UNIT-4

Objectives

- OpenCL Basics
- Execution Environment
- Memory Models
- Basic OpenCL Examples

OpenCL : Introduction

- Open Computing Language
- OpenCL was refined into an initial proposal by Apple in collaboration with technical teams at AMD, IBM, Qualcomm, Intel, and NVIDIA submitted to Khronos Group in 2008.
- In November 2013, the Khronos Group announced the ratification and public release of the finalized OpenCL 2.0 specification.
- A number of additional features were added to the OpenCL:
  - shared virtual memory,
  - nested parallelism,
  - generic address spaces.
  - .

OpenCL : Introduction

- The Khronos Group, developed an general API that can run on different architectures can still achieve high performance.
- Using the core language and correctly following the specification, any program designed for one vendor can execute on another vendor’s hardware.
- OpenCL creates portable, vendor- and device-independent   programs that are capable of being accelerated on many different hardware platforms

OpenCL : Introduction

Key features of OpenCL :

- **Platform independence**: OpenCL is designed to be cross-platform, allowing developers to write code that can run on a wide range of devices without modification.
- **Heterogeneous computing**: OpenCL enables developers to explore the power of different types of hardware, such as CPUs and GPUs, to perform computations in parallel.
- **Parallel programming model**: OpenCL provides a programming model based on data parallelism, where tasks are divided into smaller work items that can be executed simultaneously on different compute units.

OpenCL : Introduction

- **Host-device interaction**: OpenCL allows seamless interaction between the host (usually the CPU) and the devices (such as GPUs or other accelerators) through its API.
- **Memory management:** OpenCL provides mechanisms for managing memory across different devices and moving data efficiently between the host and the devices.
- **Scalability**: OpenCL is designed to scale from mobile and embedded devices to high-performance computing clusters.

OpenCL SPECIFICATION

- The OpenCL specification is defined in four parts referred as models
- **Platform model:**
- Specifies that there is one host processor coordinating execution, and one or more device processors whose job it is to execute OpenCL kernels.
- It also defines an abstract hardware model for devices.

OpenCL SPECIFICATION

- **Execution model**: Defines how the OpenCL environment is configured by the host, and how the host may direct the devices to perform work.
- This includes defining an environment for execution on the host, mechanisms for host-device interaction, and a concurrency model used when configuring kernels.
- The concurrency model defines how an algorithm is decomposed into OpenCL work-items and work-groups.

OpenCL SPECIFICATION

- **Kernel programming model**: Defines how the concurrency model is mapped to physical hardware.

- **Memory model**: Defines memory object types, and the abstract memory hierarchy that kernels use.
- It also contains requirements for memory ordering and optional shared virtual memory between the host and devices

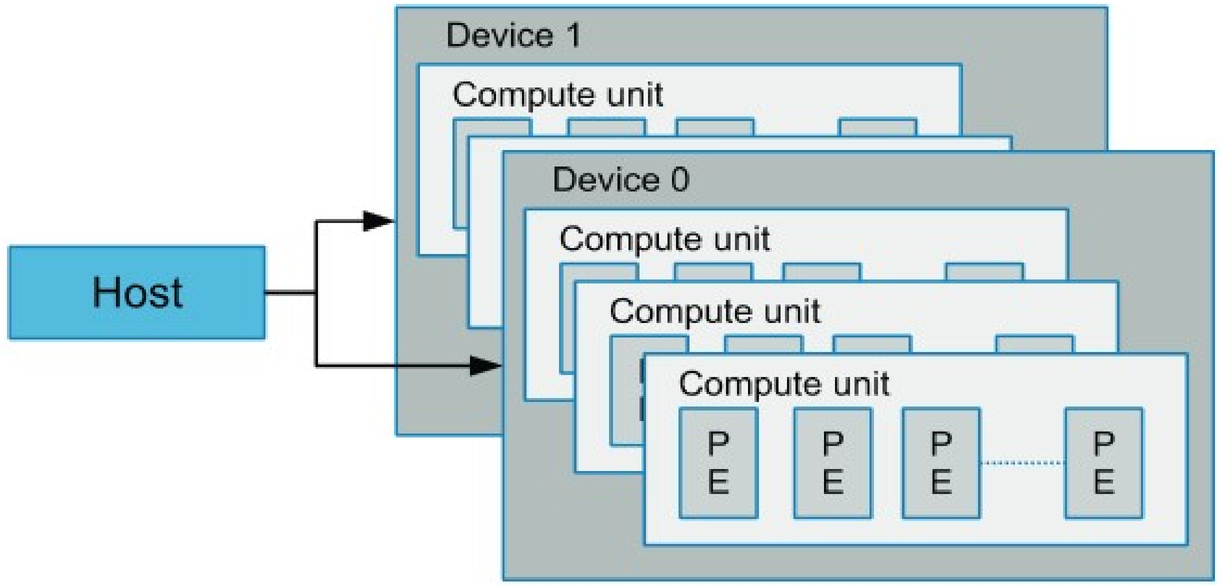
OpenCL SPECIFICATION

- OpenCL implementation executing on a **platform** consisting of a host x86 CPU using a graphics processing unit (GPU) device as an accelerator.

- The host sets up a kernel for the GPU to run and sends a command to the GPU to execute the kernel with some specified degree of parallelism. This is the **execution model**.
- The memory for the data used by the kernel is allocated by the programmer to specific parts of an abstract memory hierarchy specified by the **memory model.**
- The GPU creates hardware threads to execute the kernel, and maps them to its hardware units. This is done using the **programming Model**

OpenCL PLATFORM MODEL

OpenCL PLATFORM MODEL



An OpenCL platform with multiple compute devices

OpenCL PLATFORM MODEL

- An **OpenCL platform** consists of a **host** connected to **one or more OpenCL devices**.
- The platform model defines the roles of the host and the devices, and provides an abstract hardware model for devices.
- A device is divided into one or more compute units, which are further divided into one or more processing elements.

OpenCL PLATFORM MODEL

- The platform model is key to application
- Even within a single system, there could be a number of different OpenCL platforms

Ex: a system could have an **AMD** platform and an **Intel** platform present at the same time.

- The platform model also presents an abstract device architecture that programmers target when writing OpenCL C code.
- Vendors map this abstract architecture to the physical hardware.

OpenCL PLATFORM MODEL

- The **AMD Radeon R9 290X** graphics card (device) comprises **44 vector** processors (compute units).
- Each compute unit has **four 16-lane SIMD** engines, for a total of 64 lanes (processing elements).
- Each SIMD lane on the Radeon R9 290X executes a scalar instruction.
- This allows the GPU device to execute a total of **$44 \times 16 \times 4 = 2816$** instructions at a time.

PLATFORMS AND DEVICES

- The API call **clGetPlatformIDs()** is used to discover the set of available OpenCL platforms for a given system.
- The most robust code will call **clGetPlatformIDs()** twice when querying the system for OpenCL platforms.
- The **first call** to **clGetPlatformIDs()** passes an unsigned integer pointer as the num_platforms argument and NULL for the platforms argument.
- The pointer is populated with the available number of platforms.
- The programmer can then allocate space (pointed to by platforms)

  to hold the platform objects.

PLATFORMS AND DEVICES

- The second call to **clGetPlatformIDs(),** the platforms pointer is passed to the implementation with enough space allocated for the desired number (num_entries) of platforms.

- After platforms have been discovered, the **clGetPlatformInfo()** API call can be used to determine which implementation (vendor) the platform was defined .

PLATFORMS AND DEVICES

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

- Once a platform has been selected, the next step is to query the devices available to that platform

PLATFORMS AND DEVICES

- The call to **clGetDeviceIDs()** takes the arguments:

  platform

  device type,

  quantity of devices,

  allocation and retrieval of the desired number of devices.

PLATFORMS AND DEVICES

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

PLATFORMS AND DEVICES

- **clGetPlatformInfo()** and **clGetDeviceInfo()** print detailed information about the OpenCL-supported platforms and devices in a system.

OpenCL EXECUTION MODEL

OpenCL EXECUTION MODEL: CONTEXTS

- a **context** is an abstract environment within which coordination and memory management for kernel execution is well defined
- a **context** must be configured that enables the host to pass commands and data to the device.
- A context coordinates :

  host-device interaction,

  manages the memory objects available to the devices,

  keeps track of the programs and kernels that are created for each device.

OpenCL EXECUTION MODEL: 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
                    )
```

OpenCL EXECUTION MODEL: CONTEXTS

- The **properties** argument is used to restrict the scope of the context.
- It may provide a specific platform, enable graphics interoperability, or enable other parameters in the future.
- the devices that the programmer wants to use with the context must be supplied.

OpenCL EXECUTION MODEL:CONTEXTS

- The call **clCreateContextFromType()** allows a programmer to create a context that automatically includes all devices of the specified type (e.g. CPUs, GPUs, and all devices).

- The function **clGetContextInfo()** can be used to query information such as the number of devices present and the device objects.

- In OpenCL, the process of discovering platforms and devices and setting up a context can be tedious

OpenCL EXECUTION MODEL:COMMAND-QUEUES

- A **command-queue** is the communication mechanism that the host use to request action by a device.
- Once the host has decided which devices to work with and a context has been created, one **command-queue** needs to be created per device.
- Each **command-queue** is associated with only one device.
- the host needs to be to submit commands to a specific device when multiple devices are present in the context.
- Whenever the host needs an action to be performed by a device, it will submit commands to the proper command-queue.

OpenCL EXECUTION MODEL:COMMAND-QUEUES

- The API call **clCreateCommandQueueWithProperties()** is used to create a command-queue and associate it with a device.

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

OpenCL EXECUTION MODEL: COMMAND-QUEUES

- The properties parameter of **clCreateCommandQueueWithProperties()** is a bit field that is used to enable profiling of commands & out-of-order execution of commands

  (**CL_QUEUE_PROFILING_ENABLE**)

  (**CL_QUEUE_OUT_OF_ORDER_ EXEC_MODE_ENABLE**).

- Any API call that submits a command to a command-queue will begin with **clEnqueue** and require a command-queue as a parameter.

OpenCL EXECUTION MODEL:COMMAND-QUEUES

- The API calls **clFlush()** and clFinish() are barrier operations for a command-queue.
- The **clFinish()** call blocks execution of the host thread until all of the commands in a command-queue have completed execution; it’s functionality is synonymous with a synchronization barrier.
- The **clFlush()** call blocks execution until all of the commands in a command-queue have been removed from the queue.
- Each API call requires only the desired command-queue as an argument.

```
cl_int clFlush(cl_command_queue command_queue);
cl_int clFinish(cl_command_queue command_queue)
```

OpenCL EXECUTION MODEL:EVENTS

- objects called events are used to specify dependencies between commands.
- In addition to providing dependencies, events enable the execution status of a
- command to be queried at any time. As the event makes its way through the execution
- process, its status is updated by the implementation.

OpenCL EXECUTION MODEL:EVENTS

- The command will have one of six possible states:
- Queued: The command has been placed into a command-queue.
- Submitted: The command has been removed from the command-queue and has been submitted for execution on the device.
- Ready: The command is ready for execution on the device.
- Running: Execution of the command has started on the device.
- Ended: Execution of the command has finished on the device.
- Complete: The command and all of its child commands have finished.

OpenCL EXECUTION MODEL : EVENTS

- Querying an event’s status is done using the API call **clGetEventInfo()**

- **clWaitForEvents(),** which causes the host to wait for all events specified in the wait list to complete execution

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

OpenCL EXECUTION MODEL:DEVICE-SIDE ENQUEUING

- A kernel executing on a device now has the ability to enqueue another kernel into a device-side command-queue
- In this scenario, the kernel currently executing on a device is referred to as the parent kernel, and the kernel that is enqueued is known as the child kernel.
- Parent and child kernels execute asynchronously, although a parent kernel is not registered as complete until all its child kernels have completed.

KERNELS AND PROGRAMMING MODEL

KERNELS AND PROGRAMMING MODEL

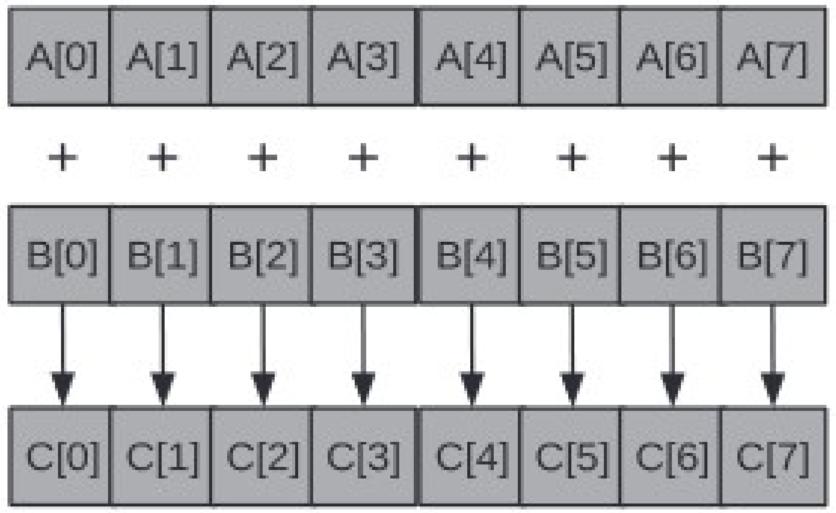
- OpenCL kernels are the parts of an OpenCL application that actually execute on a device.
- OpenCL kernel is syntactically similar to a standard C function; the key differences are a set of additional keywords and the concurrency model that OpenCL kernels implement.
- With OpenCL, the goal is often to represent parallelism programmatically at the finest granularity possible.

KERNELS AND PROGRAMMING MODEL

- Element-wise vector addition: a serial C implementation.

```
/ / P e r f o r m an e l e m e n t – w i s e a d d i t i o n o f A and B and s t o r e i n C
.
 / / T h e r e a r e N e l e m e n t s p e r a r r a y .
 v o i d v e c a d d ( i n t ∗C , i n t ∗ A, i n t ∗B , i n t N)
 {
 f o r ( i n t i = 0 ; i < N; ++ i )
 {
 C[ i ] = A[ i ] + B[ i ] ;
}
}
```

KERNELS AND PROGRAMMING MODEL



Serial vector addition

KERNELS AND PROGRAMMING MODEL

- For a simple multicore device, Writing a coarse-grained multithreaded version of the same function would require dividing the work (i.e. loop iterations) between the threads.
- Because there may be a large number of loop iterations and the work per iteration is small.
- we would need to chunk the loop iterations into a larger granularity, a technique called strip mining

KERNELS AND PROGRAMMING MODEL

```
//Perform an element−wise addition of A and B and store in C.
//There are N elements per array and NP CPU cores.
        void vecadd(int *C, int *A, int *B, int N, int NP, int tid)
        {
                int ept = N/NP; //elements per thread
                for(int i=tid*ept; i<(tid+1)*ept; ++i)
                 {
                C[i] = A[i] + B[i];
                }

         }
```

KERNELS AND PROGRAMMING MODEL

- The unit of concurrent execution in OpenCL C is a **work-item**.
- Each work-item executes the kernel function body.
- Instead of manually strip mining the loop, we will map a single iteration of the loop to a work-item.
- We can generate as many work-items as elements in the input and output arrays and allow the runtime to map those work-items to the underlying hardware
- Conceptually, this is very similar to the parallelism inherent in a functional “map” operation (cf., mapReduce) or a data-parallel for loop in OpenMP

KERNELS AND PROGRAMMING MODEL

- The call to **get_global_id(0)** allows the programmer to make use of the position of the current work-item to access a unique element in the array.
- The parameter "0" to **the get_global_id()** function assumes that we have specified a one-dimensional configuration of work-items, and therefore only need its ID in the first dimension.

KERNELS AND PROGRAMMING MODEL

```
/ / P e r f o r m an e l e m e n t − w i s e a d d i t i o n o f A and B and s t o r e i n C
 / / N work− i t e m s w i l l be c r e a t e d t o e x e c u t e t h i s k e r n e l .
 _ _ k e r n e l
 v o i d v e c a d d ( _ _ g l o b a l i n t ∗C , _ _ g l o b a l i n t ∗ A, _ _ g l o b a l i n t ∗B)
 {
        i n t t i d = g e t _ g l o b a l _ i d ( 0 ) ; / / OpenCL i n t r i n s i c f u n c t i o n
         C[ t i d ] = A[ t i d ] + B[ t i d ] ;
}
```

KERNELS AND PROGRAMMING MODEL

- When a kernel is executed, the programmer specifies the number of work-items that should be created as **an n-dimensional range** (**NDRange**).

- An **NDRange** is a one-, two-, or three- dimensional index space of **work-items** that will often map to the dimensions of either the input or the output data.

- The dimensions of the **NDRange** are specified as an N- element array of type size_t, where N represents the number of dimensions used to describe the work-items being created

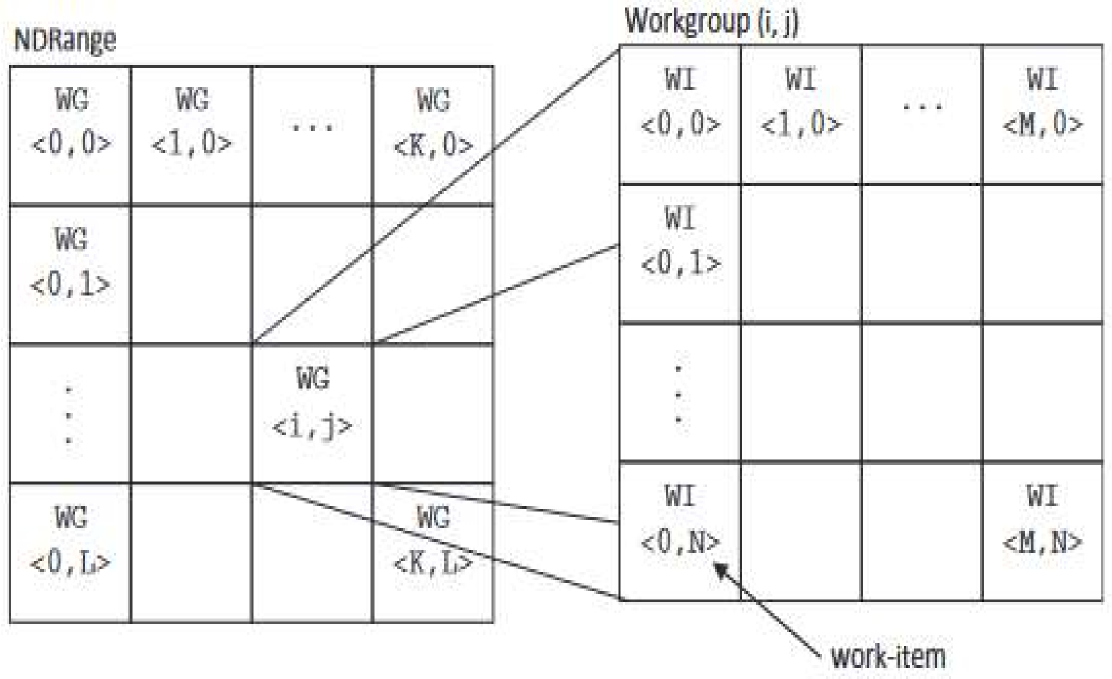
KERNELS AND PROGRAMMING MODEL

- Assume that data is one-dimensional and there are 1024 elements.
- The host code to specify a one-dimensional NDRange for 1024 elements may look like the following:

```
size_t indexSpace[3] = {1024, 1, 1};
```

- scalability comes from dividing the **work-items** of an **NDRange** into smaller, equally sized **work-groups.**
- An index space with N dimensions requires work-groups to be specified using the same N dimensions.
- A 3 dimensional index space requires three-dimensional work-groups.
- Work-items within a work-group have a special relationship with one another
- They can perform barrier operations to synchronize and they have access to a shared memory address space.

KERNELS AND PROGRAMMING MODEL



KERNELS AND PROGRAMMING MODEL

- The work-group size might be specified as

  size_t workgroupSize[3] = {64, 1, 1};

- If the total number of work-items per array is 1024 and 64 work items(elements) per group then we have 16 work-groups (1024 work-items/(64 work-items per work-group) = 16 work- groups).

o OpenCL 2.0 specification allows each dimension of the index space that is not evenly divisible by the work-group size to be divided into two regions:

  one region where the number of work-items per work-group is as specified by the programmer

  - ,another region of remainder work-groups which have fewer work-items.

COMPILATION AND ARGUMENT HANDLING

- OpenCL source code is compiled at runtime through a series of API calls.
- Runtime compilation gives the system an opportunity to optimize OpenCL kernels for a specific compute device.
- OpenCL software links only to a common runtime layer called the installable client driver (ICD).

COMPILATION AND ARGUMENT HANDLING

The process of creating a kernel from source code is as follows:

1. The OpenCL C source code is stored in a character array. If the source code is stored in a file on a disk, it must be read into memory and stored as a character array.

2. The source code is turned into a program object, cl_program, by calling

   clCreateProgramWithSource()

3.The program object is then compiled, for one or more OpenCL devices, with **clBuildProgram()**. If there are compile errors, they will be reported.

4. A kernel object, **cl_kernel**, is then created by calling **clCreateKernel** and specifying the program object and kernel name.

COMPILATION AND ARGUMENT HANDLING

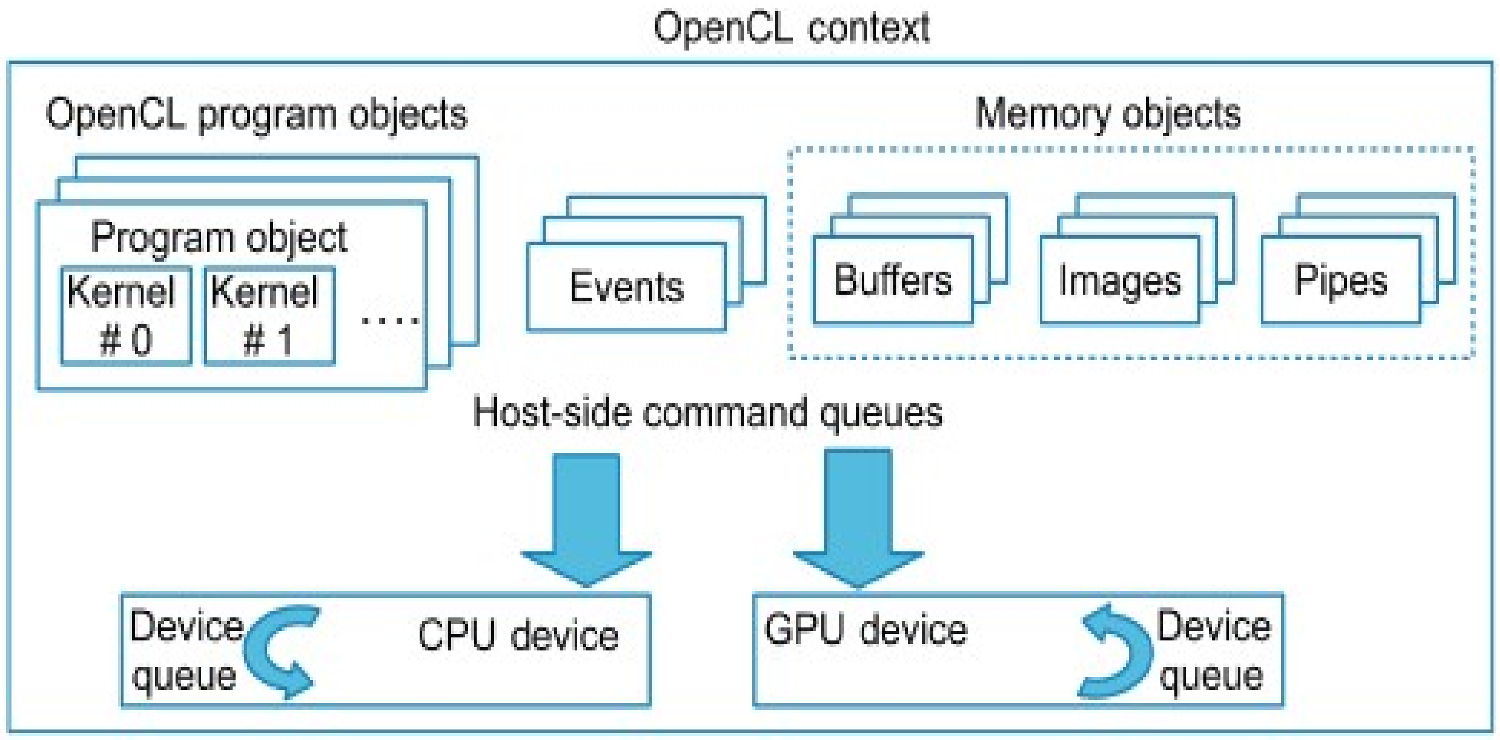
- The name of the kernel that the program exports is used to request it from the compiled program object.

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

COMPILATION AND ARGUMENT HANDLING

- The relationship between an OpenCL program and OpenCL kernels is shown in Figure below:
- multiple kernels can be extracted from an OpenCL program.
- Each context can have multiple OpenCL programs that have been generated from OpenCL source code.

COMPILATION AND ARGUMENT HANDLING



COMPILATION AND ARGUMENT HANDLING

- The OpenCL runtime shown denotes an OpenCL context with two compute devices (a CPU device and a GPU device).
- Each compute device has its own command-queues.
- Host-side and device-side command-queues are shown.
- The device-side queues are visible only from kernels executing on the compute device. The memory objects have been defined within the memory model

COMPILATION AND ARGUMENT HANDLING

- **clGetProgramInfo()** provides information about program objects.
- **CL_PROGRAM_BINARIES**, which returns a vendor-specific set of binary obje.cts generated by **clBuildProgram().**
- **clCreateProgramWithSource(),** OpenCL provides clCreateProgramWithBinary(), which takes a list of binaries that matches its device list.
- **clSetKernelArg().** This function takes:
  - a kernel object,
  - an index specifying the argument number,
  - the size of the argument,
  - and a pointer to the argument.

COMPILATION AND ARGUMENT HANDLING

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

STARTING KERNEL EXECUTION ON A DEVICE

- Enqueuing a command to a device to begin kernel execution is done with a call to clEnqueueNDRangeKernel().
- A command-queue must be specified so the target device is known.
- The kernel object identifies the code to be executed.
- Four parameters are then related to work-item creation:

  work_dim specifies the no. of dimensions in which work-items will be created.

  global_work_size specifies the no. of work-items in each dimension of the NDRange

  local_work_size specifies the no. of work-items in each dimension of the work-groups.

  global_work_offset can be used to provide an offset, so that the global IDs of the work-items do not start at zero.

STARTING KERNEL EXECUTION ON A DEVICE

- **clEnqueue** API calls, an **event_wait_list** is provided, and for non NULL values the runtime will guarantee that all corresponding events will have completed before the kernel begins execution.

OpenCL MEMORY MODEL

- The OpenCL memory model describes the structure of the memory system

exposed by an OpenCL platform to the OpenCL program.

- The memory model must define how the values in memory are seen from each of these units of execution.

MEMORY OBJECTS

- In order for data to be transferred to a device, it must first be encapsulated as a memory object.
- In order for output data to be generated, space must also be allocated and encapsulated as a memory object.
- OpenCL defines three types of memory objects:

  buffers, images, and pipes.

MEMORY OBJECTS: Buffers

- **Buffers** are equivalent to arrays in C created using **malloc(),** where data elements are stored contiguously in memory.
- The API function **clCreateBuffer()** allocates space for the buffer and returns a memory object.

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

- **clCreateBuffer()** API call is similar to **malloc** in C, or C++

MEMORY OBJECTS:Buffers

- Creating a buffer requires supplying the size of the buffer and a context in which the buffer will be allocated; it is visible for all devices associated with the context.
- Optionally, the caller can supply flags that specify that the data is read only, write only, or read-write.
- specifya host pointer with data used to initialize the buffer.

MEMORY OBJECTS : IMAGES

- Images are OpenCL memory objects that abstract the storage of physical data to allow device-specific optimizations.
- The purpose of using images is to allow the hardware to take advantage of spatial locality and to utilize the hardware acceleration available on many devices.

MEMORY OBJECTS:IMAGES

```
cl_mem
clCreateImage (
        cl_context context,
        cl_mem_flags flags,
        const cl_image_format *image_format,
        const cl_image_desc *image_desc,
        void *host_ptr,
        cl_int *errcode_ret
                )
```

MEMORY OBJECTS:IMAGES

- **cl_image_format** specifies how the image elements are stored in memory using the concept of channels.

- The channel order specifies the number of elements that make up an image element
- up to four elements, based on the traditional use of RGBA Pixels
- channel type specifies the size of each element.
- **cl_image_desc** specifies the type of the image and the dimension

MEMORY OBJECTS:PIPES

- A pipe memory object is an ordered sequence of data items referred as packets that are stored on the basis of a first in, first out (FIFO) method.
- A pipe has a write endpoint into which data items are inserted, and a read endpoint from which data items are removed.
- **clCreatePipe()** specifies the packet size along with the number of entries

  in the pipe
- **clGetPipeInfo()** can return information about the size of the pipe and

the maximum number of packets that can reside in the pipe.

MEMORY OBJECTS:PIPES

```
cl_mem
clCreatePipe (
cl_context context,
cl_mem_flags flags,
cl_uint pipe_packet_size,
cl_uint pipe_max_packets,
const cl_pipe_properties *properties,
cl_int *errcode_ret)
```

MEMORY OBJECTS:PIPES

- At any time, only one kernel may write into a pipe, and only one kernel may read from a pipe.
- To support the producer-consumer design pattern, one kernel connects to the write endpoint (the producer), while another kernel connects to the read endpoint (the consumer).
- The same kernel may not be both the writer and the reader for a pipe

DATA TRANSFER COMMANDS

- Before a kernel is executed, it is necessary to copy data from a host array into an allocated area of memory that is encapsulated as a memory object.
- The host pointer arguments within the **clCreate** calls can be used to initialize memory objects(images, buffers etc) with data from host memory.
- This allows us to initialize a memory object without the need to consider data movement any further.
- Assuming that our memory object is a buffer , data in host memory is transferred to and from a buffer using calls to **clEnqueueWriteBuffer()** and **clEnqueueReadBuffer()**

DATA TRANSFER COMMANDS

The signature for **clEnqueueWriteBuffer()** is as follows:

```
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
        )
```

DATA TRANSFER COMMANDS

clEnqueueWriteBuffer function requires:

the buffer memory object,

the number of bytes to transfer,

and an offset within the buffer.

- The combination of offset and number of bytes allows a subset of the buffer data to be written.

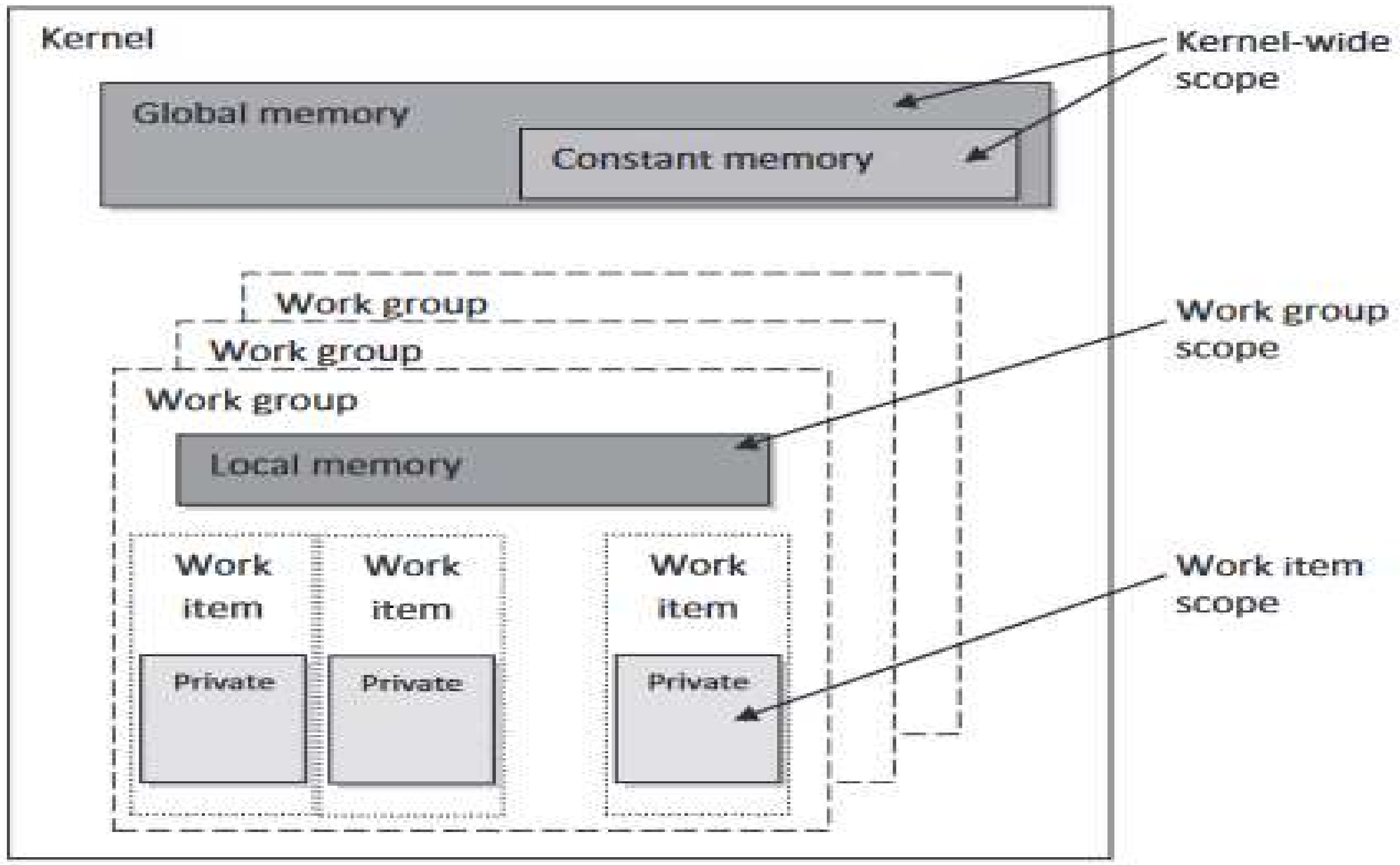
DATA TRANSFER COMMANDS

- The **blocking_write** option should be set to **CL_TRUE** if the programmer wants the transfer to complete before the function returns—effectively

  turning the otherwise asynchronous. API call into a blocking call.

- Alternatively, setting blocking_write to **CL_FALSE** will cause **clEnqueueWriteBuffer()** to return immediately (likely well before the write operation has completed).

MEMORY REGIONS



MEMORY REGIONS

- **Global memory :** is visible to all work-items executing a kernel (similarly to the main memory on a CPU-based host system).
- Whenever data is transferred from the host to the device, the data will reside in global memory.
- Any data that is to be transferred back from the device to the host must also reside in global memory.
- The keyword global or **_ _global** is added to a pointer declaration to specify that data referenced by the pointer resides in global memory.
- Example, global int* A denotes that the data pointed to by A resides in global memory

MEMORY REGIONS

- **Constant memory :** is not specifically designed for every type of read-only data, but specifically designed for data where each element is accessed simultaneously by all work-items.
- Variables whose values never change (e.g. a data variable holding the value of π) also fall into this category.
- Constant memory is modelled as a part of global memory, so memory objects that are transferred to global memory can be specified as constant.
- Data is mapped toconstant memory by using either the keyword **constant** or **_ _constant**

MEMORY REGIONS

- **Local memory** is a memory that is shared between work-items within a work-group.
- It is common for local memory to be mapped to on-chip memory,

such as software-managed scratchpad memory.

- Calling **clSetKernelArg()** with a size, but no argument, allows local memory to be allocated at runtime.
- Within an OpenCL C kernel, a kernel parameter that corresponds to local memory is defined as a **local** or **_ _local** pointer

THE OpenCL RUNTIME WITH AN EXAMPLE

STEPS

- The main steps to execute a simple OpenCL application are as follows:

1.Discovering the platform and devices

2. Creating a context

3. Creating a command-queue per device

4. Creating memory objects (buffers) to hold data

5. Copying the input data onto the device

6. Creating and compiling a program from the OpenCL C source code

7. Extracting the kernel from the program

8. Executing the kernel

9. Copying output data back to the host

10. Releasing the OpenCL resource

Discovering the platform and devices

- Before a host can request that a kernel be executed on a device, a platform and a device or devices must be discovered.

```
        cl_int status;    // Used for error checking
// 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);
// Retrieve the number of devices
        cl_uint numDevices = 0;
```

Discovering the platform and devices

```
        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);
```

- In the complete program listing that follows, we will assume that we are using the first platform and device that are found, which will allow us to reduce the number of function calls required

Creating a context

- Once the device or devices have been discovered, thecontext can be configured on the host.

// Create a context that includes all devices

```
        cl_context context = clCreateContext(NULL, numDevices, devices,
NULL, NULL,                                                  &status);
```

Creating a command-queue per device

- Once the host has decided which devices to work with and a context has been created, one command-queue needs to be created per device .

```
// Only create a command-queue for the first device
        cl_command_queue cmdQueue =
        clCreateCommandQueueWithProperties(context, devices[0], 0,
                                           &status);
```

Creating buffers to hold data

- Creating a buffer requires supplying the size of the buffer and a context in which the buffer will be allocated; it is visible to all devices associated with the context.
- Optionally, the caller can supply flags that specify that the data is read only, write only, or read-write.
- By passing NULL as the fourth argument, we are not initializing the buffer at this step.

Creating buffers to hold data

```
// Allocate 2 input and one output buffer for the three vectors in the vector addition
        cl_mem bufA = clCreateBuffer(context, CL_MEM_READ_ONLY, datasize,
                                           NULL, &status);
        cl_mem bufB = clCreateBuffer(context, CL_MEM_READ_ONLY, datasize,
                                           NULL, &status);
        cl_mem bufC = clCreateBuffer(context, CL_MEM_WRITE_ONLY, datasize,
                                           NULL, &status);
```

Copying the input data onto the device

- The next step is to copy data from a host pointer to a buffer.
- The API call takes a command-queue argument, so data will likely be copied directly to the device.
- By setting the third argument to CL_TRUE, we can ensure that data is copied before the API call returns.

```
// Write data from the input arrays to the buffers
        status = clEnqueueWriteBuffer(cmdQueue, bufA, CL_TRUE, 0,
                                      datasize, A, 0, NULL, NULL);
        status = clEnqueueWriteBuffer(cmdQueue, bufB, CL_TRUE, 0,
                                      datasize, B, 0, NULL, NULL);
```

Creating and compiling a program

- vector addition kernel is stored in a character array, programSource, and is used to create a program object which is then compiled.

```
// Create a program with source code
        cl_program program = clCreateProgramWithSource(context, 1,
                        (const char**)&programSource, NULL, &status);
// Build (compile) the program for the device
        status = clBuildProgram(program, numDevices, devices, NULL,
                                NULL, NULL);
```

Extracting the kernel from the program

- The kernel is created by selecting the desired function from within the program.

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

Executing the kernel

- Once the kernel has been created and data has been initialized, the buffers are set as arguments to the kernel.
- A command to execute the kernel can now be enqueued into the command queue.
- The kernel, the command requires specification of the NDRange configuration.

Executing the kernel

```
// Set the kernel arguments
        status = clSetKernelArg(kernel, 0, sizeof(cl_mem), &bufA);
        status = clSetKernelArg(kernel, 1, sizeof(cl_mem), &bufB);
        status = clSetKernelArg(kernel, 2, sizeof(cl_mem), &bufC);
// Define an index space of work-items for execution.
// A work-group size is not required, but can be used.
        size_t indexSpaceSize[1], workGroupSize[1];
        indexSpaceSize[0] = datasize/sizeof(int);
        workGroupSize[0] = 256;
// Execute the kernel for execution
        status = clEnqueueNDRangeKernel(cmdQueue, kernel, 1, NULL,
        indexSpaceSize, workGroupSize, 0, NULL, NULL);
```

Copying output data back to the host

- This step reads data back to a pointer on the host.

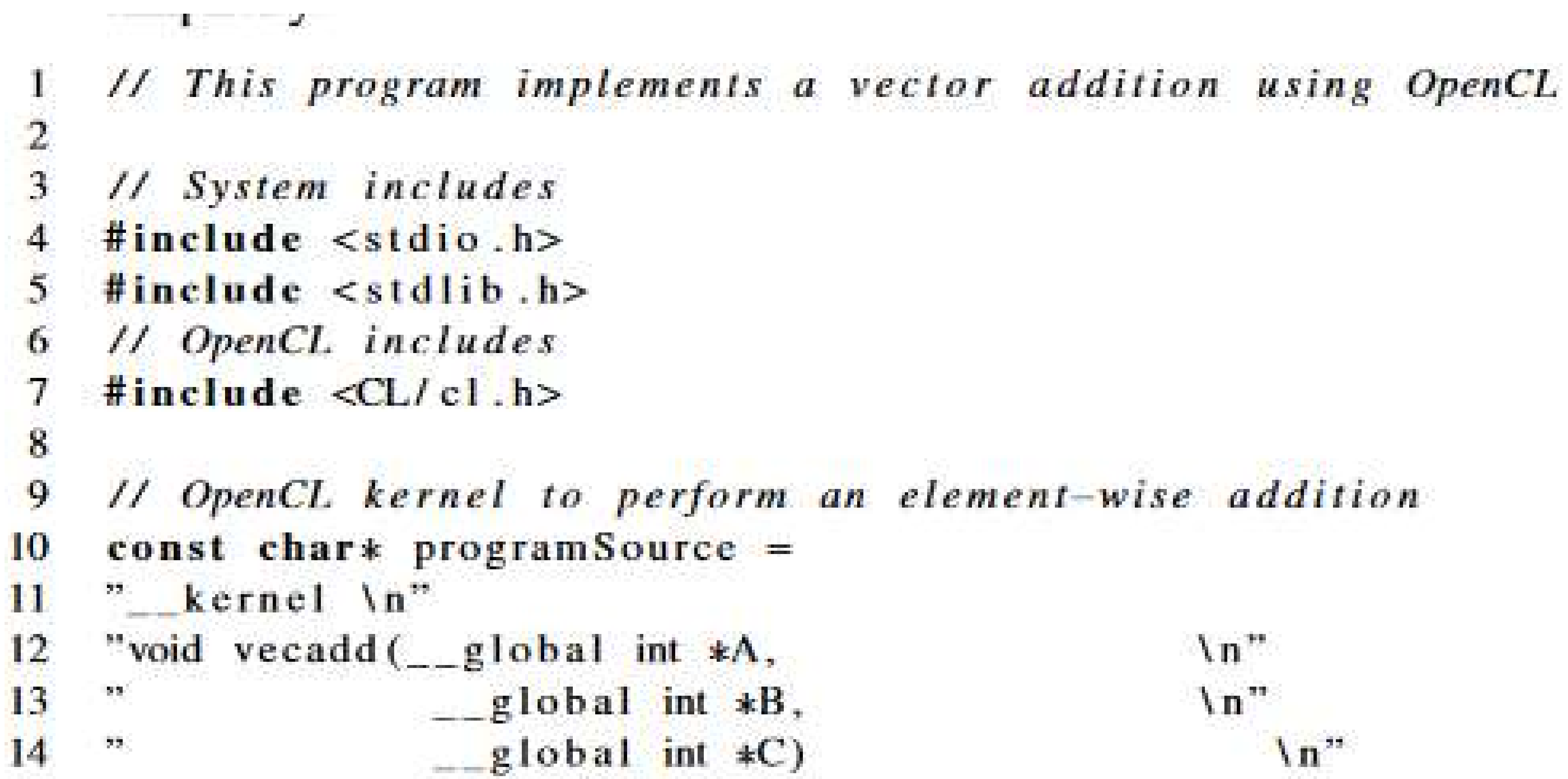
```
// Read the device output buffer to the host output array
        status = clEnqueueReadBuffer(cmdQueue, bufC, CL_TRUE, 0,
                 datasize, C, 0, NULL, NULL);
```

Releasing resources

- Once the kernel has completed execution and the resulting output has been retrieved from the device, the OpenCL resources that were allocated can be freed.

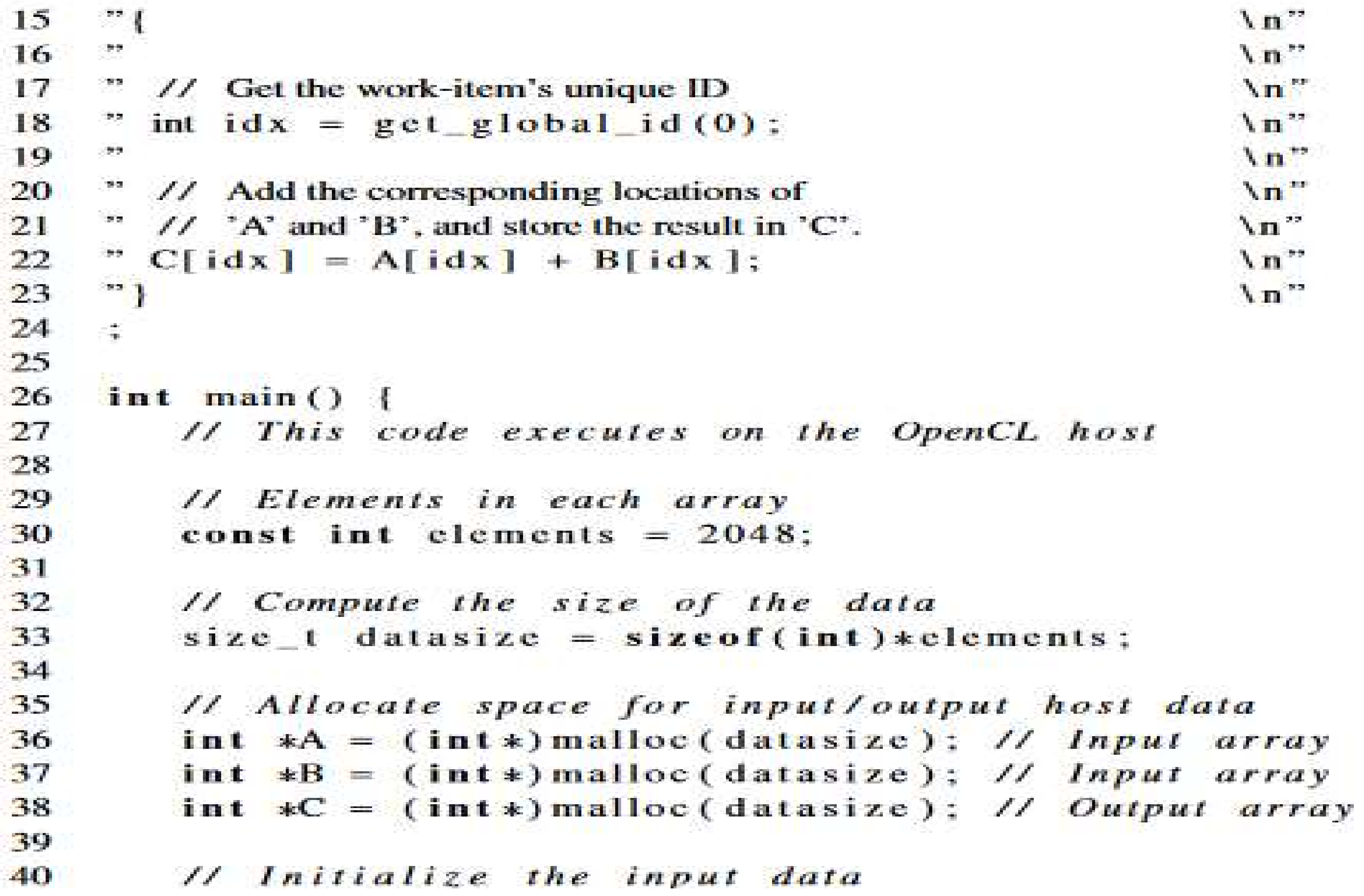
```
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(cmdQueue);
clReleaseMemObject(bufA);
clReleaseMemObject(bufB);
clReleaseMemObject(bufC);
clReleaseContext(context);
```

COMPLETE VECTOR ADDITION LISTING



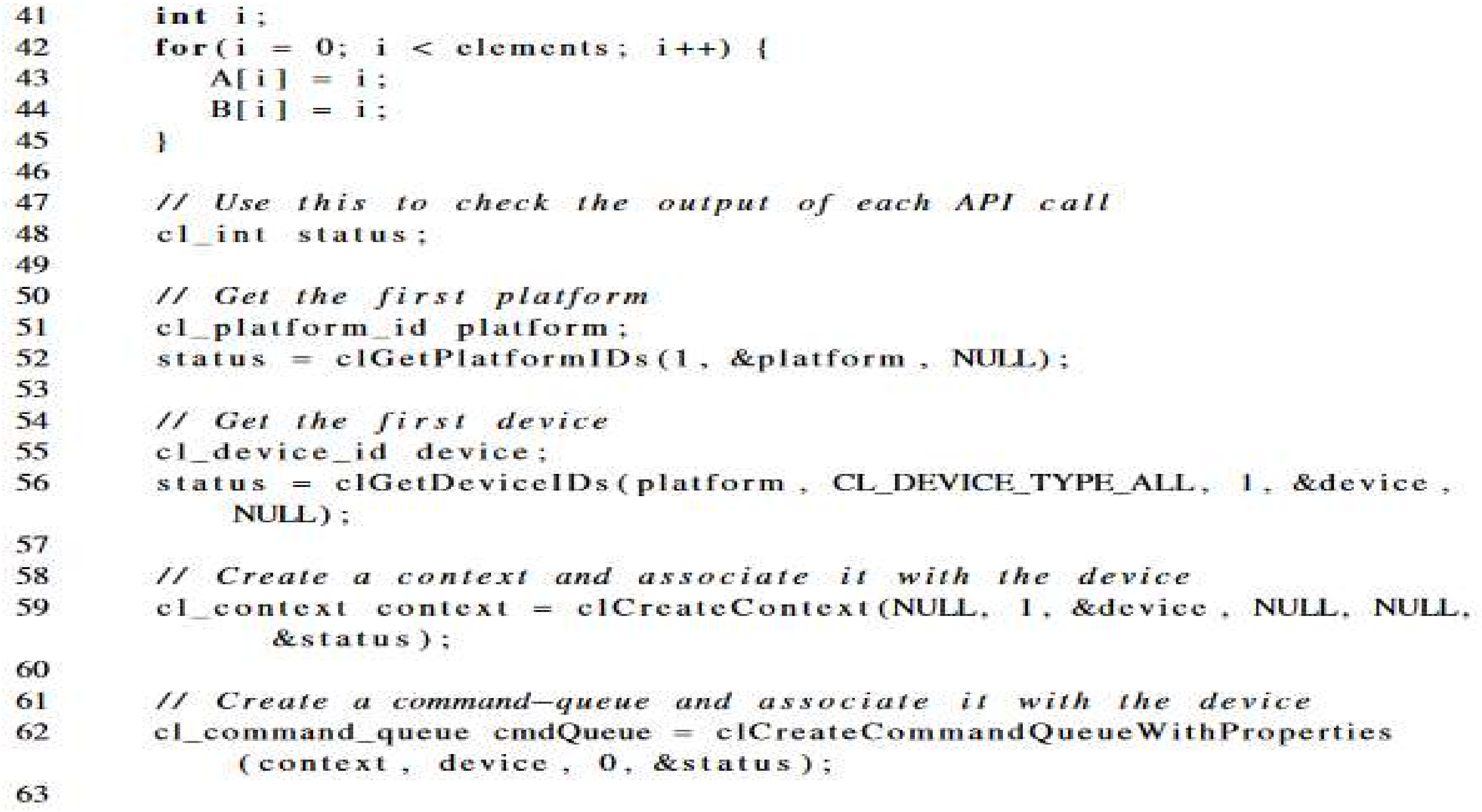
```
// This program implements a vector addition using OpenCL

// 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"
```

COMPLETE VECTOR ADDITION LISTING



```
"{                                                    \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

    //  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  host  data
    int  *A  =  (int*)malloc(datasize);  //  Input  array
    int  *B  =  (int*)malloc(datasize);  //  Input  array
    int  *C  =  (int*)malloc(datasize);  //  Output  array

    //  Initialize  the  input  data
```

COMPLETE VECTOR ADDITION LISTING

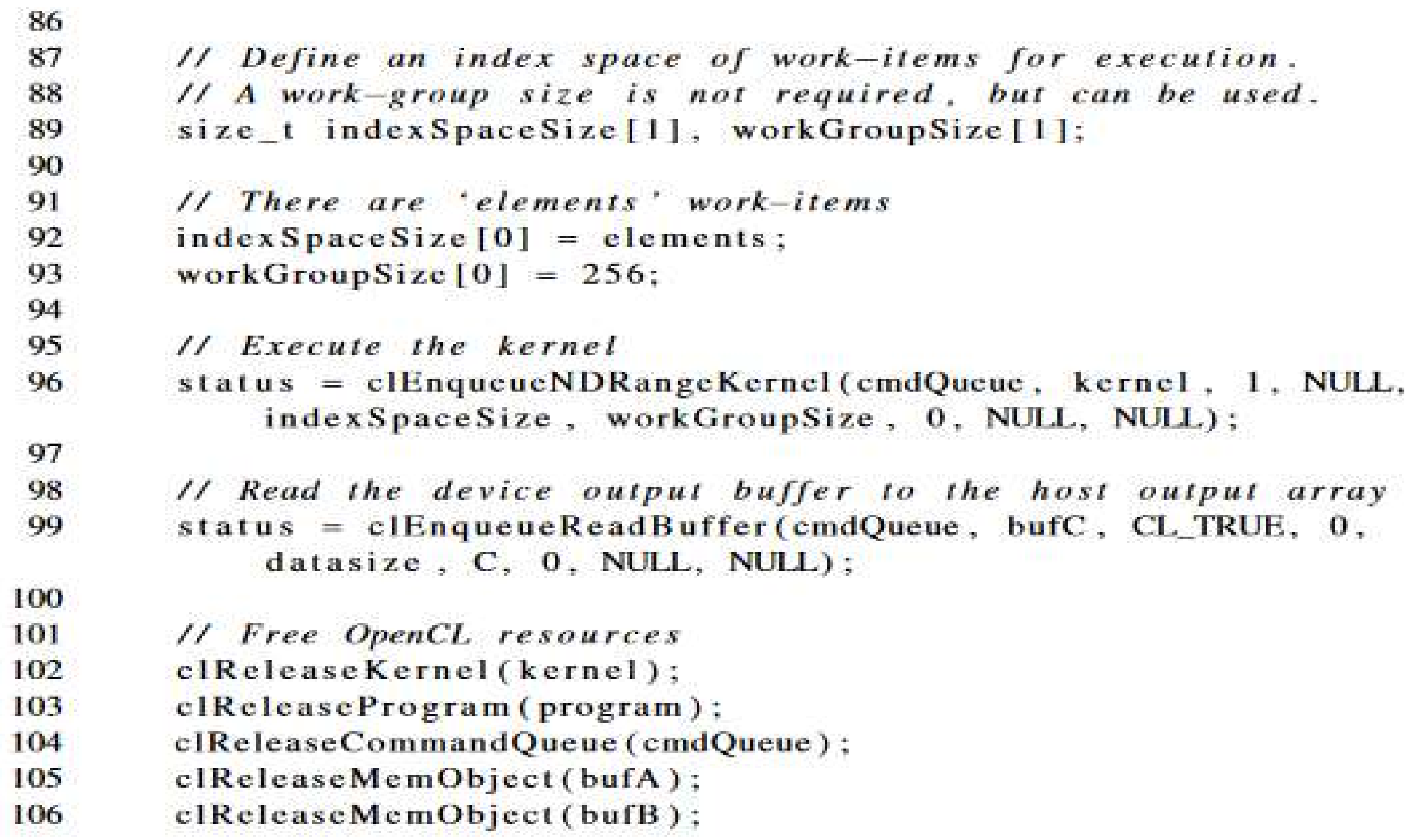


```
    int i;
    for(i = 0; i < elements; i++) {
        A[i] = i;
        B[i] = i;
    }

    // Use this to check the output of each API call
    cl_int status;

    // Get the first platform
    cl_platform_id platform;
    status = clGetPlatformIDs(1, &platform, NULL);

    // Get the first device
    cl_device_id device;
    status = clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 1, &device,
         NULL);

    // Create a context and associate it with the device
    cl_context context = clCreateContext(NULL, 1, &device, NULL, NULL,
            &status);

    // Create a command-queue and associate it with the device
    cl_command_queue cmdQueue = clCreateCommandQueueWithProperties
         (context, device, 0, &status);

```

COMPLETE VECTOR ADDITION LISTING

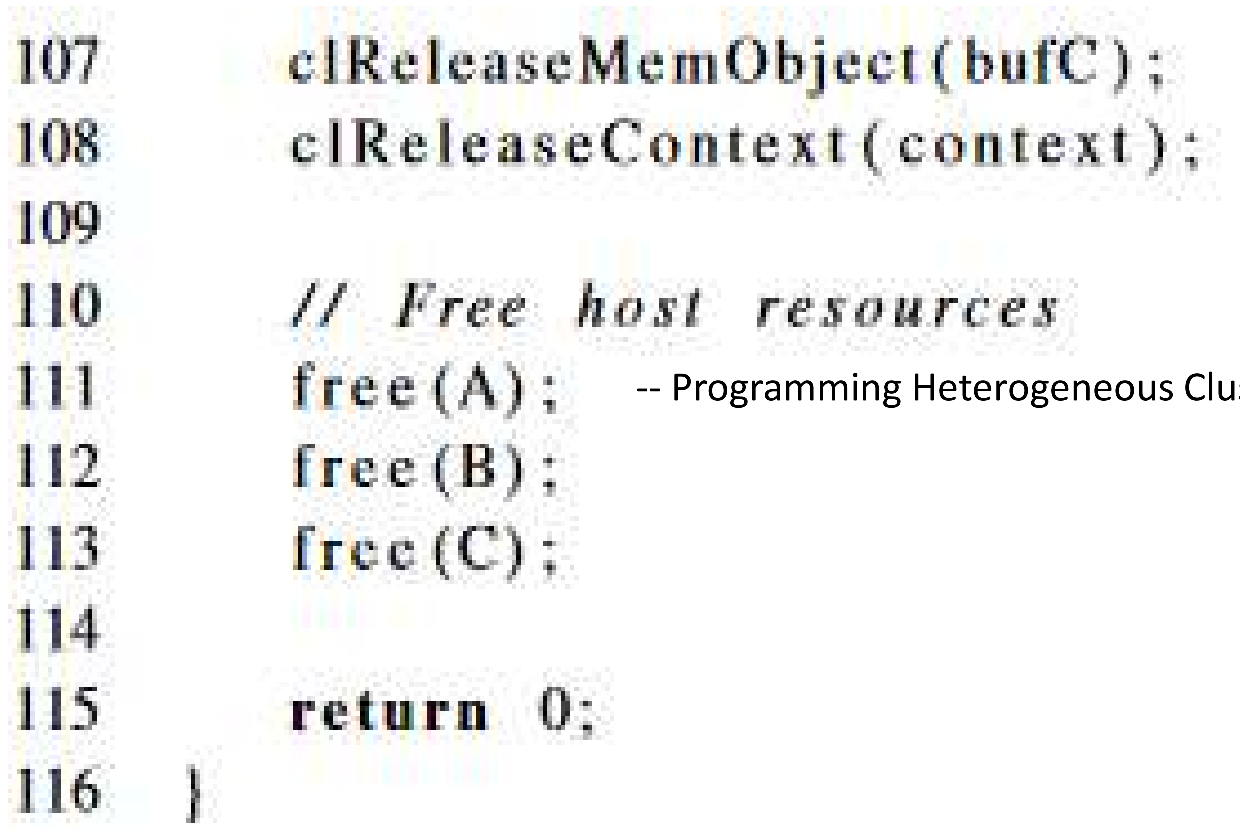
```
// Allocate two input buffers and one output buffer for the three vectors in the vector addition
cl_mem bufA = clCreateBuffer(context, CL_MEM_READ_ONLY, datasize, NULL, &status);
cl_mem bufB = clCreateBuffer(context, CL_MEM_READ_ONLY, datasize, NULL, &status);
cl_mem bufC = clCreateBuffer(context, CL_MEM_WRITE_ONLY, datasize, NULL, &status);

// // Write data from the input arrays to the buffers
status = clEnqueueWriteBuffer(cmdQueue, bufA, CL_FALSE, 0, datasize, A, 0, NULL, NULL);
status = clEnqueueWriteBuffer(cmdQueue, bufB, CL_FALSE, 0, datasize, B, 0, NULL, NULL);

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

// Build (compile) the program for the device
status = clBuildProgram(program, 1, &device, NULL, NULL, NULL);

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

// Set the kernel arguments
status = clSetKernelArg(kernel, 0, sizeof(cl_mem), &bufA);
status = clSetKernelArg(kernel, 1, sizeof(cl_mem), &bufB);
status = clSetKernelArg(kernel, 2, sizeof(cl_mem), &bufC);
```

COMPLETE VECTOR ADDITION LISTING



```
// Define an index space of work-items for execution.
// A work-group size is not required, but can be used.
size_t indexSpaceSize[1], workGroupSize[1];

// There are 'elements' work-items
indexSpaceSize[0] = elements;
workGroupSize[0] = 256;

// Execute the kernel
status = clEnqueueNDRangeKernel(cmdQueue, kernel, 1, NULL,
     indexSpaceSize, workGroupSize, 0, NULL, NULL);

// Read the device output buffer to the host output array
status = clEnqueueReadBuffer(cmdQueue, bufC, CL_TRUE, 0,
     datasize, C, 0, NULL, NULL);

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

COMPLETE VECTOR ADDITION LISTING



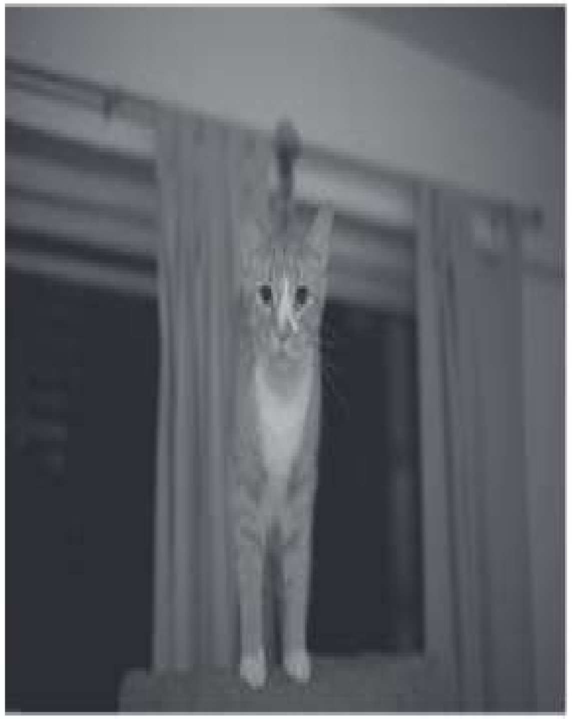
```
    clReleaseMemObject(bufC);
    clReleaseContext(context);

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

    return 0;
}
```

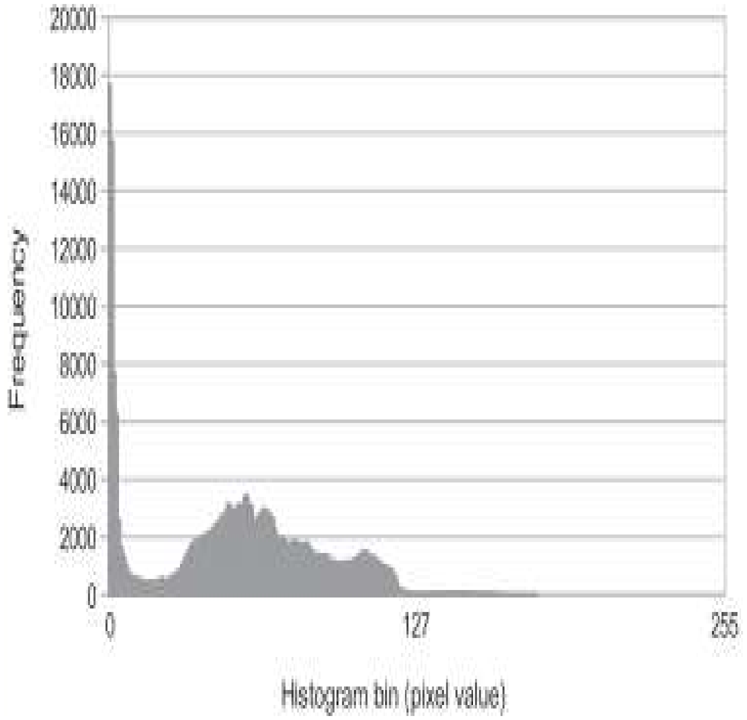
-- Programming Heterogeneous Cluster.

HISTOGRAM

- A histogram is used to count or visualize the frequency of data (i.e. the number of occurrences) over units of discrete intervals, called **bins**.
- Histograms have many applications within data and image processing.
- In this example, we will create a histogram of the frequency of pixel values within a 256-bit image.

HISTOGRAM





20000
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
Frequency
0
127
255
Histogram bin (pixel value)

HISTOGRAM

- Conceptually, the histogram algorithm itself is very simple. In the case where each value corresponds to a bin, a histogram could be computed as follows:

```
int histogram[HIST_BINS]
main( ) {
for (each input value) {
histogram[value]++
}
}
```

HISTOGRAM

- The pseudocode below, which could be used to run a multithreaded version of a histogram computation.

```
int histogram[HIST_BINS]
createHistogram( ) {
int localHistogram[HIST_BINS]
for (each of my values) {
localHistogram[value]++
}
for (each bin) {
atomic_add(histogram[bin], localHistogram[bin])
}
}
```

HISTOGRAM

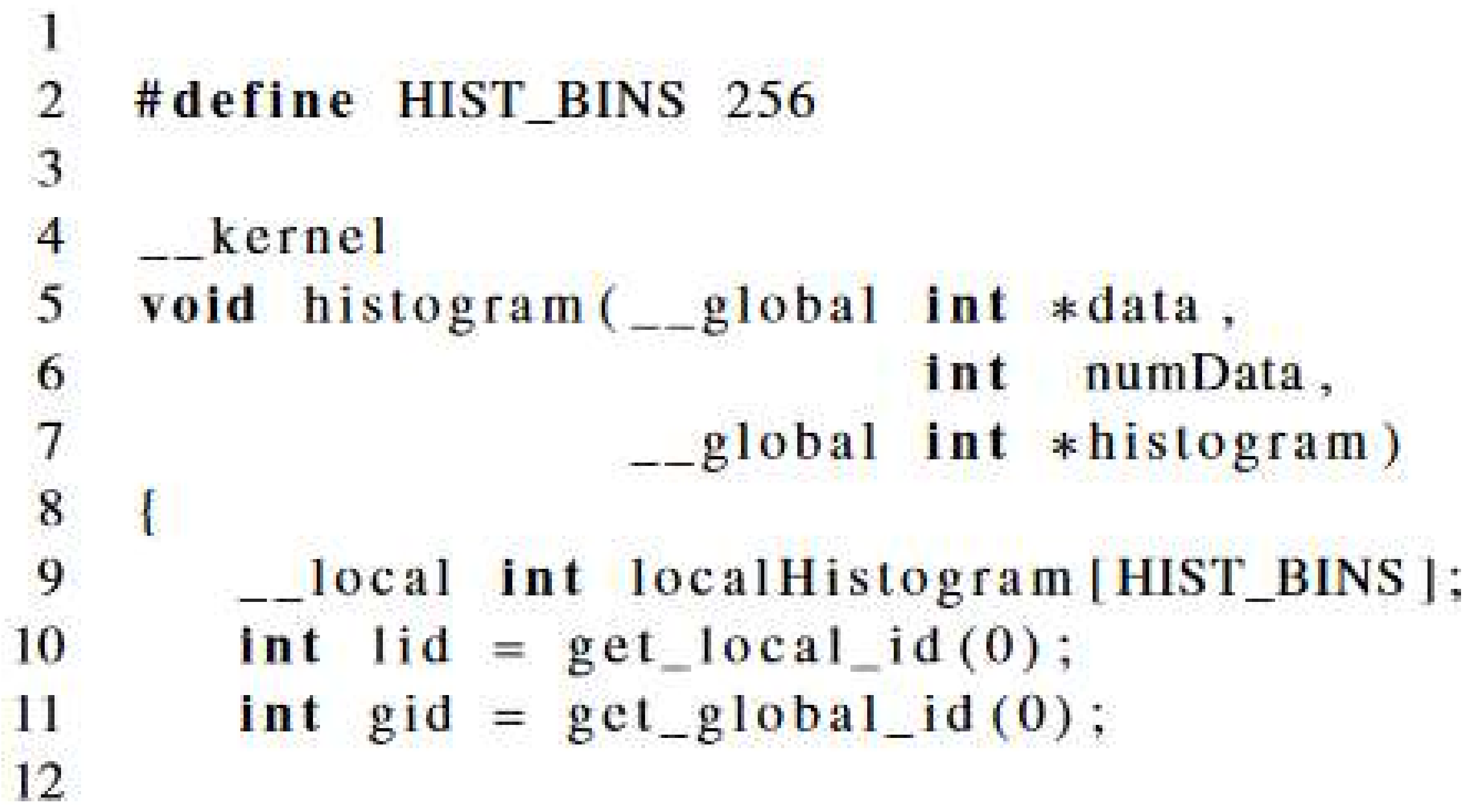
```
main( ) {
for (number of threads) {
spawn_thread(createHistogram)
}
}
```

HISTOGRAM

- The implementation of the histogram algorithm has five steps:

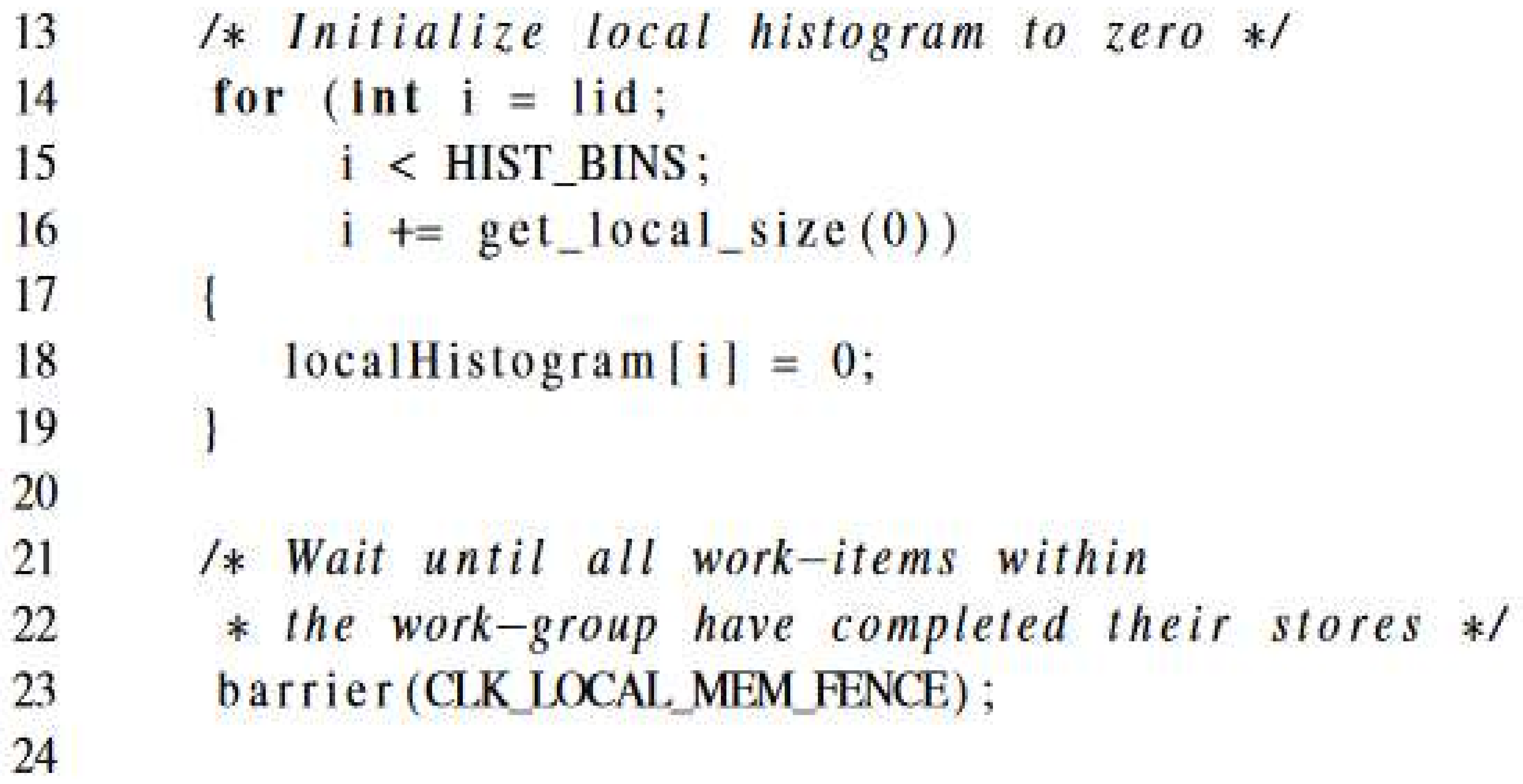
1. Initialize the local histogram bins to zero (Line 14).
2. Synchronize to ensure that all updates have completed (Line 23).
3. Compute the local histogram (Line 26).
4. Synchronize again to ensure that all updates have completed (Line 35).
5. Write the local histogram out to global memory (Line 39).

HISTOGRAM



```

#define HIST_BINS 256

__kernel
void histogram(__global int *data,
                        int  numData,
               __global int *histogram)
{
    __local int localHistogram[HIST_BINS];
    int lid = get_local_id(0);
    int gid = get_global_id(0);

```

HISTOGRAM

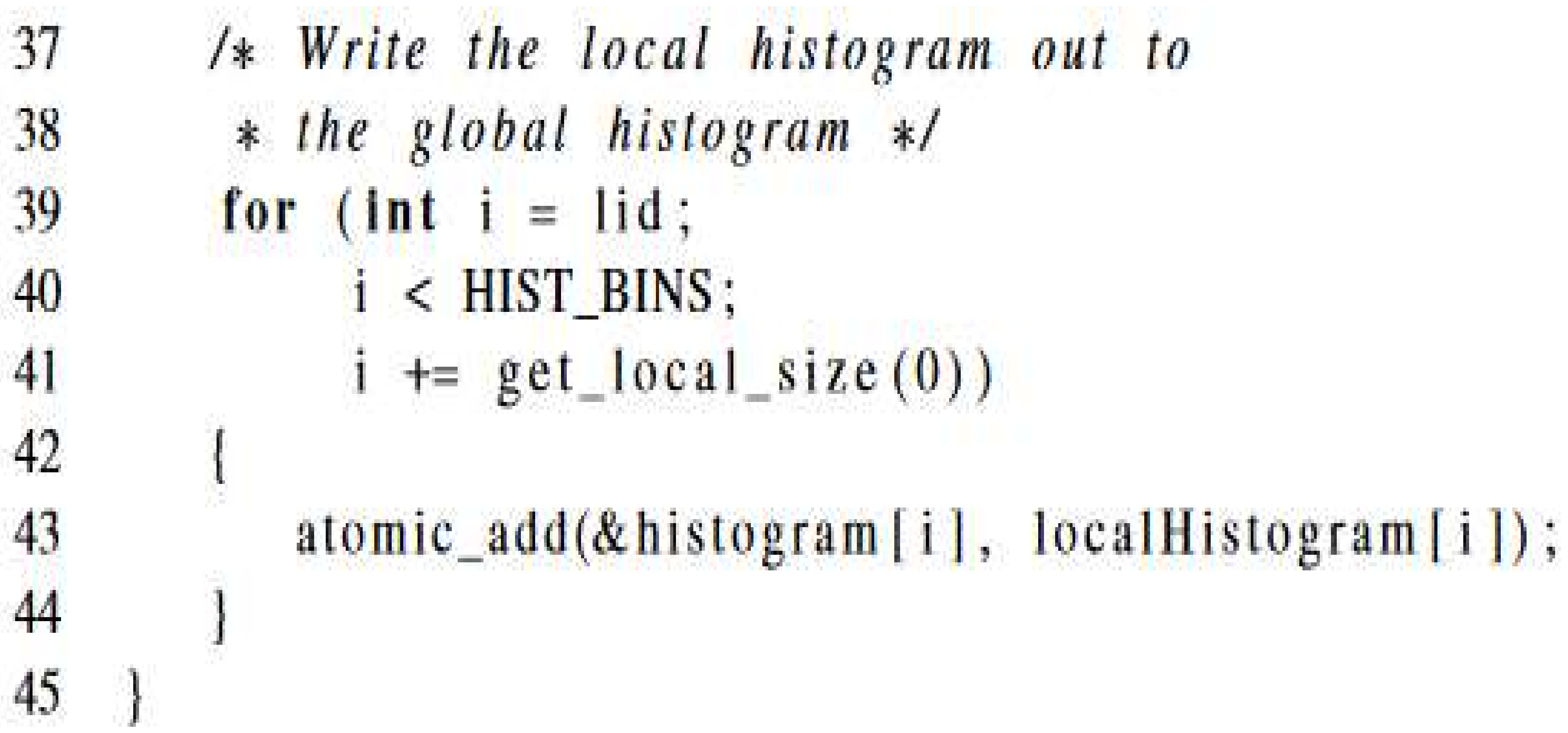


```
/* Initialize local histogram to zero */
 for (int i = lid;
        i < HIST_BINS;
        i += get_local_size(0))
{
    localHistogram[i] = 0;
}

/* Wait until all work-items within
 * the work-group have completed their stores */
 barrier(CLK_LOCAL_MEM_FENCE);

```

HISTOGRAM

```
/* Compute local histogram */
for (int i = gid;
        i < numData;
        i += get_global_size(0))
{
    atomic_add(&localHistogram[data[i]], 1);
}

/* Wait until all work-items within
 * the work-group have completed their stores */
barrier(CLK_LOCAL_MEM_FENCE);

```

HISTOGRAM



```
    /* Write the local histogram out to
     * the global histogram */
    for (int i = lid;
            i < HIST_BINS;
            i += get_local_size(0))
    {
        atomic_add(&histogram[i], localHistogram[i]);
    }
}
```

HISTOGRAM

- In step 1, we stride by the work-group size to initialize each value in the local histogram to zero.
- This allows code to change work-group sizes, potentially as a performance optimization, and still remain functionally correct.
- Step 3 uses the same technique to read from global memory.
- Steps 2 and 4 use a barrier to synchronize between steps, and specify a memory fence to ensure that all work-items in the work-group have the same view of memory before proceeding.
- step 5 again uses the technique to write out of local memory

HISTOGRAM

- Steps 1, 3, and 5 illustrate a common OpenCL technique for reading data from, or writing data to a shared location (global or local memory).
- When we need each location to be accessed by only a single work-item, we can begin with a work-item’s unique ID and stride by the size of the total number of work-items (i.e. the workgroup size when accessing local memory, or the NDRange size when accessing global
- memory).

IMAGE ROTATION

- Rotation is a common image processing routine with applications in matching, alignment, and other image-based algorithms.
- The input to an image rotation routine is an image, the rotation angle θ, and a point about which rotation is done.
- The coordinates of a point (x, y) when rotated by an angle θ around (x0 , y0)
- become (x′, y′), as shown by the following equations.

$$x' = \cos\theta(x - x0) + \sin\theta(y - y0)$$

$$y' = -\sin\theta(x - x0) + \cos\theta(y - y0)$$

- From the equations, it is clear that the pixel that will be stored in each output location (x′, y′) can be computed independently.

IMAGE ROTATION

Original image

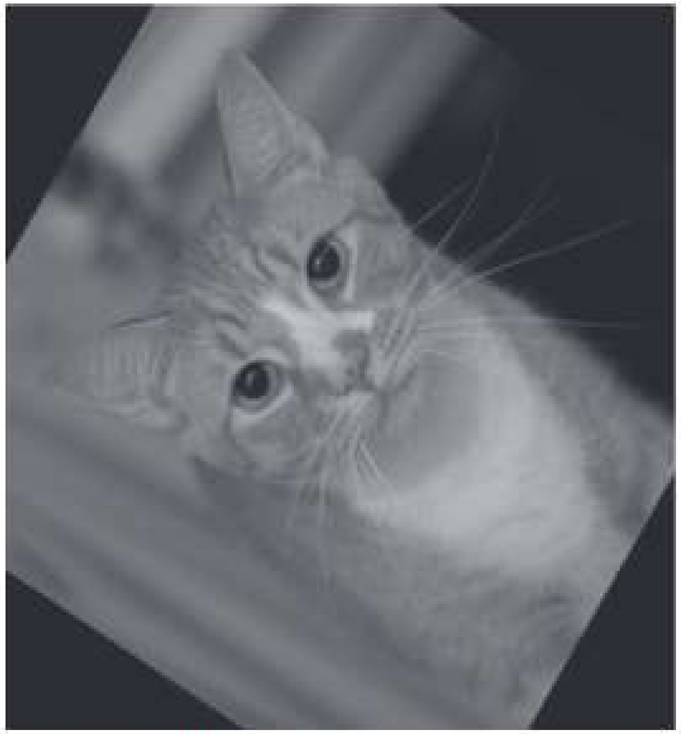

After rotation of 45°

IMAGE ROTATION

- Each work-item correspond to an output location, we can intuitively

map each work-item’s global ID to (x', y') in the previous equations.

- We can also determine $(x0, y0)$, which corresponds to the center of the image, as soon as we load the image from disk.
- We have two equations and two unknowns, which allows us to compute the location read by each work-item when computing the rotation:

$$x = x' \cos\theta - y' \sin\theta + x0 ,$$

$$y = x' \sin\theta + y' \cos\theta + y0 .$$

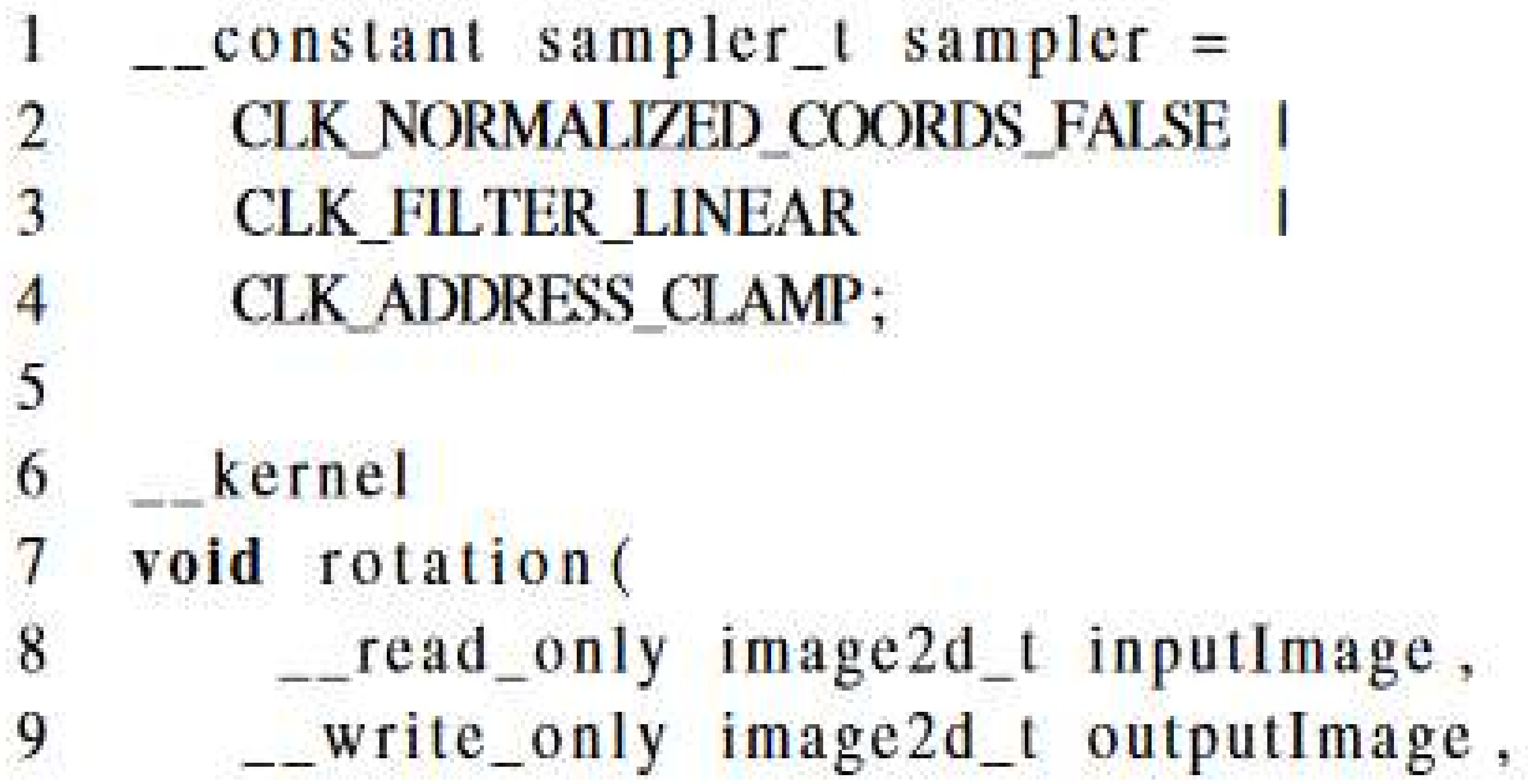
IMAGE ROTATION

- This corresponds to the following OpenCL C pseudocode:

```
gidx = get_global_id(0)
gidy = get_global_id(1)
x0 = width/2
y0 = height/2
x = gidx*cos(theta) - gidy*sin(theta) + x0
y = gidx*sin(theta) + gidy*cos(theta) + y0
```

IMAGE ROTATION

- OpenCL kernel that performs image rotation.

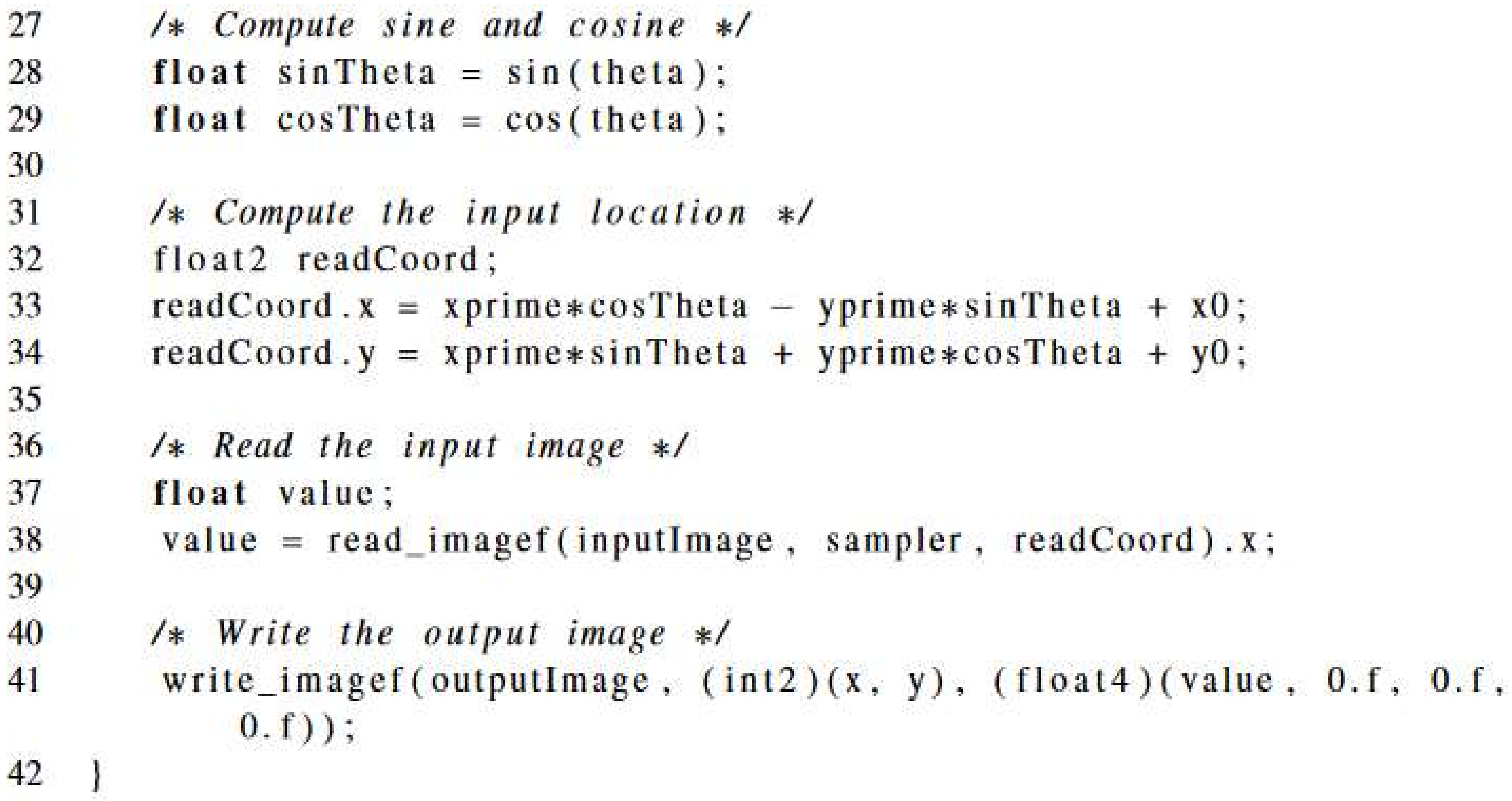


```
__constant sampler_t sampler =
    CLK_NORMALIZED_COORDS_FALSE |
    CLK_FILTER_LINEAR           |
    CLK_ADDRESS_CLAMP;

__kernel
void rotation(
      __read_only image2d_t inputImage,
     __write_only image2d_t outputImage,
```

IMAGE ROTATION

```
                              int imageWidth,
                              int imageHeight,
                            float theta)
{
    /* Get global ID for output coordinates */
    int x = get_global_id(0);
    int y = get_global_id(1);

    /* Compute image center */
    float x0 = imageWidth/2.0f;
    float y0 = imageHeight/2.0f;

    /* Compute the work-item's location relative
     * to the image center */
    int xprime = x-x0;
    int yprime = y-y0;

```

IMAGE ROTATION



```
    /* Compute sine and cosine */
    float sinTheta = sin(theta);
    float cosTheta = cos(theta);

    /* Compute the input location */
    float2 readCoord;
    readCoord.x = xprime*cosTheta - yprime*sinTheta + x0;
    readCoord.y = xprime*sinTheta + yprime*cosTheta + y0;

    /* Read the input image */
    float value;
    value = read_imagef(inputImage, sampler, readCoord).x;

    /* Write the output image */
    write_imagef(outputImage, (int2)(x, y), (float4)(value, 0.f, 0.f,
        0.f));
}
```

IMAGE ROTATION

- we use read_imagef() (Line 38) since we are working with floating-point data.
- read_imagef() returns a 4-wide vector data type.
- Since we are working with a single-channel image we are interested only in the first component, which we can access by appending .x to the end of the function call (Line 38).
- The call to write to an image also takes a 4-wide vector
- regardless of the actual data type, but will be handled appropriately by the hardware.

Summary

The following topics have been discussed:

- CUDA Error Handling
- Parallel Programming Issues
- Synchronization
- Algorithmic Issues
- Finding and Avoiding Errors

Objectives

- OpenCL Basics
- Execution Environment
- Memory Models
- Basic OpenCL Examples

Test Your Understanding

- Write a OpenCL code to sort an array of elements using quick sort
- Write a OpenCL code to search for an element in an an array using binary search