Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

A Gentle Introduction to OpenCL

Writing and running your first app with code executing on the CPU and the GPU

By Matthew Scarpino
July 31, 2011
URL:http://drdobbs.com/high-performance-computing/231002854

OpenCL provides many benefits in the field of high-performance computing, and one of the most important is portability. OpenCL-coded routines, called kernels, can execute on GPUs and CPUs from such popular manufacturers as Intel, AMD, Nvidia, and IBM. New OpenCL-capable devices appear regularly, and efforts are underway to port OpenCL to embedded devices, digital signal processors, and field-programmable gate arrays.

Not only can OpenCL kernels run on different types of devices, but a single application can dispatch kernels to multiple devices at once. For example, if your computer contains an AMD Fusion processor and an AMD graphics card, you can synchronize kernels running on both devices and share data between them. OpenCL kernels can even be used to accelerate OpenGL or Direct3D processing.

Despite these advantages, OpenCL has one significant drawback: it's not easy to learn. OpenCL isn't derived from MPI or PVM or any other distributed computing framework. Its overall operation resembles that of NVIDIA's CUDA, but OpenCL's data structures and functions are unique. Even the most introductory application is difficult for a newcomer to grasp. You really can't just dip your foot in the pool — you either know OpenCL or you don't.

My goal in writing this article is to explain the concepts behind OpenCL as simply as I can and show how these concepts are implemented in code. I'll explain how host applications work and then show how kernels execute on a device. Finally, I'll walk through an example application with a kernel that adds 64 floating-point values together.

Host Application Development

In developing an OpenCL project, the first step is to code the host application. This runs on a user's computer (the host) and dispatches kernels to connected devices. The host application can be coded in C or C++, and every host application requires five data structures: `cl_device_id`, `cl_kernel`, `cl_program`, `cl_command_queue`, and `cl_context`.

When I started learning OpenCL, I found it hard to remember these structures and how they work together, so I devised an analogy: An OpenCL host application is like a game of cards.

A Game of Cards

In a card game, a dealer sits at a table with one or more players and distributes cards from a deck. Each player receives these cards as part of a hand and then analyzes how best to play. The players can't interact with one

another or see another player's cards, but they can make requests to the dealer for additional cards or a change in stakes. The dealer handles these requests and takes control once the game is over. Figure 1 illustrates this analogoy.

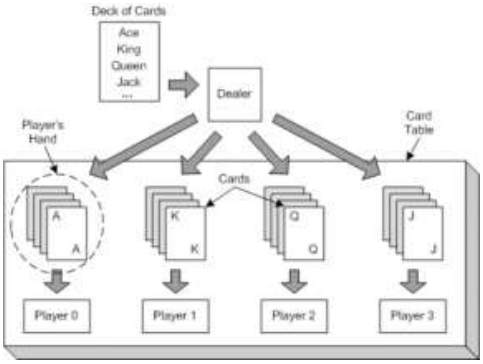

Figure 1: The card game analogy.

In addition to the dealer and the players, Figure 1 also depicts the table that supports the game. The players seated at the table don't have to take part, but only those seated at the table can participate in the game.

The Five Data Structures

In my analogy, the card dealer represents the host. The other aspects of the game correspond to the five OpenCL data structures that must be created and configured in a host application:

- Device: OpenCL devices correspond to the players. Just as a player receives cards from the dealer, a device receives kernels from the host. In code, a device is represented by a `cl_device_id`.
- Kernel: OpenCL kernels correspond to the cards. A host application distributes kernels to devices in much the same way a dealer distributes cards to players. In code, a kernel is represented by a `cl_kernel`.
- Program: An OpenCL program is like a deck of cards. In the same way that a dealer selects cards from a deck, the host selects kernels from a program. In code, a program is represented by a `cl_program`.
- Command queue: An OpenCL command queue is like a player's hand. Each player receives cards as part of a hand, and each device receives kernels through a command queue. In code, a command queue is represented by a `cl_command_queue`.
- Context: OpenCL contexts correspond to card tables. Just as a card table makes it possible for players to transfer cards to one another, an OpenCL context allows devices to receive kernels and transfer data. In code, a context is represented by a `cl_context`.

To clarify this analogy, Figure 2 shows how these five data structures work together in a host application. As shown, a program contains multiple functions, and each kernel encapsulates a function taken from the program.

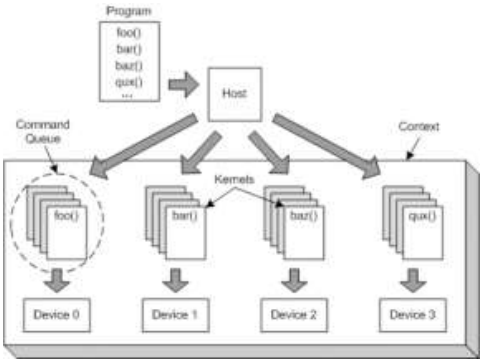

Figure 2: Distributing kernels to OpenCL-compliant devices.

Once you understand how host applications work, learning how to write code is straightforward. Most of the functions in the OpenCL API have straightforward names like `clCreateCommandQueue` and `clGetDeviceInfo`. Given the analogy, it should be clear that `clCreateKernel` requires a `cl_program` structure to execute, and `clCreateContext` requires one or more `cl_device_id` structures.

Shortcomings of the Analogy

My analogy has its flaws. Six significant shortcomings are given as follows:

1. There's no mention of platforms. A platform is a data structure that identifies a vendor's implementation of OpenCL. Platforms make it possible to access devices. For example, you can access an Nvidia device through the Nvidia platform.
2. A card dealer doesn't choose which players sit at the table. However, an OpenCL host selects which devices should be placed in a context.
3. A card dealer can't deal the same card to multiple players, but an OpenCL host can dispatch the same kernel to multiple devices through their command queues.
4. The analogy doesn't mention how devices execute kernels. Many OpenCL devices contain multiple processing elements, and each element may process a subset of the input data. The host identifies the number of work items that should be generated to execute the kernel.
5. In a card game, the dealer distributes cards to players and each player arranges the cards to form a hand. In OpenCL, the host creates a command queue for each device and enqueues commands. One type of command tells a device to execute a kernel.
6. In a card game, the dealer passes cards in a round-robin fashion. OpenCL sets no constraints on how host applications distribute kernels to devices.

At this point, you should understand that a large part of a host application's job involves creating kernels and deploying them to OpenCL-compliant devices such as GPUs, CPUs, or hybrid processors. Next, I'll discuss how these kernels execute on the devices.

OpenCL Kernels

One of OpenCL's great advantages is that kernels can execute on high-performance computing devices such as GPUs. To take advantage of these parallel-processing capabilities in code, an OpenCL developer needs to clearly understand two points:

1. The OpenCL Execution Model: Kernels are executed by one or more work-items. Work-items are collected into work-groups and each work-group executes on a compute unit.
2. The OpenCL Memory Model: Kernel data must be specifically placed in one of four address spaces — global memory, constant memory, local memory, or private memory. The location of the data determines how quickly it can be processed.

It takes time to understand these models, and the more familiar you are with them, the faster your code will run. To explain how the models interact, I've devised a second analogy: The execution of a kernel is like a day at school.

A Day at School

When I was in middle school, my teacher assigned 30 problems to her 30 students every day. But she didn't check all 900 answers. Instead, she assigned a different number to each of us, and we'd go to the front of the class and solve the problem with that number. We'd copy our work from our notebook to the blackboard, and if the teacher liked what she saw, we'd get a good grade. Clever, huh?

An OpenCL device is like a school composed of classrooms like mine. Each classroom contains students performing math problems. The students in a class share the same blackboard, but each student has a separate

notebook. Students in the same class can work together at their blackboard, but students in different classes can't work together.

Here's where it gets tricky: None of these classrooms have a teacher. Also, every student in the school works on the same math problem, but with different values. For example, if the problem involves adding two numbers, one student might do 1+2, another might do 3+4, and another might do 5+6. When all the students in a classroom complete their calculations, they can leave. Then, the blackboard will be erased and a new class of students will come in and work on the same problem, but with different values.

Each student entering a class automatically knows what problem they'll be solving, but they don't know what values they'll be working with. The blackboard in each classroom is initially blank, so students go to a central blackboard that contains values for the entire school. This central blackboard is much larger than the blackboards in the classrooms, but because of the long hallway, it takes a great deal of time for students to read its values. Figure 3 depicts the relationship between classes, classrooms, students, notebooks, and blackboards.

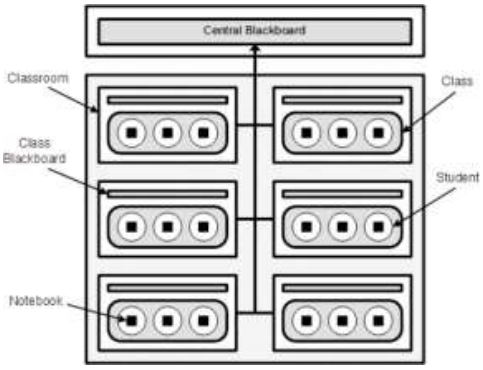

Figure 3: Students solving math problems.

For most math problems, each student will go to the central blackboard only twice — once to read the values for their problem, and once to write down their final answer. Because the central blackboard is so far away, students do their actual solving using their notebooks and classroom blackboards. Once all of the final answers are on the central blackboard, the school day is over.

Students in different classes can't talk to one another, so the students in class 1 won't know when the students in class 2 have finished. The only way to be certain that a class has finished is when the school day ends. It's important to make the distinction between a classroom and a class. A classroom is a physical area with a blackboard. A class is a group of students that occupy a classroom. As one class leaves a classroom, another can enter.

To keep things organized, each class has an identifier that distinguishes it from every other class. Each student has two identifiers: one that distinguishes it from every other student in the class, and one that distinguishes it from every other student in the school. As an example, a student may have a class ID of 12 and a school ID of 638.

Kernel Execution on a Device

In my analogy, the school corresponds to an OpenCL device and the math problem represents the kernel. Each student corresponds to a work-item and each class corresponds to a work-group. A classroom corresponds to a compute unit (processing core); and just as each classroom can be occupied by a class, each compute unit can be occupied by a work-group. Figure 4 depicts this.

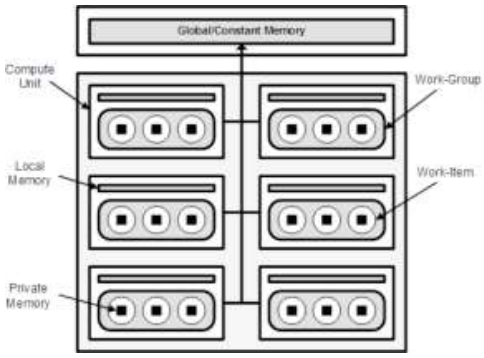

Figure 4: The OpenCL device model.

Identification numbers play a large role in OpenCL, and each work-item has two IDs: a global ID and a local ID. The global ID identifies the work-item among all other work-items executing the kernel. The local ID identifies the work-item among other work-items in the work-group. Work-items in different work-groups may have the same local ID, but they'll never have the same global ID. Returning to the school analogy, a work-item's local ID corresponds to a student's class ID and a work-item's global ID corresponds to the student's school ID.

Now let's talk about memory. The OpenCL device model identifies four address spaces:

1. Global memory: Stores data for the entire device.
2. Constant memory: Similar to global memory, but is read-only.
3. Local memory: Stores data for the work-items in a work-group.
4. Private memory: Stores data for an individual work-item.

In my analogy, the central blackboard corresponds to global memory, which can be read from and written to by both the host and the device. When the host application transfers data to the device, the data is stored in global memory. Similarly, when the host reads data from a device, the data comes from the device's global memory. This memory is commonly the largest memory region on an OpenCL device, but it's also the slowest for work-items to access.

Work-items can access local memory much faster (~100x) than they can access global/constant memory; and local memory blocks correspond to the blackboards in each classroom. Local memory isn't nearly as large as global/constant memory, but because of the access speed, it's a good place for work-items to store their intermediate results. Just as students in the same class can work together at the classroom blackboard, work-items in the same work-group can access the same block of local memory.

The private memory in an OpenCL device corresponds to the notebook each student uses to solve a math problem. Each work-item has exclusive access to its private memory and it can access this memory faster than it can access local memory or global/constant memory. But this address space is much smaller than any other address space, so it's important not to use too much of it.

When I started using OpenCL, I wondered how many work-items could be generated for a kernel. As I hope this analogy has made clear, you can generate as many work-items and work-groups you like. However, if the device only contains *M* compute units and *N* work-items per work-group, only *MN* work-items will execute the kernel at any given time.

A Simple Example

Now that you have a conceptual understanding of OpenCL's host applications and kernels, it's time to look at real code. I've provided two source files: add_numbers.c and add_numbers.cl. The first contains code for a simple host application and the second contains code for a kernel that adds a series of numbers together. This discussion won't present the OpenCL API in any depth, but will focus on how the code relates to the material discussed earlier.

Example Host Application

In general, the first step of a host application is to obtain a `cl_device_id` for each device that will execute a kernel. The add_numbers application accesses the first GPU device associated with the first platform. This is shown in the following code:

```
clGetPlatformIDs(1, &platform, NULL);
clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
```

Here, the `platform` structure identifies the first platform identified by the OpenCL runtime. A platform identifies a vendor's installation, so a system may have an NVIDIA platform and an AMD platform. The `device` structure corresponds to the first accessible device associated with the platform. Because the second parameter is `CL_DEVICE_TYPE_GPU`, this device must be a GPU.

Next, the application creates a context containing only one device — the `device` structure created earlier.

```
context = clCreateContext(NULL, 1, &device, NULL, NULL, &err);
```

After creating the context, the application creates a program from the source code in the add_numbers.cl file. Specifically, the code reads the file's content into a `char` array called `program_buffer`, and then calls `clCreateProgramWithSource`.

```
program = clCreateProgramWithSource(context, 1,
          (const char**)&program_buffer, &program_size, &err);
```

Once the program is created, its source code must be compiled for the devices in the context. The function that accomplishes this is `clBuildProgram`, and the following code shows how it's used in the add_numbers application:

```
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
```

The fourth parameter accepts options that configure the compilation. These are similar to the flags used by gcc. For example, you can define a macro with the option `-DMACRO=VALUE` and turn off optimization with `-cl-opt-disable`.

After a `cl_program` has been compiled, kernels can be created from its functions. The following code creates a `cl_kernel` from the function called `add_numbers`:

```
kernel = clCreateKernel(program, "add_numbers", &err);
```

Before the application can dispatch this kernel, it needs to create a command queue to a target device. With the right configuration, a command queue can support out-of-order kernel execution and/or profiling, which allows us to measure the time taken for a kernel's execution. The following code, however, creates a `cl_command_queue` that does not support profiling or out-of-order-execution:

```
queue = clCreateCommandQueue(context, device, 0, &err);
```

At this point, the application has created all the data structures (device, kernel, program, command queue, and context) needed by an OpenCL host application. Now, it deploys the kernel to a device with the following code:

```
global_size = 8;
local_size = 4;
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global_size, &local_size, 0, NULL, NULL);
```

Of the OpenCL functions that run on the host, `clEnqueueNDRangeKernel` is probably the most important to understand. Not only does it deploy kernels to devices, it also identifies how many work-items should be generated to execute the kernel (`global_size`) and the number of work-items in each work-group (`local_size`).

In this example, the kernel is executed by eight work-items divided into two work-groups of four work-items each. Returning to my analogy, this corresponds to a school containing eight students divided into two classrooms of four students each.

Example Kernel

A practical application may generate thousands or even millions of work-items, but for the simple task of adding 64 numbers, eight work-items will suffice. The program file add_numbers.cl contains a function called `add_numbers` that performs this operation. Like all kernel functions, it returns `void` and its name is preceded by the `__kernel` identifier.

The kernel has 64 values to add together and eight work-items with which to add them. After each work-item computes its sum of eight values, these partial results will be added together to form a sum for the entire group. In the end, the kernel will return two sums — one for each work-group executing the kernel.

When a work-item starts executing `add_numbers`, it begins by determining which region of global memory contains the values it needs to access. This is accomplished with the following code:

```
global_addr = get_global_id(0) * 2;
```

The first line of code provides the work-item with its global ID, which distinguishes it from all the other executing work-items. With this ID, it computes `global_addr`, the address in global memory from which it should load data.

Next, the work-item loads its global data into private memory. Returning to my analogy, this is like a student going to the school-wide blackboard and copying values into a notebook. The following code shows how this works.

```
input1 = data[global_addr];
input2 = data[global_addr+1];
sum_vector = input1 + input2;
```

In the kernel code, `input1, input2,` and `sum_vector` all have the data type `float4`. This is a vector type and it's similar to an array of four `floats`, but there's one significant difference: When you operate on a `float4`, all four `floats` are operated upon in the same clock cycle. If the target device supports vector operations, the last line of this code will have it perform four floating-point additions simultaneously.

Each work-item stores its final sum to local memory, which corresponds to a classroom blackboard in my analogy. This is accomplished as follows:

```
local_addr = get_local_id(0);
local_result[local_addr] = sum_vector.s0 + sum_vector.s1 +
                           sum_vector.s2 + sum_vector.s3;
```

The first line tells the work-item where it can store the sum of its eight values. The second line computes the sum and places it in local memory.

Each work-group contains four work-items, so each block of local memory contains four partial sums. To add these partial sums together, one work-item is designated to read the values for the group and arrive at a group result by adding them together. This is accomplished with the following code:

```
if(get_local_id(0) == 0) {
   sum = 0.0f;
   for(int i=0; i<get_local_size(0); i++) {
      sum += local_result[i];
   }
   group_result[get_group_id(0)] = sum;
}
```

The `group_result` data is stored in global memory. This is important because the host application can only read values stored in this address space. In the case of add_numbers, the application reads the two sums in `group_result`, adds them together, and checks the kernel's output against the correct answer. The host application completes its operation by printing the result and deallocating the structures it used to perform OpenCL processing.

Conclusion

With great power comes great complexity. OpenCL, with its wealth of features, makes it possible to code routines capable of executing on devices ranging from graphics cards to supercomputers. But to take full advantage of OpenCL, you need to have a thorough understanding of host applications and kernels.

This article has presented two analogies intended to ease the learning process, but in the end, it's the code that matters. The OpenCL API takes time and effort to understand, but once you've ascended the learning curve, you'll be able to tap into computing performance that exceeds anything a regular C/C++ programmer could hope for.

— Matthew Scarpino is a software consultant in the San Francisco Bay area. He is the author of the upcoming book OpenCL in Action from Manning Publications. This article was derived from an early draft of the book.