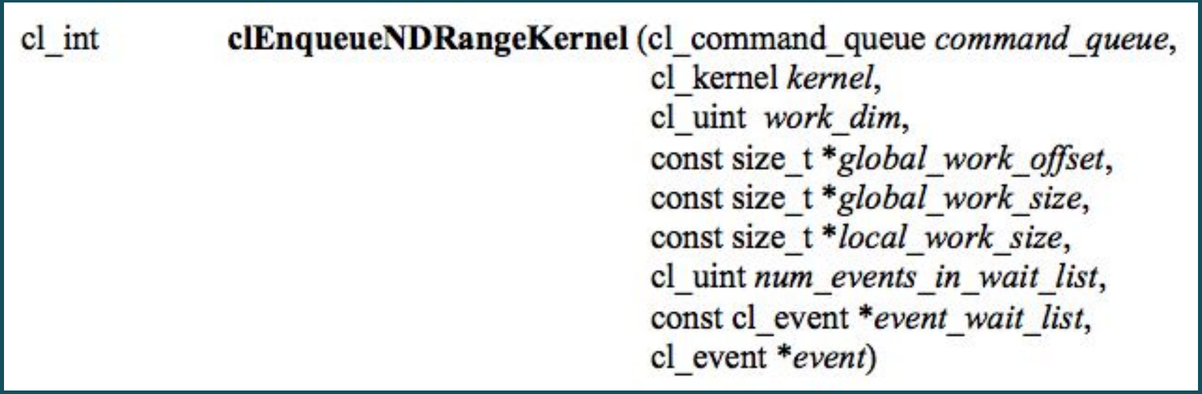
Executing the Kernel



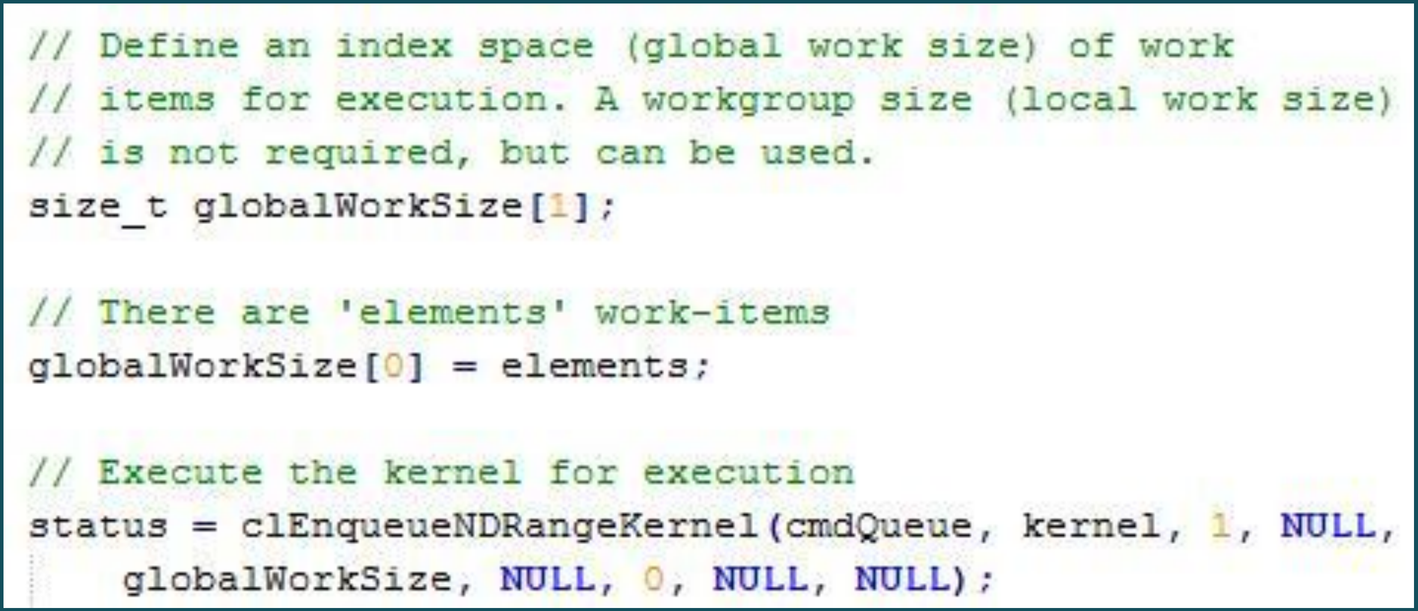
```
cl_int    clEnqueueNDRangeKernel (cl_command_queue command_queue,
                                  cl_kernel kernel,
                                  cl_uint  work_dim,
                                  const size_t *global_work_offset,
                                  const size_t *global_work_size,
                                  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



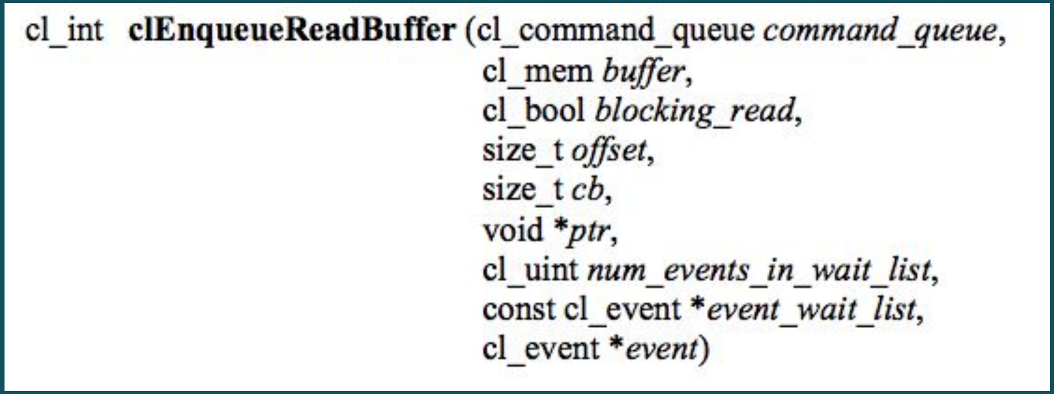
```
cl_int    clEnqueueNDRangeKernel (cl_command_queue command_queue,
                                  cl_kernel kernel,
                                  cl_uint work_dim,
                                  const size_t *global_work_offset,
                                  const size_t *global_work_size,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



```
// Define an index space (global work size) of work
// items for execution. A workgroup size (local work size)
// is not required, but can be used.
size_t globalWorkSize[1];

// There are 'elements' work-items
globalWorkSize[0] = elements;

// Execute the kernel for execution
status = clEnqueueNDRangeKernel(cmdQueue, kernel, 1, NULL,
    globalWorkSize, NULL, 0, NULL, NULL);
```

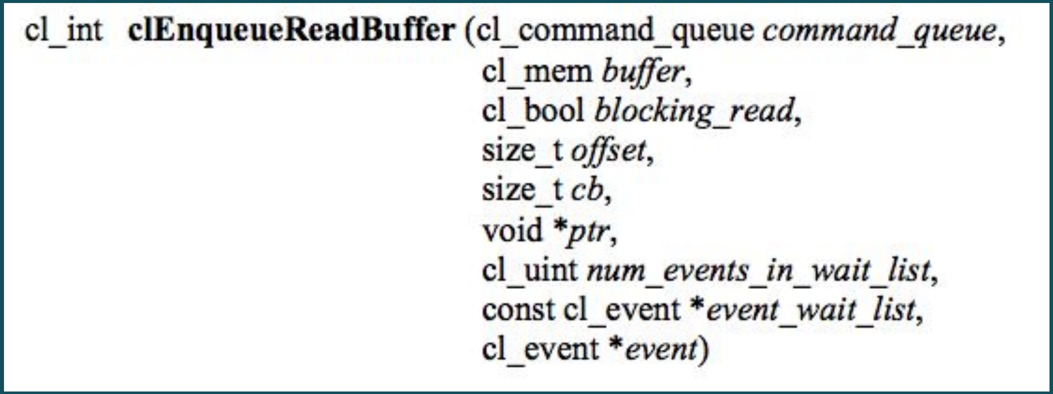
Copying Data Back

- 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

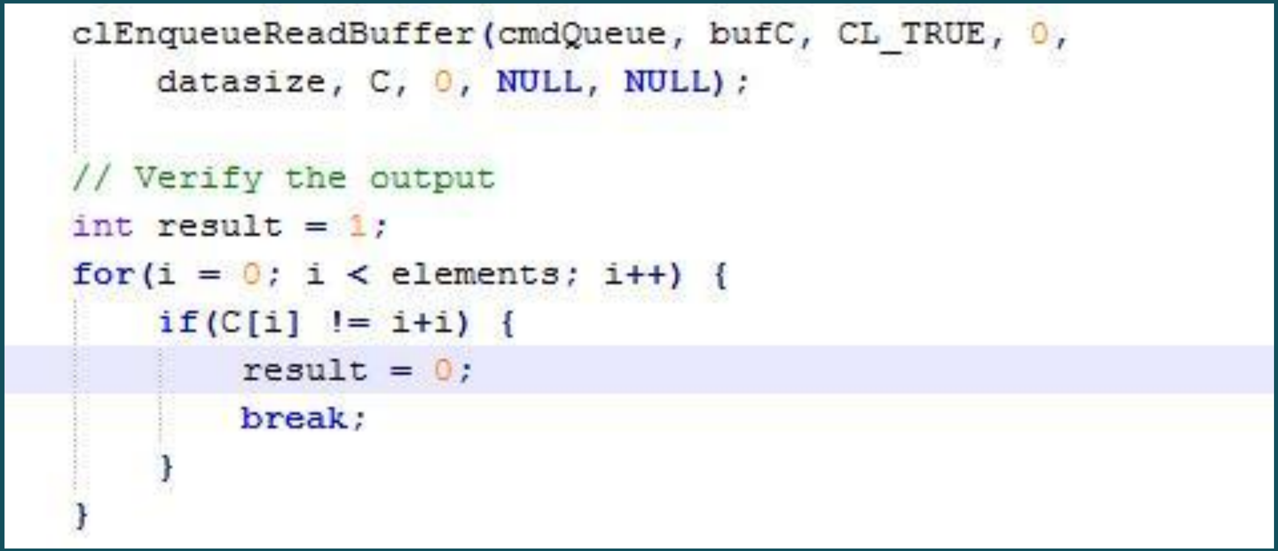


```
cl_int  clEnqueueReadBuffer (cl_command_queue command_queue,
                             cl_mem buffer,
                             cl_bool blocking_read,
                             size_t offset,
                             size_t cb,
                             void *ptr,
               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

Copying Data Back



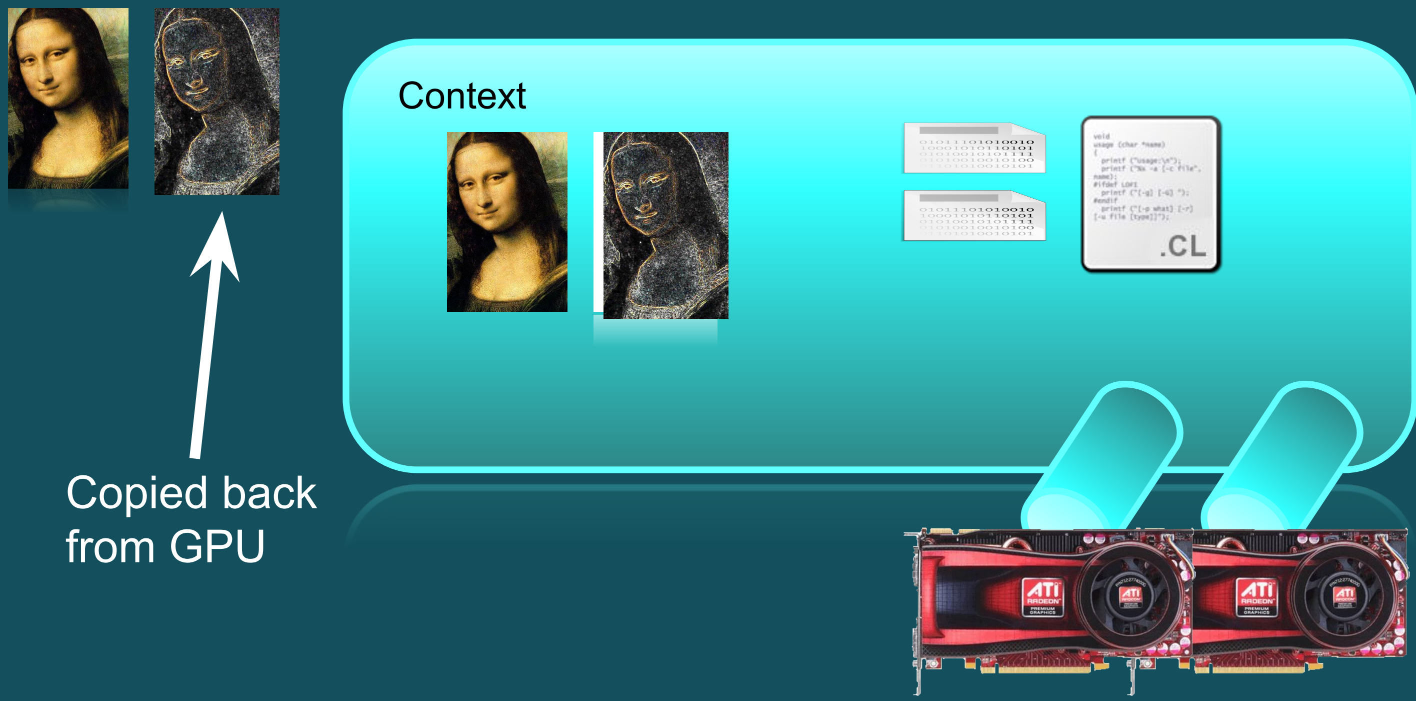
```
cl_int  clEnqueueReadBuffer (cl_command_queue command_queue,
                             cl_mem buffer,
                             cl_bool blocking_read,
                             size_t offset,
                             size_t cb,
                             void *ptr,
               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



```
clEnqueueReadBuffer(cmdQueue, bufC, CL_TRUE, 0,
    datasize, C, 0, NULL, NULL);

// Verify the output
int result = 1;
for(i = 0; i < elements; i++) {
    if(C[i] != i+i) {
        result = 0;
        break;
    }
}
```

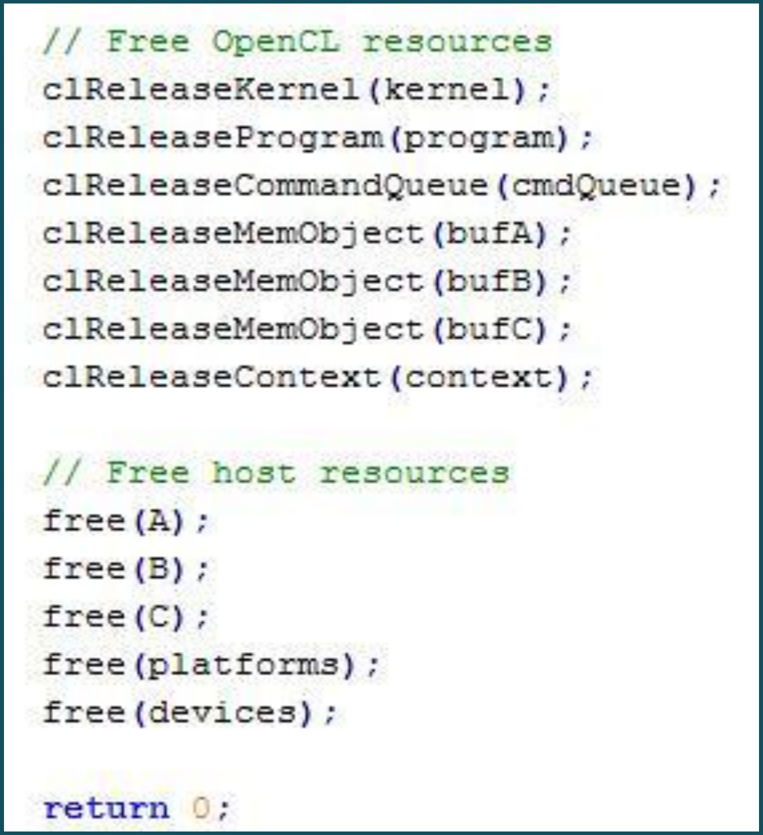
Copying Data Back

- 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()



```
// Free OpenCL resources
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(cmdQueue);
clReleaseMemObject(bufA);
clReleaseMemObject(bufB);
clReleaseMemObject(bufC);
clReleaseContext(context);

// Free host resources
free(A);
free(B);
free(C);
free(platforms);
free(devices);

return 0;
```

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

Programming Model

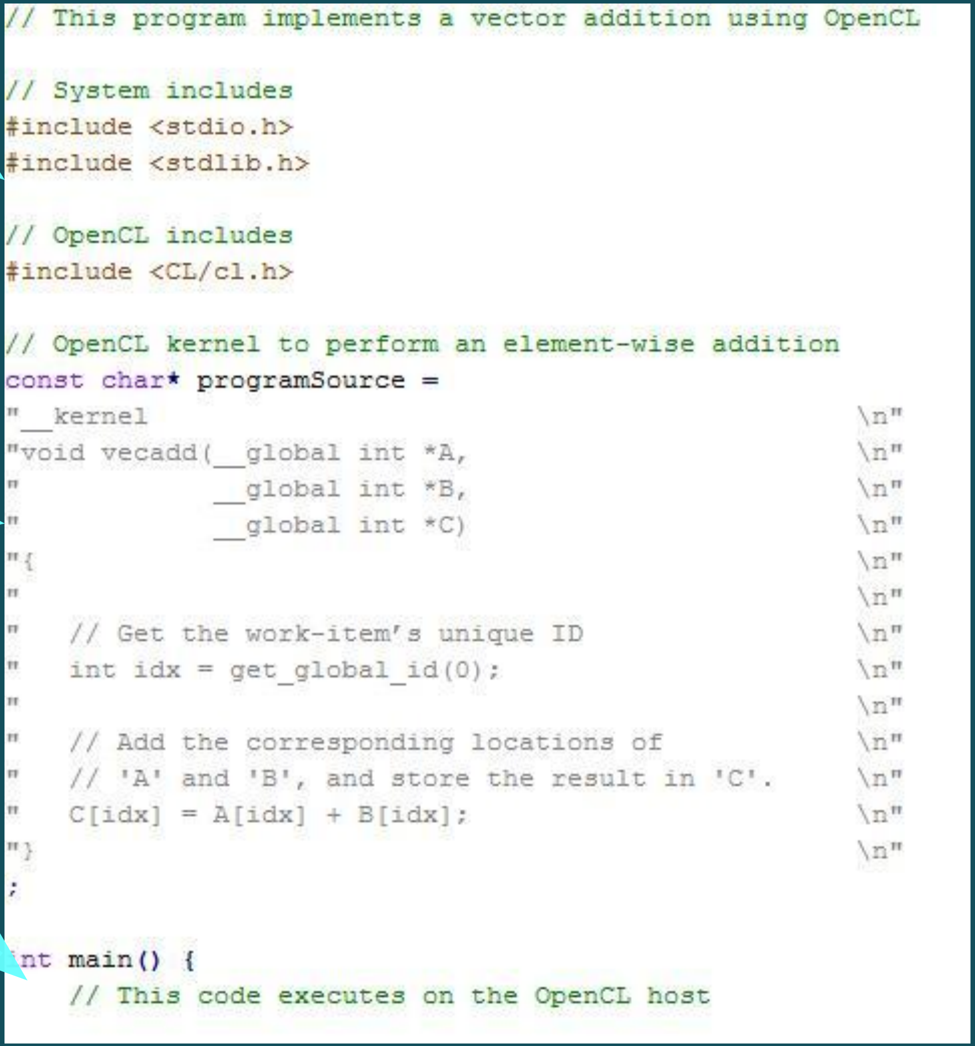
- Data parallel
  - One-to-one mapping between work-items and elements in a memory object
  - Work-groups can be defined explicitly (like CUDA) or implicitly (specify the number of work-items and OpenCL creates the work-groups)
- Task parallel
  - Kernel is executed independent of an index space
  - 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

Host Code

Headers!

Kernel in code!

Host code!



```
// This program implements a vector addition using OpenCL

// System includes
#include <stdio.h>
#include <stdlib.h>

// OpenCL includes
#include <CL/cl.h>

// OpenCL kernel to perform an element-wise addition
const char* programSource =
"__kernel                                                    \n"
"void vecadd(__global int *A,                                \n"
"            __global int *B,                                \n"
"            __global int *C)                                \n"
"{                                                           \n"
"                                                            \n"
"   // Get the work-item's unique ID                         \n"
"   int idx = get_global_id(0);                              \n"
"                                                            \n"
"   // Add the corresponding locations of                    \n"
"   // 'A' and 'B', and store the result in 'C'.             \n"
"   C[idx] = A[idx] + B[idx];                                \n"
"}                                                           \n"
;

int main() {
   // This code executes on the OpenCL host
```

Ejercicio

Utilizar las herramientas y guías presentadas en las filminas de clases para crear un primer programa OpenCL que calcule la suma elemento a elemento de dos arreglos de **2048** elementos. Finalmente, se deberá controlar en el código host si el resultado es correcto e imprimirlo en pantalla.