CHAPTER

An Introduction to OpenCL™ 14

CHAPTER OUTLINE

Now that we have discussed high-performance parallel programming using CUDA C, we would like to introduce another way to exploit the parallel computing capabilities of heterogeneous computing systems with GPUs and CPUs: OpenCL™. In this chapter, we will give a brief overview of OpenCL for CUDA programmers. The fundamental programming model of OpenCL is so similar to CUDA that there is a one-to-one correspondence for most features. With your understanding of CUDA, you will be able to start writing OpenCL programs with the material presented in this chapter. In our opinion, the best way to learn OpenCL is actually to learn CUDA first and then map the OpenCL features to their CUDA equivalents.

14.1 BACKGROUND

OpenCL is a standardized, cross-platform parallel computing API based on the C language. It is designed to enable the development of portable parallel applications for systems with heterogeneous computing devices. The development of OpenCL was motivated by the need for a standardized high-performance application development platform for the fast-growing variety of parallel computing platforms. In particular, it addresses significant application portability limitations of the previous programming models for heterogeneous parallel computing systems.

CPU-based parallel programming models have been typically based on standards such as OpenMP but usually do not encompass the use of special memory types or SIMD (single instruction, multiple data) execution by high-performance programmers. Joint CPU−GPU heterogeneous parallel programming models such as CUDA have constructs that address complex memory hierarchies and SIMD execution but have been platform-, vendor-, or hardware-specific. These limitations make it difficult for an application developer to access the computing power of CPUs, GPUs, and other types of processing units from a single multiplatform source code base.

The development of OpenCL was initiated by Apple and managed by the Khronos Group, the same group that manages the OpenGL standard. On one hand, it draws heavily on CUDA in the areas of supporting a single code base for heterogeneous parallel computing, data parallelism, and complex memory hierarchies. This is the reason why a CUDA programmer will find these aspects of OpenCL familiar once we connect the terminologies. Readers will especially appreciate the similarities between OpenCL and the low-level CUDA driver model.

On the other hand, OpenCL has a more complex platform and device management model that reflects its support for multiplatform and multivendor portability. OpenCL implementations already exist on AMD/ATI and NVIDIA GPUs as well as X86 CPUs. In principle, one can envision OpenCL implementations on other types of devices such as digital signal processors (DSPs) and field programmable gate arrays (FPGAs). While the OpenCL standard is designed to support code portability across devices produced by different vendors, such portability does not come for free. OpenCL programs must be prepared to deal with much greater hardware diversity and thus will exhibit more complexity. Also, many OpenCL features are optional and may not be supported on all devices. A portable OpenCL code will need to avoid using these optional features. However, some of these optional features allow applications to achieve significantly more performance in devices that support them. As a result, a portable OpenCL code may not be able to achieve its performance potential on any of the devices. Therefore, one should expect that a portable application that achieves high performance on multiple devices will employ sophisticated runtime tests and chose among multiple code paths according to the capabilities of the actual device used.

The objective of this chapter is not to provide full details on all programming features of OpenCL. Rather, the objective is to give a CUDA programmer a conceptual understanding of the OpenCL programming model features. It also provides some basic host and kernel code patterns for jumpstarting an OpenCL coding project. With this foundation, readers can immediately start to program in OpenCL and consult the OpenCL specification [KHR] and programming guides [NVIDIA,AMD] on a need basis.

14.2 DATA PARALLELISM MODEL

OpenCL employs a data-parallel execution model that has direct correspondence with CUDA. An OpenCL program consists of two parts: kernels that execute on one or more OpenCL devices and a host program that manages the execution of kernels. Table 14.1 summarizes the mapping of OpenCL data parallelism concepts to their CUDA equivalents. Like CUDA, the way to submit work for parallel execution in OpenCL is for the host program to launch kernel functions. We will discuss the additional kernel preparation, device selection, and management work that an OpenCL host program needs to do as compared to its CUDA counterpart in Section 14.4.

When a kernel function is launched, its code is run by *work items*, which correspond to CUDA threads. An index space defines the work items and how data is mapped to the work items. That is, OpenCL work items are identified by global dimension index ranges (NDRanges). Work items form *work groups* that correspond to CUDA thread blocks. Work items in the

Table 14.1 Mapping between OpenCL and CUDA Data Parallelism Model Concepts

OpenCL Parallelism Concept	CUDA Equivalent
Kernel	Kernel
Host program	Host program
NDRange (index space)	Grid
Work item	Thread
Work group	Block

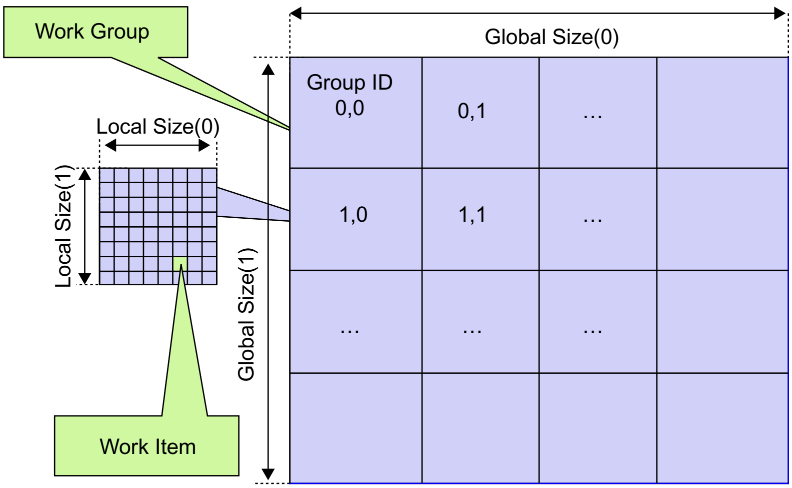

FIGURE 14.1

Overview of the OpenCL parallel execution model.

same work group can synchronize with each other using barriers that are equivalent to `__syncthreads()` in CUDA. Work items in different work groups cannot synchronize with each other except by terminating the kernel function and launching a new one. As we discussed in Chapter 4, this limited scope of barrier synchronization enables transparent scaling.

Figure 14.1 illustrates the OpenCL data-parallel execution model. Readers should compare Figure 14.1 with Figure 12.8 for similarities. The NDRange (CUDA grid) contains all work items (CUDA threads). For this example, we assume that the kernel is launched with a 2D NDRange.

All work items have their own unique global index values. There is a minor difference between OpenCL and CUDA in the way they manage these index values. In CUDA, each thread has a `blockIdx` value and a `threadIdx` value. The two values are combined to form a global thread ID value for the thread. For example, if a CUDA grid and its blocks are organized as 2D arrays, the kernel code can form a unique global thread index value in the x dimension as `blockIdx.x*blockDim.x+threadIdx.x`. These `blockIdx` and `threadIdx` values are accessible in a CUDA kernel as predefined variables.

In an OpenCL kernel, a thread can get its unique global index values by calling an API function `get_global_id()` with a parameter that identifies the dimension. See the `get_global_id(0)` entry in Table 14.2. The

Table 14.2 Mapping of OpenCL Dimensions and Indices to CUDA Dimensions and Indices

OpenCL API Call	Explanation	CUDA Equivalent
`get_global_id(0)`	Global index of the work item in the *x* dimension	`blockIdx.x*blockDim.x + threadIdx.x`
`get_local_id(0)`	Local index of the work item within the work group in the *x* dimension	`threadIdx.x`
`get_global_size(0)`	Size of NDRange in the *x* dimension	`gridDim.x*blockDim.x`
`get_local_size(0)`	Size of each work group in the *x* dimension	`blockDim.x`

functions `get_global_id(0)` and `get_global_id(1)` return the global thread index values in the x dimension and the y dimension, respectively. The global index value in the x dimension is equivalent to the `blockIdx.x*blockDim.x + threadIdx.x` in CUDA. See Table 14.2 for the `get_local_id(0)` function, which is equivalent to `threadIdx.x`. We did not show the parameter values in Table 14.2 for selecting the higher-dimension indices: 1 for the y dimension and 2 for the z dimension.

An OpenCL kernel can also call an API function `get_global_size()` with a parameter that identifies the dimensional sizes of its NDRanges. The calls `get_global_size(0)` and `get_global_size(1)` return the total number of work items in the x and y dimensions of the NDRanges. Note that this is slightly different from the CUDA `gridDim` values, which are in terms of blocks. The CUDA equivalent for the `get_global_size(0)` return value would be `gridDim.x * blockDim.x`.

14.3 DEVICE ARCHITECTURE

Like CUDA, OpenCL models a heterogeneous parallel computing system as a host and one or more OpenCL devices. The host is a traditional CPU that executes the host program. Figure 14.2 shows the conceptual architecture of an OpenCL device. Each device consists of one or more *compute units* (CUs) that correspond to CUDA streaming multiprocessors (SMs). However, a compute unit can also correspond to CPU cores or other types of execution units in compute accelerators such as DSPs and FPGAs.

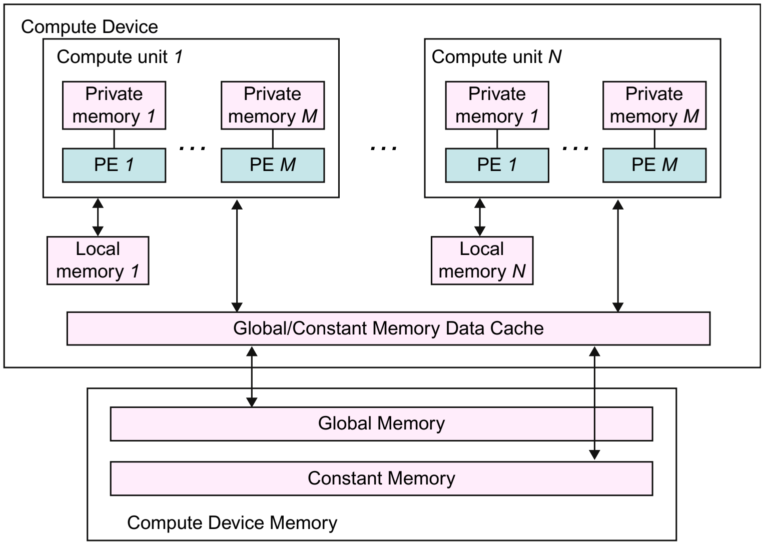

FIGURE 14.2

Conceptual OpenCL device architecture.

Each compute unit, in turn, consists of one or more *processing elements* (PEs), which corresponds to the streaming processors (SPs) in CUDA. Computation on a device ultimately happens in individual PEs.

Like CUDA, OpenCL also exposes a hierarchy of memory types that can be used by programmers. Figure 14.2 illustrates these memory types: global, constant, local, and private. Table 14.3 summarizes the supported use of OpenCL memory types and the mapping of these memory types to CUDA memory types. The OpenCL global memory corresponds to the CUDA global memory. Like CUDA, the global memory can be dynamically allocated by the host program and supports read/write access by both host and devices.

Unlike CUDA, the constant memory can be dynamically allocated by the host. Like CUDA, the constant memory supports read/write access by the host and read-only access by devices. To support multiple platforms, OpenCL provides a device query that returns the constant memory size supported by the device.

The mapping of OpenCL local memory and private memory to CUDA memory types is more interesting. The OpenCL local memory actually

Table 14.3 Mapping of OpenCL Memory Types to CUDA Memory Types

Memory Type	Host Access	Device Access	CUDA Equivalent
Global memory	Dynamic allocation; read/write access	No allocation; read/write access by all work items in all work groups, large and slow, but may be cached in some devices	Global memory
Constant memory	Dynamic allocation; read/write access	Static allocation; read-only access by all work items	Constant memory
Local memory	Dynamic allocation; no access	Static allocation; shared read/write access by all work items in a work group	Shared memory
Private memory	No allocation; no access	Static allocation; read/write access by a single work item	Registers and local memory

corresponds to CUDA shared memory. The OpenCL local memory can be dynamically allocated by the host or statically allocated in the device code. Like the CUDA shared memory, the OpenCL local memory cannot be accessed by the host and it supports shared read/write access by all work items in a work group. The private memory of OpenCL corresponds to the CUDA automatic variables.

14.4 KERNEL FUNCTIONS

OpenCL kernels have an identical basic structure as CUDA kernels. All OpenCL kernel declarations start with a `__kernel` keyword, which is equivalent to the `__global__` keyword in CUDA. Figure 14.3 shows a simple OpenCL kernel that performs vector addition.

The function takes three arguments: pointers to the two input arrays and one pointer to the output array. The `__global` declarations in the function header indicate that the input and output arrays all reside in the global memory. Note that this keyword has the same meaning in OpenCL as in CUDA, except that there are two underscore characters (__) after the global keyword in CUDA.

The body of the kernel function is instantiated once for each work item. In Figure 14.3, each work item calls the `get_global_id(0)` function

```
__kernel void  vadd(__global const float *a,
     __global const float *b, __global float *result) {

     int i = get_global_id(0);
     result[i] = a[i] + b[i];
}
```

FIGURE 14.3

A simple OpenCL kernel example.

to receive their unique global index. This index value is then used by the work item to select the array elements to work on. Once the array element index `i` is formed, the rest of the kernel is virtually identical to the CUDA kernel.

14.5 DEVICE MANAGEMENT AND KERNEL LAUNCH

OpenCL defines a much more complex model of device management than CUDA. The extra complexity stems from the OpenCL support for multiple hardware platforms. OpenCL supports runtime construction and compilation of kernels to maximize an application's ability to address portability challenges across a wide range of CPUs and GPUs. Interested readers should refer to the OpenCL specification for more insight into the work that went into the OpenCL specification to cover as many types of potential OpenCL devices as possible [KHR2011].

In OpenCL, devices are managed through *contexts*. Figure 14.4 illustrates the main concepts of device management in OpenCL. To manage one or more devices in the system, the OpenCL programmer first creates a context that contains these devices. A context is essentially an address space that contains the accessible memory locations to the OpenCL devices in the system. This can be done by calling either `clCreateContext()` or `clCreateContextFromType()` in the OpenCL API.

Figure 14.5 show a simple host code pattern for managing OpenCL devices. In line 4, we use `clGetContextInfo()` to get the number of bytes needed (`parmsz`) to hold the device information, which is used in line 5 to allocate enough memory to hold the information about all the devices available in the system. This is because the amount of memory needed to hold the information depends on the number of OpenCL devices in the system. We then call `clGetContextInfo()` again in line 6 with the size of the device information and a pointer to the allocated memory for the

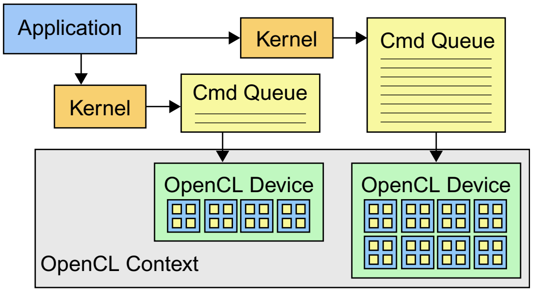

FIGURE 14.4

An OpenCL context is needed to manage devices.

```
…
1. cl_int clerr = CL_SUCCESS;

2. cl_context clctx=clCreateContextFromType(0, CL_DEVICE_TYPE_ALL,
        NULL, NULL, &clerr);

3. size_t parmsz;
4. clerr= clGetContextInfo(clctx, CL_CONTEXT_DEVICES, 0, NULL, &parmsz);

5. cl_device_id* cldevs= (cl_device_id *) malloc(parmsz);
6. clerr= clGetContextInfo(clctx, CL_CONTEXT_DEVICES, parmsz, cldevs, NULL);

7. cl_command_queue clcmdq=clCreateCommandQueue(clctx, cldevs[0], 0, &clerr);
```

FIGURE 14.5

Creating OpenCL context and command queue.

device information so that the function can deposit information on all the devices in the system into the allocated memory. An application could also use the `clGetDeviceIDs()` API function to determine the number and types of devices that exist in a system. Readers should read the *OpenCL Programming Guide* on the details of the parameters to be used for these functions [Khronos].

To submit work for execution by a device, the host program must first create a command queue for the device. This can be done by calling the `clCreateCommandQueue()` function is the OpenCL API. Once a command queue is created for a device, the host code can perform a sequence of API function calls to insert a kernel along with its execution configuration parameters into the command queue. When the device is available for executing the next kernel, it removes the kernel at the head of the queue for execution.

Figure 14.5 shows a simple host program that creates a context for a device and submits a kernel for execution by the device. Line 2 shows a call to create a context that includes all OpenCL available devices in the system. Line 4 calls the `clGetContextInfo()` function to inquire about the number of devices in the context. Since line 2 asks that all OpenCL available devices be included in the context, the application does not know the number of devices actually included in the context after the context is created. The second argument of the call in line 4 specifies that the information being requested is the list of all devices included in the context. However, the fourth argument, which is a pointer to a memory buffer where the list should be deposited, is a NULL pointer. This means that the call does not want the list itself. The reason is that the application does not know the number of devices in the context and does not know the size of the memory buffer required to hold the list.

Rather, line 4 provides a pointer to the variable `parmsz`. After line 4, the `parmsz` variable holds the size of the buffer needed to accommodate the list of devices in the context. The application now knows the amount of memory buffer needed to hold the list of devices in the context. It allocates the memory buffer using `parmsz` and assigns the address of the buffer to the pointer variable `cldevs` at line 5.

Line 6 calls `clGetContextInfo()` again with the pointer to the memory buffer in the fourth argument and the size of the buffer in the third argument. Since this is based on the information from the call at line 4, the buffer is guaranteed to be the right size for the list of devices to be returned. The `clGetContextInfo` function now fills the device list information into the memory buffer pointed to by `cldevs`.

Line 7 creates a command queue for the first OpenCL device in the list. This is done by treating `cldevs` as an array of which the elements are descriptors of OpenCL devices in the system. Line 7 passes `cldevs[0]` as the second argument into the `clCreateCommandQueue(0)` function. Therefore, the call generates a command queue for the first device in the list returned by the `clGetContextInfo()` function.

Readers may wonder why we did not need to see this complex sequence of API calls in our CUDA host programs. The reason is that we have been using the CUDA runtime API that hides all this complexity for the common case where there is only one CUDA device in the system. The kernel launch in CUDA handles all the complexities on behalf of the host code. If the developer wanted to have direct access to all CUDA devices in the system, he or she would need to use the CUDA driver API, where similar API calling sequences would be used. To date, OpenCL has not defined a higher-level API that is equivalent to the CUDA runtime API. Until such a higher-level interface is available, OpenCL will remain much more tedious to use than the CUDA runtime API. The benefit, of course, is that an OpenCL application can execute on a wide range of devices.

14.6 ELECTROSTATIC POTENTIAL MAP IN OPENCL

We now present an OpenCL case study based on the DCS kernel in Figure 12.9. This case study is designed to give a CUDA program a practical, top-to-bottom experience with OpenCL. The first step in porting the kernel to OpenCL is to design the organization of the NDRange, which is illustrated in Figure 14.5. The design is a straightforward mapping of CUDA threads to OpenCL work items and CUDA blocks to OpenCL work groups. As shown in Figure 14.6, each work item will calculate up to eight grid points and each work group will have 64 to 256 work items. All the efficiency considerations in Chapter 12 also apply here.

The work groups are assigned to the CUs the same way that CUDA blocks are assigned to the SMs. Such assignment is illustrated in

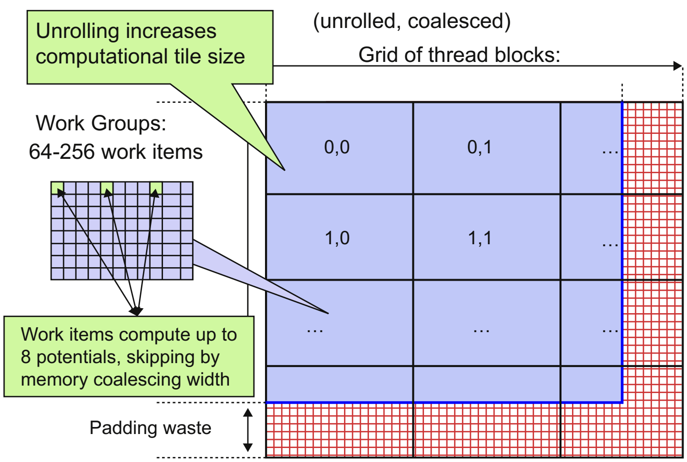

FIGURE 14.6

DCS kernel version 3 NDRange configuration.

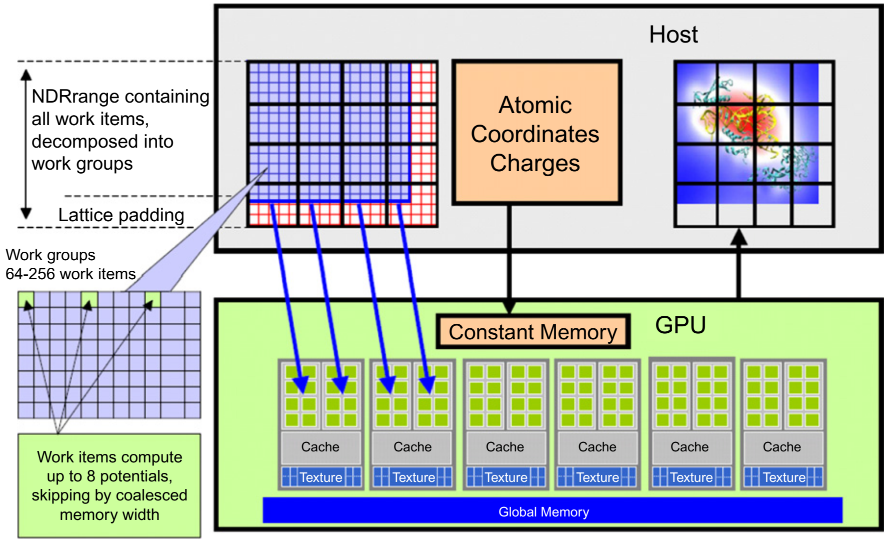

FIGURE 14.7

Mapping DCS NDRange to OpenCL device.

Figure 14.7. One can use the same methodology used in Chapters 6 and 12 to derive high-performance OpenCL DCS kernel. Although the syntax is different, the underlying thought process involved in developing a high-performance OpenCL kernel is very much the same as CUDA.

The OpenCL kernel function implementation matches closely the CUDA implementation. Figure 14.8 shows the key differences. One is the `__kernel` keyword in OpenCL versus the `__global` keyword in CUDA. The main difference lies in the way the data access indices are calculated. In this case, the OpenCL `get_global_id(0)` function returns the equivalent of CUDA `blockIdx.x*blockDim.x + threadIdx.x`.

Figure 14.9 shows the inner loop of the OpenCL kernel. Readers should compare this inner loop with the CUDA code in Figure 12.9. The only difference is that the `__rsqrt()` call has been changed to the `native_rsqrt()` call, the OpenCL way for using the hardware implementation of math functions on a particular device.

OpenCL adopts a dynamic compilation model. Unlike CUDA, the host program can explicitly compile and create a kernel program at runtime. This is illustrated in Figure 14.10 for the DCS kernel. Line 1 declares the entire OpenCL DCS kernel source code as a string. Line 3 delivers the

```
Device
OpenCL:
__kernel voidclenergy(…) {
unsigned int xindex= get_global_id(0);
unsigned int yindex= get_global_id(1);
unsigned int outaddr= get_global_size(0) * UNROLLX
*yindex+xindex;

CUDA:
__global__ void cuenergy(…) {
Unsigned int xindex= blockIdx.x *blockDim.x +threadIdx.x;
unsigned int yindex= blockIdx.y *blockDim.y +threadIdx.y;
unsigned int outaddr= gridDim.x *blockDim.x *
UNROLLX*yindex+xindex
```

FIGURE 14.8

Data access indexing in OpenCL and CUDA.

```
…
for (atomid=0; atomid<numatoms; atomid++) {
float dy = coory -atominfo[atomid].y;
float dyz2= (dy * dy) + atominfo[atomid].z;
float dx1 = coorx -atominfo[atomid].x;
float dx2 = dx1 + gridspacing_coalesce;
float dx3 = dx2 + gridspacing_coalesce;
float dx4 = dx3 + gridspacing_coalesce;
float charge = atominfo[atomid].w;
energyvalx1 += charge* native_rsqrt(dx1*dx1 + dyz2);
energyvalx2 += charge* native_rsqrt(dx2*dx2 + dyz2);
energyvalx3 += charge* native_rsqrt(dx3*dx3 + dyz2);
energyvalx4 += charge* native_rsqrt(dx4*dx4 + dyz2);
}
```

FIGURE 14.9

Inner loop of the OpenCL DCS kernel.

source code string to the OpenCL runtime system by calling the `clCreateProgramWith Source()` function. Line 4 sets up the compiler flags for the runtime compilation process. Line 5 invokes the runtime compiler to build the program. Line 6 requests that the OpenCL runtime create the kernel and its data structures so that it can be properly launched. After line 6, `clkern` points to the kernel that can be submitted to a command queue for execution.

Figure 14.11 shows the host code that launches the DCS kernel. It assumes that the host code for managing OpenCL devices in Figure 14.5

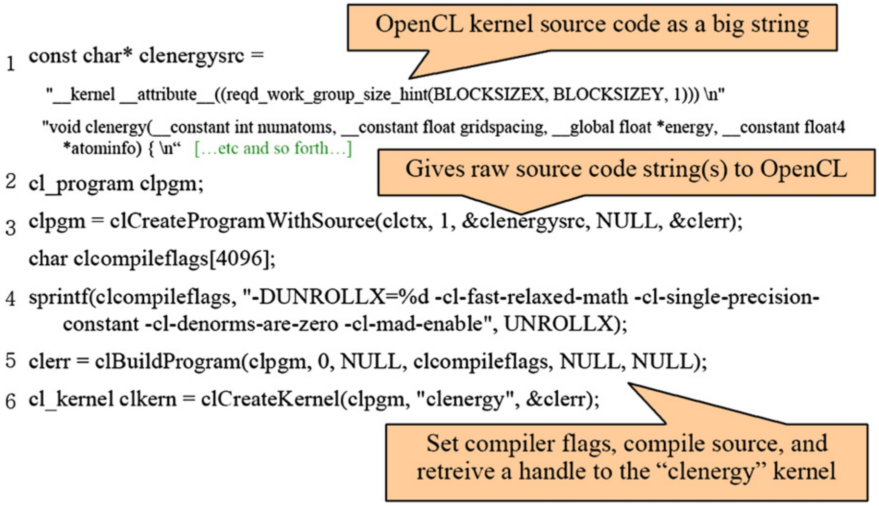

FIGURE 14.10

Building OpenCL kernel.

```
1.  doutput= clCreateBuffer(clctx, CL_MEM_READ_WRITE,volmemsz,
         NULL, NULL);
2.  datominfo= clCreateBuffer(clctx, CL_MEM_READ_ONLY,
         MAXATOMS *sizeof(cl_float4), NULL, NULL);
…
3.  clerr= clSetKernelArg(clkern, 0,sizeof(int), &runatoms);
4.  clerr= clSetKernelArg(clkern, 1,sizeof(float), &zplane);
5.  clerr= clSetKernelArg(clkern, 2,sizeof(cl_mem), &doutput);
6.  clerr= clSetKernelArg(clkern, 3,sizeof(cl_mem), &datominfo);
7.  cl_event event;
8.  clerr= clEnqueueNDRangeKernel(clcmdq,clkern, 2, NULL,
         Gsz,Bsz, 0, NULL, &event);
9.  clerr= clWaitForEvents(1, &event);
10. clerr= clReleaseEvent(event);
…
11. clEnqueueReadBuffer(clcmdq,doutput, CL_TRUE, 0,
         volmemsz, energy, 0, NULL, NULL);
12. clReleaseMemObject(doutput);
13. clReleaseMemObject(datominfo);
```

FIGURE 14.11

OpenCL host code for kernel launch and

has been executed. Lines 1 and 2 allocate memory for the energy grid data and the atom information. The `clCreateBuffer()` function corresponds to the `cudaMalloc()` function. The constant memory is implicitly requested by setting the mode of access to read only for the `atominfo` array. Note that each memory buffer is associated with a context, which is specified by the first argument to the `clCreateBuffer()` function call.

Lines 3–6 in Figure 14.11 set up the arguments to be passed into the kernel function. In CUDA, the kernel functions are launched with C function call syntax extended with $<<<>>>$, which is followed by the regular list of arguments. In OpenCL, there is no explicit call to kernel functions. Therefore, one needs to use the `clSetKernelArg()` functions to set up the arguments for the kernel function.

Line 8 in Figure 14.11 submits the DCS kernel for launch. The arguments to the `clEnqueueNDRangeKernel()` function specifies the command queue for the device that will execute the kernel, a pointer to the kernel, and the global and local sizes of the NDRange. Lines 9 and 10 check for errors if any.

Line 11 transfers the contents of the output data back into the energy array in the host memory. The OpenCL `clEnqueueReadBuffer()` copies data from the device memory to the host memory and corresponds to the device the host direction of the `cudaMemcpy()` function.

The `clReleaseMemObject()` function is a little more sophisticated than `cudaFree()`. OpenCL maintains a reference count for data objects. OpenCL host program modules can retain (`clRetainMemObject()`) and release (`clReleaseMemObject()`) data objects. Note that `clCreateBuffer()` also serves as a retain call. With each retain call, the reference count of the object is incremented. With each release call, the reference count is decremented. When the reference count for an object reaches 0, the object is freed. This way, a library module can “hang on” to a memory object even though the other parts of the application no longer need the object and thus have released the object.

14.7 SUMMARY

OpenCL is a standardized, cross-platform API designed to support portable parallel application development on heterogeneous computing systems. Like CUDA, OpenCL addresses complex memory hierarchies and data-parallel execution. It draws heavily on the CUDA driver API experience. This is the reason why a CUDA programmer finds these aspects of OpenCL familiar. We have seen this through the mappings of the OpenCL data parallelism model concepts, NDRange API calls, and memory types to their CUDA equivalents.

On the other hand, OpenCL has a more complex device management model that reflects its support for multiplatform and multivendor portability. While the OpenCL standard is designed to support code portability across devices produced by different vendors, such portability does not come for free. OpenCL programs must be prepared to deal with much greater hardware diversity and thus will exhibit more complexity. We see that the OpenCL device management model, the OpenCL kernel compilation model, and the OpenCL kernel launch are much more complex than their CUDA counterparts.

We have by no means covered all the programming features of OpenCL. Readers are encouraged to read the OpenCL specification [KHR2011] and tutorials [Khronos] for more OpenCL features. In particular, we recommend that readers pay special attention to the device query, object query, and task parallelism model.

14.8 EXERCISES

14.1 Use the code base in Appendix A and examples in Chapters 3, 4, 5, and 6 to develop an OpenCL version of the matrix–matrix multiplication application.

14.2 Read the "OpenCL Platform Layer" section of the OpenCL specification. Compare the platform querying API functions with what you have learned in CUDA.

14.3 Read the "Memory Objects" section of the OpenCL specification. Compare the object creation and access API functions with what you have learned in CUDA.

14.4 Read the "Kernel Objects" section of the OpenCL specification. Compare the kernel creation and launching API functions with what you have learned in CUDA.

14.5 Read the "OpenCL Programming Language" section of the OpenCL specification. Compare the keywords and types with what you have learned in CUDA.

References

AMD OpenCL Resources. Available at: <http://developer.amd.com/gpu/ATIStreamSDK/pages/TutorialOpenCL.aspx>.

Khronos Group, The OpenCL Specification version 1.1, rev44. Available at: <http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf>, 2011.

Khronos OpenCL samples, tutorials, etc., Available at: <http://www.khronos.org/developers/resources/opencl/>.

NVIDIA OpenCL Resources. Available at: <http://www.nvidia.com/object/cuda_opencl.html>.