OpenCL
Unit 5

Dr. Minal Moharir

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

BACKGROUND

- The development of OpenCL was initiated by Apple and developed by the Khronos Group, the same group that manages the OpenGL standard.
- similarities between OpenCL and the low-level CUDA driver model

BACKGROUND

- 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

BACKGROUND

- OpenCL implementations on other types of devices such as digital signal processors (DSPs) and field-programmable gate arrays (FPGAs).
- Whereas the OpenCL standard is designed to support code portability across devices produced by different vendors, such portability does not come free.
- OpenCL programs must be prepared to deal with much greater hardware diversity and thus will exhibit more complexity.

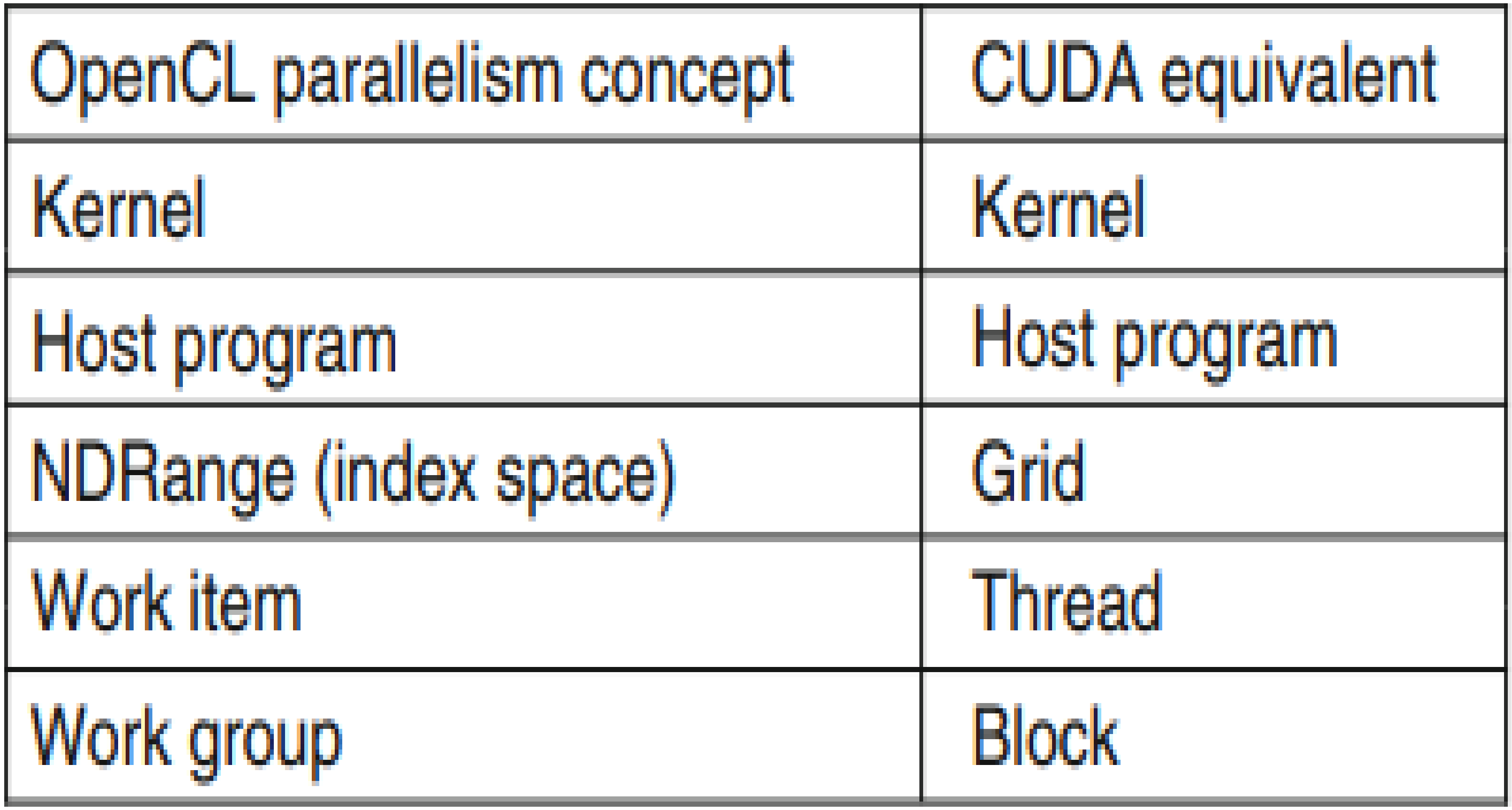
DATA PARALLELISM MODEL

- OpenCL employs a data parallelism model that has direct correspondence with the CUDA data parallelism model.
- 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

DATA PARALLELISM MODEL

- Figure 11.1 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.
- Here the additional kernel generation, device selection, and management work that an OpenCL host program must do as compared to its CUDA counterpart

DATA PARALLELISM MODEL



OpenCL parallelism concept	CUDA equivalent
Kernel	Kernel
Host program	Host program
NDRange (index space)	Grid
Work item	Thread
Work group	Block

DATA PARALLELISM MODEL

- 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 are mapped to the work items; that is, OpenCL work items are identified by global dimension index ranges (NDRanges).
- Work items form work groups, which correspond to CUDA thread blocks.
- Work items in the 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

DATA PARALLELISM MODEL

- Figure 11.2 illustrates the OpenCL data parallelism model.
- The NDRange (CUDA grid) contains all work items (CUDA threads).
- For this example, we assume that the kernel is launched with a two-dimensional (2D) NDRange

DATA PARALLELISM MODEL

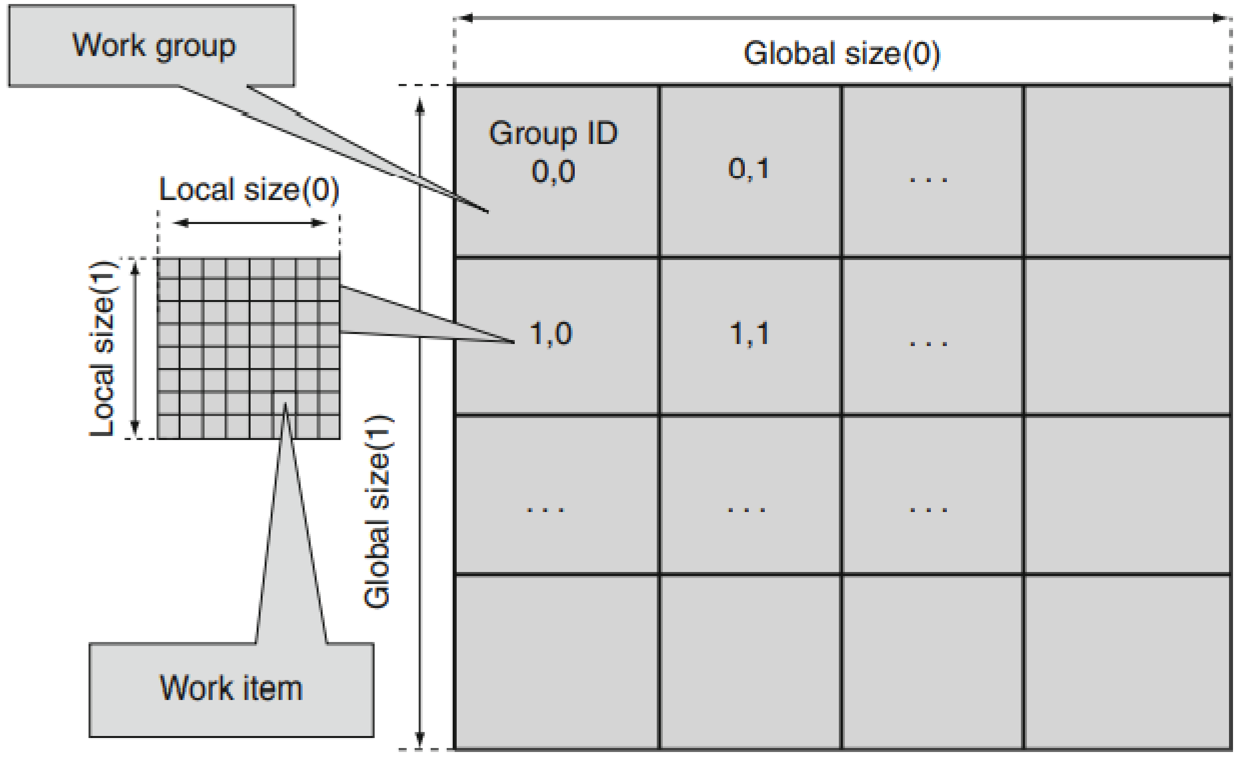
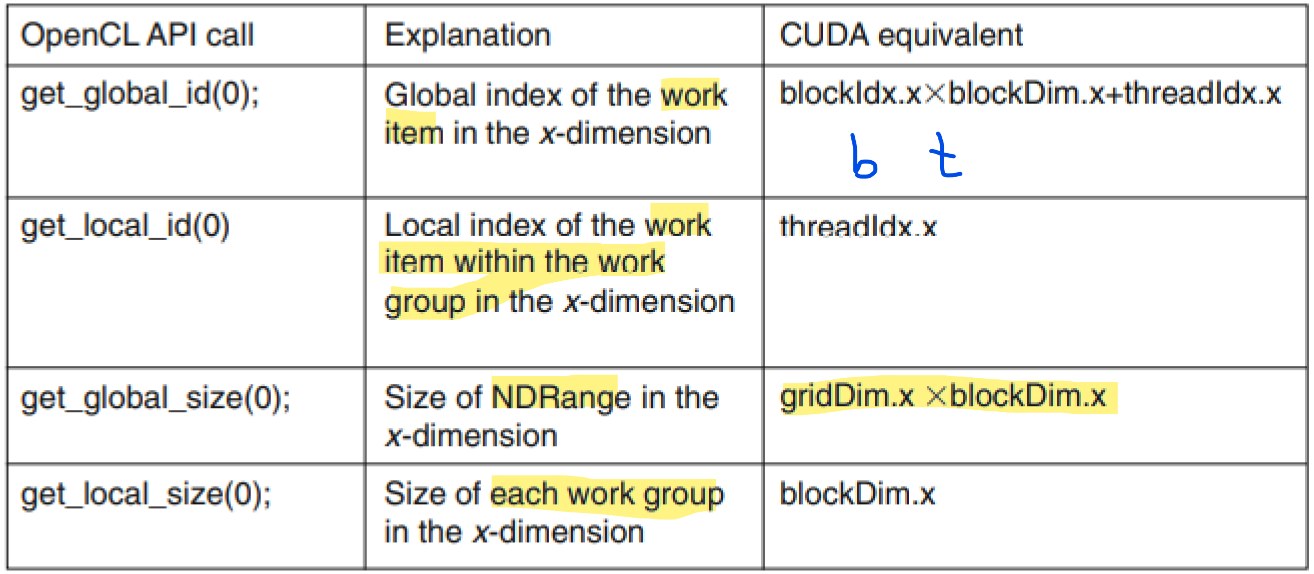


FIGURE 11.2

Overview of the OpenCL parallel execution model.

DATA PARALLELISM MODEL



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 b t
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

FIGURE 11.3

Mapping of OpenCL dimensions and indices to CUDA dimensions and indices.

DATA PARALLELISM MODEL

- 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 Figure 11.3. The calls get_global_id(0) and get_global_id(1) return the global thread index values in the x-dimension and the y-dimension, respectively.
- higher dimension parameter values; they are 1 for the y-dimension and 2 for the z-dimension

DATA PARALLELISM MODEL

- An OpenCL kernel can also call the 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.xblockDim.x ✓

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 11.4 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 CU can also correspond to CPU cores or other types of execution units in compute accelerators such as DSPs and FPGAs.

DEVICE ARCHITECTURE

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

DEVICE ARCHITECTURE

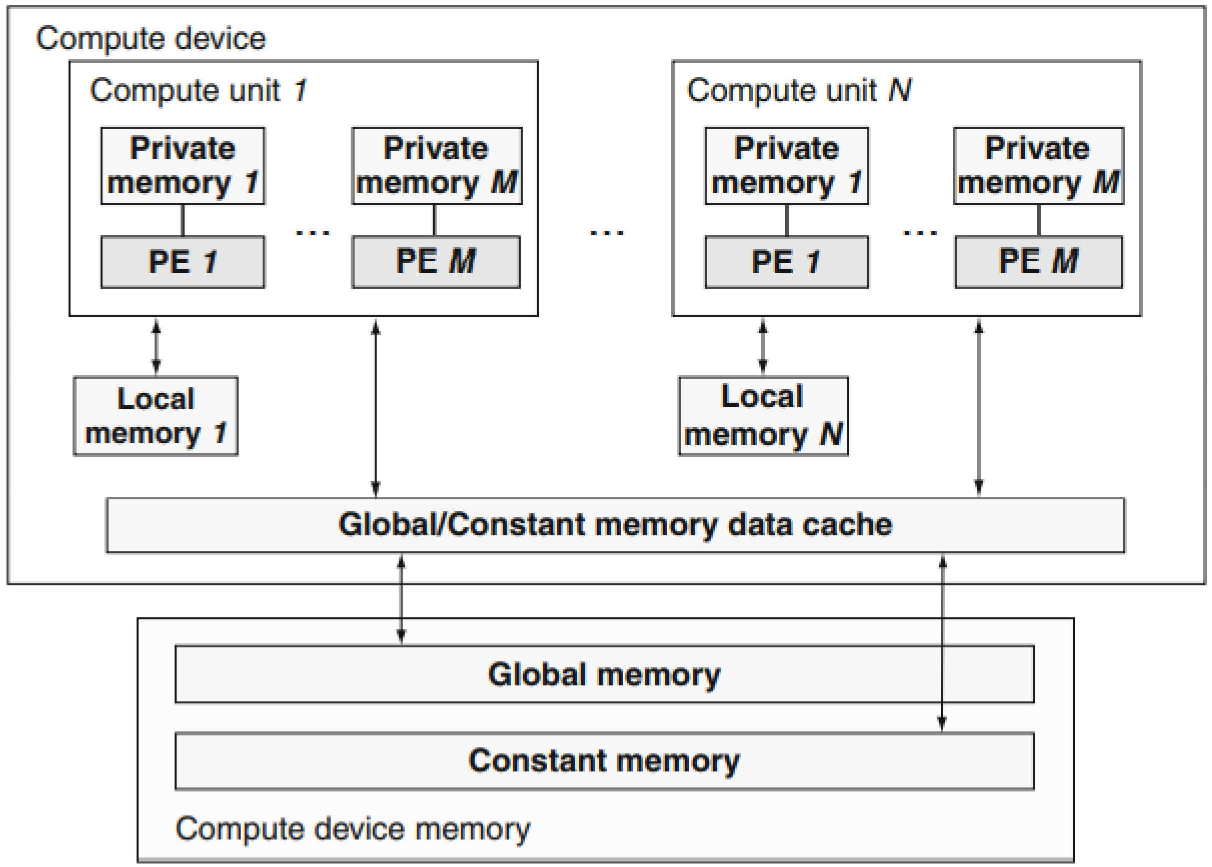


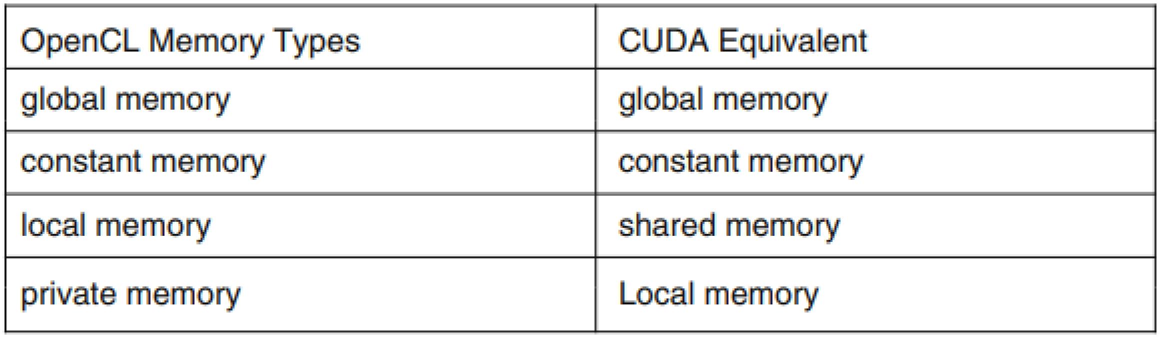
FIGURE 11.4

Conceptual OpenCL device architecture; the host is not shown.

DEVICE ARCHITECTURE

- Like CUDA, OpenCL also exposes a hierarchy of memory types that can be used by programmers.
- Figure 11.4 illustrates these memory types: global, constant, local, and private.
- Figure 11.5 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

DEVICE ARCHITECTURE



OpenCL Memory Types	CUDA Equivalent
global memory	global memory
constant memory	constant memory
local memory	shared memory
private memory	Local memory

FIGURE 11.5

Mapping of OpenCL memory types to CUDA memory types.

DEVICE ARCHITECTURE

- Unlike CUDA, the constant memory can be dynamically allocated by the host.
- The constant memory supports read/write access by the host and read-only access by devices.
- To support multiple platforms, the size of constant memory is not limited to 64 kB in OpenCL.
- Instead, a device query returns the constant memory size supported by the device.

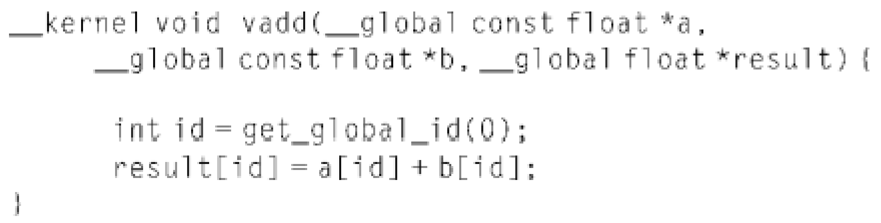
DEVICE ARCHITECTURE

- The mapping of OpenCL local memory and private memory to CUDA memory types is more interesting.
- The OpenCL local memory actually corresponds to CUDA shared memory.
- The OpenCL local memory can be dynamically allocated by the host and statically allocated in the device code.
- Like the CUDA shared memory, the OpenCL local memory cannot be accessed by the host and support shared read/write access by all work items in a work group.
- The private memory of OpenCL corresponds to the CUDA local memory.

KERNEL FUNCTIONS

- OpenCL kernels have identical basic structure as CUDA kernels.
- All OpenCL kernel function declarations start with a __kernel keyword, which is equivalent to the __global keyword in CUDA.
- Figure 11.6 shows a simple OpenCL kernel function 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.
- The body of the kernel function is instantiated once for each work item. In Figure 11.6, each work item calls the get_global_id(0) function to receive its unique global index.
- This index value is then used by the work item to select the array elements to work on.

KERNEL FUNCTIONS



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

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

FIGURE 11.6

A simple OpenCL kernel example.

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.
- In OpenCL, devices are managed through contexts.
- Figure 11.7 illustrates the main concepts of device management in OpenCL.

DEVICE MANAGEMENT AND KERNEL LAUNCH

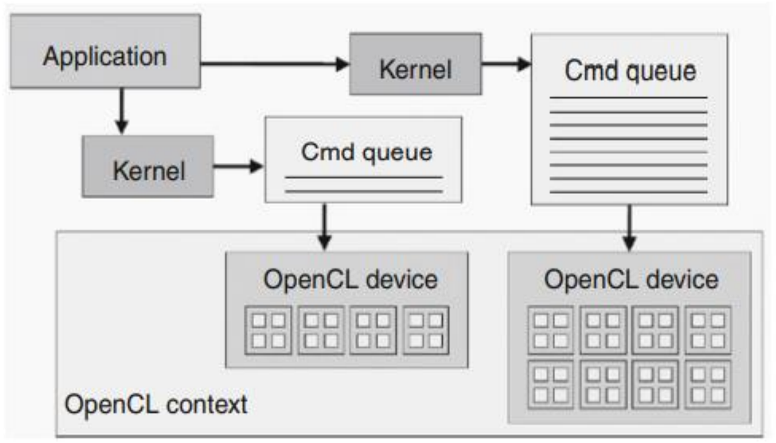


FIGURE 11.7

OpenCL context required to manage devices.

DEVICE MANAGEMENT AND KERNEL LAUNCH

- In order to manage one or more devices in the system, the OpenCL programmer first creates a context that contains these devices.
- This can be done by calling either clCreateContext() or clCreateContextFromType() in the OpenCL API.
- An application typically needs to use the clGetDeviceIDs() API function to determine the number and types of devices that exist in a system and to pass the information on to the CreateContext functions.
- For more Details:OpenCL programming guide

DEVICE MANAGEMENT AND KERNEL LAUNCH

- 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 in 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.

DEVICE MANAGEMENT AND KERNEL LAUNCH

- Figure 11.8 shows a simple host programs 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.
- ✓ Because 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.

DEVICE MANAGEMENT AND KERNEL LAUNCH

```
...
cl_int clerr = CL_SUCCESS;

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

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

cl_device_id* cldevs= (cl_device_id *) malloc(parmsz);
clerr= clGetContextInfo(clctx, CL_CONTEXT_DEVICES, parmsz,cldevs, NULL); ✓

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

FIGURE 11.8

Creating an OpenCL context and command queue.

DEVICE MANAGEMENT AND KERNEL LAUNCH

- 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.

DEVICE MANAGEMENT AND KERNEL LAUNCH

- Line 4 provides a pointer to the variable parmsz, where the size of a memory buffer required to accommodate the device list is to be deposited;
- therefore, after the call in 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 pointer variable cldevs at Line 5

DEVICE MANAGEMENT AND KERNEL LAUNCH

- 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.
- Because 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.

DEVICE MANAGEMENT AND KERNEL LAUNCH

- Line 7 creates a command queue for the first OpenCL device in the list.
- This is done by treating cldevs as an array whose 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.

DEVICE MANAGEMENT AND KERNEL LAUNCH

- 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

ELECTROSTATIC POTENTIAL MAP IN OpenCL

- OpenCL case study based on the DCS kernel:direct Coulomb summation
- Computation Chemistry Problem simulation to calculate iteration between molecules
- The design is a straightforward mapping of CUDA threads to OpenCL work items and CUDA blocks to OpenCL work groups.
- As shown in Figure 11.9, each work item will calculate up to eight grid points, and each work group will have 64–256 work items.

ELECTROSTATIC POTENTIAL MAP IN OpenCL

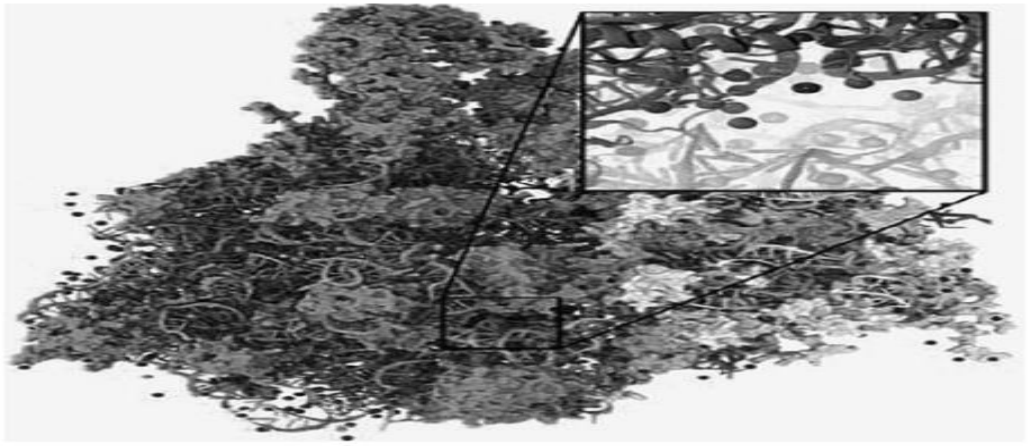
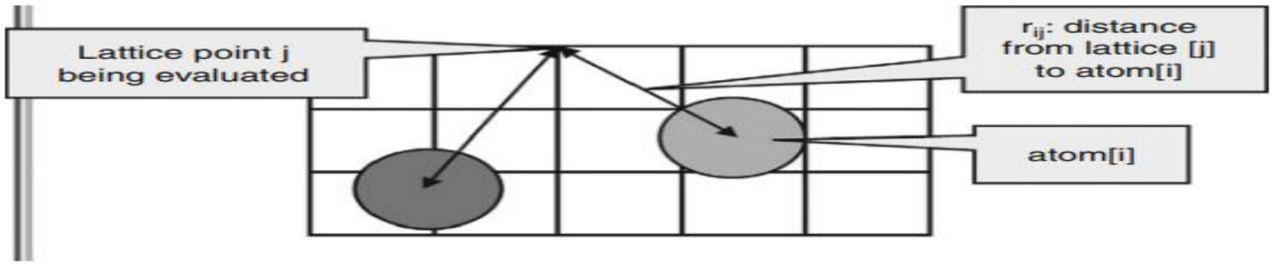
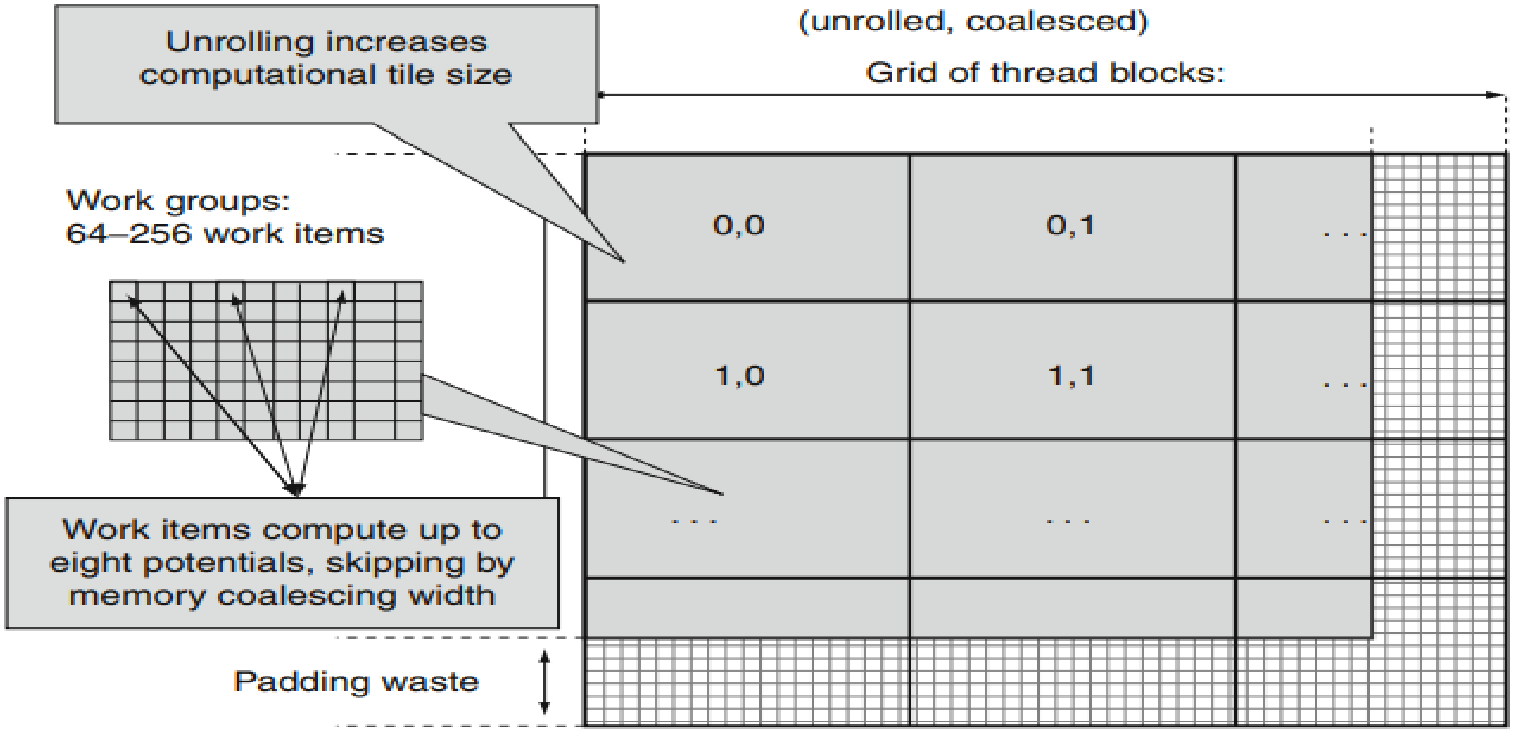


FIGURE 9.1

Electrostatic potential map used to build stable structures for molecular dynamics simulation.


Lattice point j being evaluated
r_{ij}: distance from lattice [j] to atom[i]
atom[i]

ELECTROSTATIC POTENTIAL MAP IN OpenCL



Unrolling increases computational tile size
(unrolled, coalesced)
Grid of thread blocks:
Work groups: 64–256 work items
0,0
0,1
. . .
1,0
1,1
. . .
. . .
. . .
. . .
Work items compute up to eight potentials, skipping by memory coalescing width
Padding waste

FIGURE 11.9

DCS Kernel Version 3 NDRange configuration.

ELECTROSTATIC POTENTIAL MAP IN OpenCL

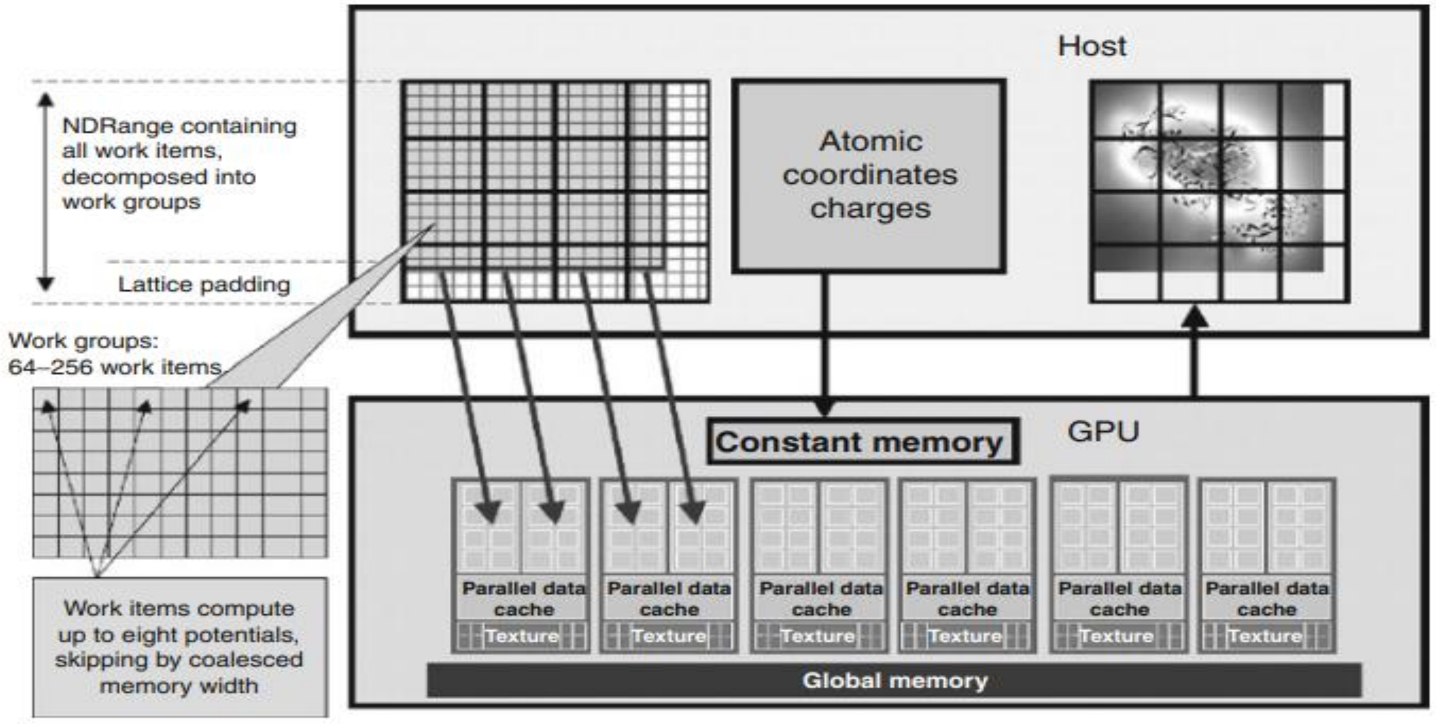

Host
NDRange containing all work items, decomposed into work groups
Lattice padding
Atomic coordinates charges
Work groups: 64–256 work items
Work items compute up to eight potentials, skipping by coalesced memory width
Constant memory
GPU
Parallel data cache
Texture
Global memory

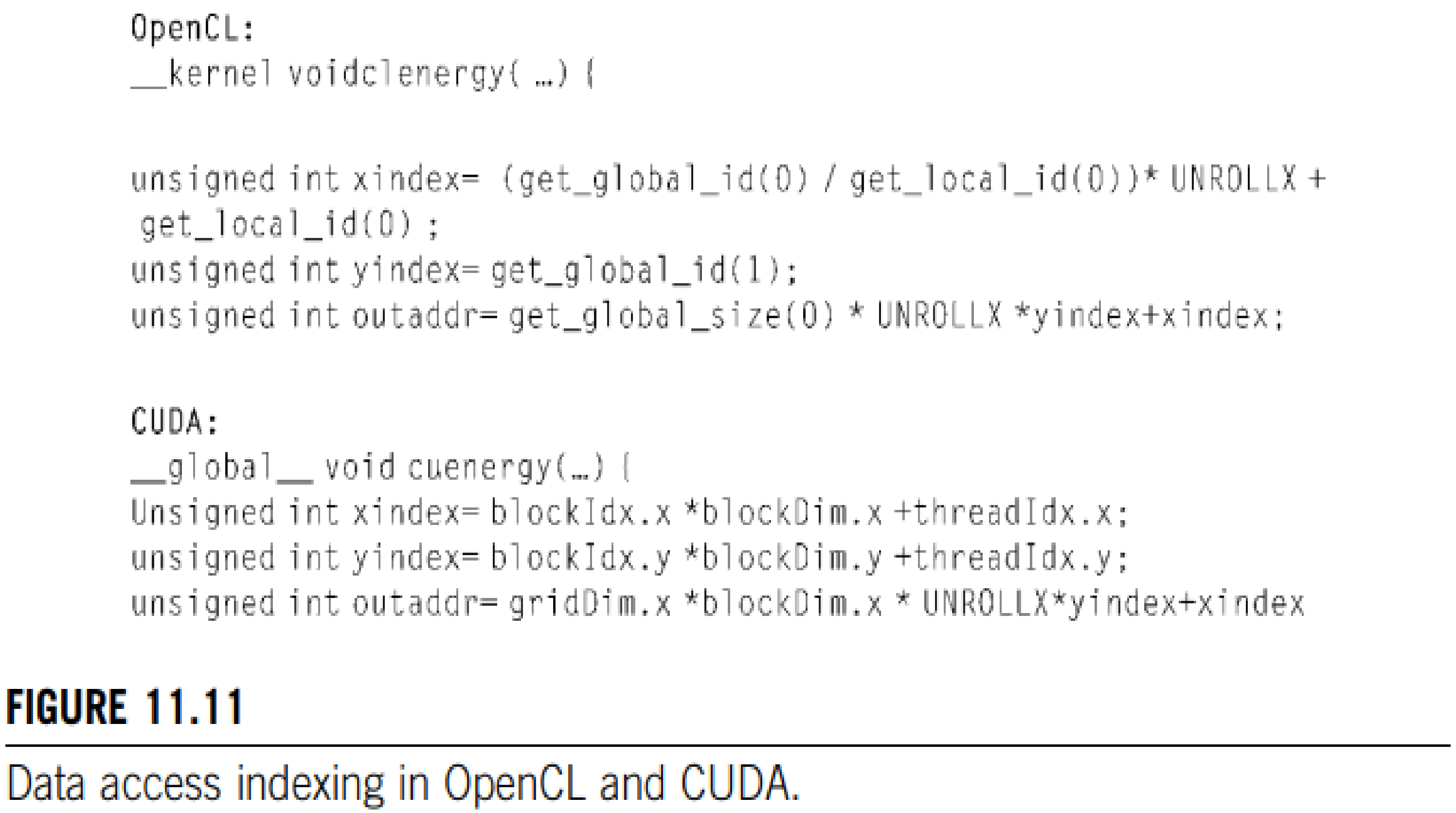
FIGURE 11.10

Mapping DCS NDRange to OpenCL device.

ELECTROSTATIC POTENTIAL MAP IN OpenCL

- The OpenCL kernel function implementation closely matches the CUDA implementation. Figure 11.11 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.

ELECTROSTATIC POTENTIAL MAP IN OpenCL



```
OpenCL:
__kernel voidclenergy( …) {

unsigned int xindex= (get_global_id(0) / get_local_id(0))* UNROLLX +
 get_local_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
```

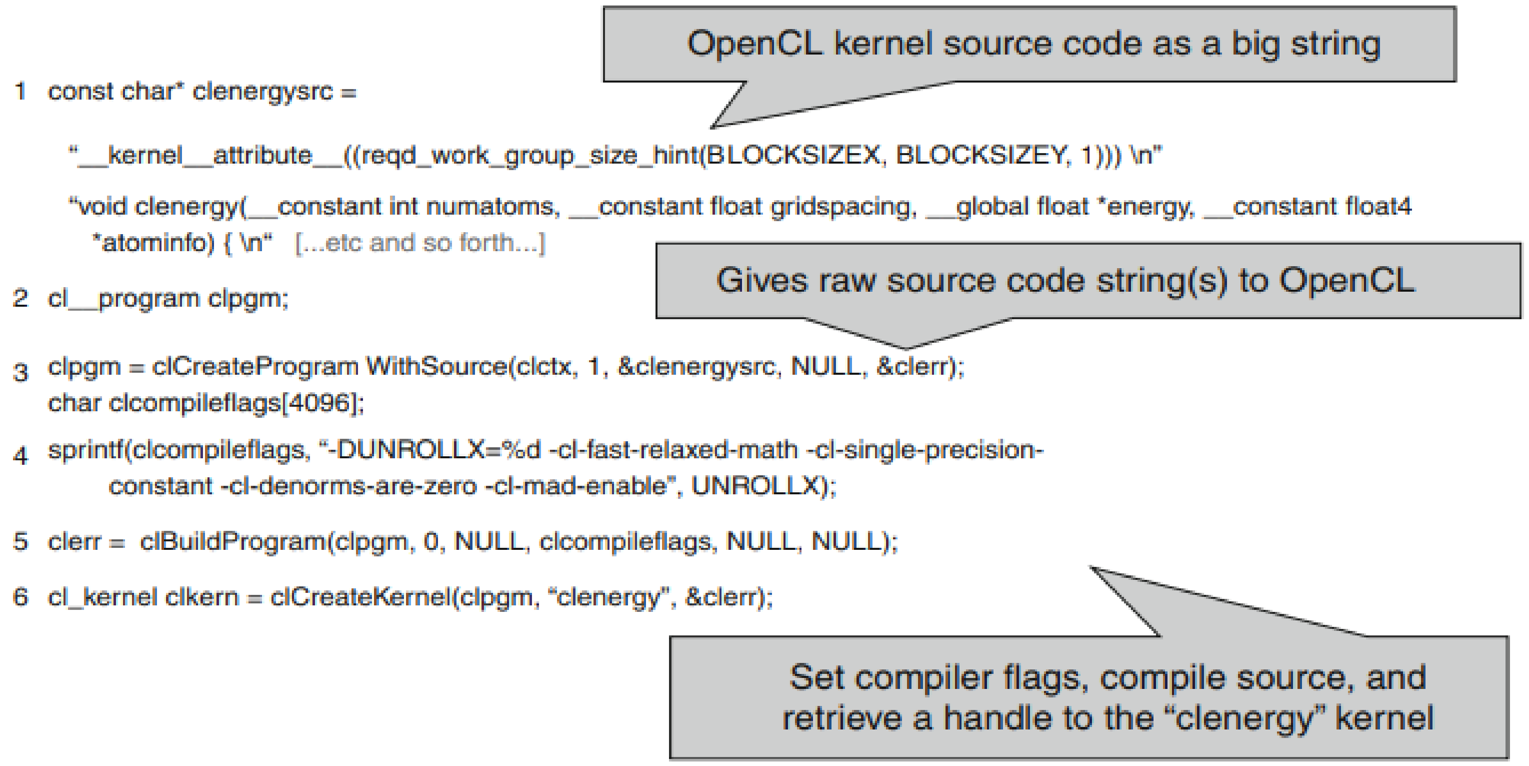
FIGURE 11.11

Data access indexing in OpenCL and CUDA.

ELECTROSTATIC POTENTIAL MAP IN OpenCL

- OpenCL adopts a dynamic compilation model. Unlike CUDA, the host program needs to explicitly compile and create a kernel program.
- This is illustrated in Figure 11.13 for the DCS kernel. Line 1 declares the entire OpenCL DCS kernel source code as a string.
- Line 3 delivers the source code string to the OpenCL runtime system by calling the clCreateProgramWithSource() function.

ELECTROSTATIC POTENTIAL MAP IN OpenCL


OpenCL kernel source code as a big string
Gives raw source code string(s) to OpenCL
Set compiler flags, compile source, and retrieve a handle to the "clenergy" kernel

```
const char* clenergysrc =
   "__kernel__attribute__((reqd_work_group_size_hint(BLOCKSIZEX, BLOCKSIZEY, 1))) \n"
   "void clenergy(__constant int numatoms, __constant float gridspacing, __global float *energy, __constant float4
     *atominfo) { \n"   [...etc and so forth...]
cl__program clpgm;
clpgm = clCreateProgram WithSource(clctx, 1, &clenergysrc, NULL, &clerr);
char clcompileflags[4096];
sprintf(clcompileflags, "-DUNROLLX=%d -cl-fast-relaxed-math -cl-single-precision-
      constant -cl-denorms-are-zero -cl-mad-enable", UNROLLX);
clerr =  clBuildProgram(clpgm, 0, NULL, clcompileflags, NULL, NULL);
cl_kernel clkern = clCreateKernel(clpgm, "clenergy", &clerr);
```

FIGURE 11.13

Building an OpenCL kernel.

ELECTROSTATIC POTENTIAL MAP IN OpenCL

- 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 it can be properly launched.
- After Line 6, clkern points to the kernel that can be submitted to a command queue for execution.

ELECTROSTATIC POTENTIAL MAP IN OpenCL

- one GPU to execute the kernel. Figure 11.14 shows the host code that actually launches the DCS kernel.
- Line 1 and Line 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 ready 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.

ELECTROSTATIC POTENTIAL MAP IN OpenCL

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

FIGURE 11.14

OpenCL host code for kernel launch and .

ELECTROSTATIC POTENTIAL MAP IN OpenCL

- Lines 3 through 6 in Figure 11.14 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 <<<>>>.
- In OpenCL, there is no explicit call to kernel functions, so one needs to use the clSetKernelArg() functions to set up the arguments for the kernel function.

ELECTROSTATIC POTENTIAL MAP IN OpenCL

- Lines 3 through 6 in Figure 11.14 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 <<<>>>.
- In OpenCL, there is no explicit call to kernel functions, so one needs to use the clSetKernelArg() functions to set up the arguments for the kernel function.

ELECTROSTATIC POTENTIAL MAP IN OpenCL

- Line 8 in Figure 11.14 submits the DCS kernel for launch.
- The arguments to the clEnqueueNDRangeKernel() function specify 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 to host direction of the cudaMemcpy() function.

ELECTROSTATIC POTENTIAL MAP IN OpenCL

- 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.

ELECTROSTATIC POTENTIAL MAP IN OpenCL

- 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 onto a memory object even though the other parts of the application no longer need the object and thus have released the object.