Chapter No. 2
"OpenCL Architecture"
In this package, you will find:
A Biography of the authors of the book
A preview chapter from the book, Chapter NO.2 "OpenCL Architecture"
A synopsis of the book’s content
Information on where to buy this book

About the Authors

**Ravishekhar Banger** calls himself a "Parallel Programming Dogsbody". Currently he is a specialist in OpenCL programming and works for library optimization using OpenCL. After graduation from SDMCET, Dharwad, in Electrical Engineering, he completed his Masters in Computer Technology from Indian Institute of Technology, Delhi. With more than eight years of industry experience, his present interest lies in General Purpose GPU programming models, parallel programming, and performance optimization for the GPU. Having worked for Samsung and Motorola, he is now a Member of Technical Staff at Advanced Micro Devices, Inc. One of his dreams is to cover most of the Himalayas by foot in various expeditions. You can reach him at ravibanger@gmail.com

**Koushik Bhattacharyya** is working with Advanced Micro Devices, Inc. as Member Technical Staff and also worked as a software developer in NVIDIA®. He did his M.Tech in Computer Science (Gold Medalist) from Indian Statistical Institute, Kolkata, and M.Sc in pure mathematics from Burdwan University. With more than ten years of experience in software development using a number of languages and platforms, Koushik's present area of interest includes parallel programming and machine learning.

For More Information:
We would like to take this opportunity to thank "PACKT publishing" for giving us an opportunity to write this book.

Also a special thanks to all our family members, friends and colleagues, who have helped us directly or indirectly in writing this book.
OpenCL Programming by Example

This book is designed as a concise introduction to OpenCL programming for developers working on diverse domains. It covers all the major topics of OpenCL programming and illustrates them with code examples and explanations from different fields such as common algorithm, image processing, statistical computation, and machine learning. It also dedicates one chapter to Optimization techniques, where it discusses different optimization strategies on a single simple problem.

Parallel programming is a fast developing field today. As it is becoming increasingly difficult to increase the performance of a single core machine, hardware vendors see advantage in packing multiple cores in a single SOC. The GPU (Graphics Processor Unit) was initially meant for rendering better graphics which ultimately means fast floating point operation for computing pixel values. GPGPU (General purpose Graphics Processor Unit) is the technique of utilization of GPU for a general purpose computation. Since the GPU provides very high performance of floating point operations and data parallel computation, it is very well suited to be used as a co-processor in a computing system for doing data parallel tasks with high arithmetic intensity.

Before NVIDIA® came up with CUDA (Compute Unified Device Architecture) in February 2007, the typical GPGPU approach was to convert general problems' data parallel computation into some form of a graphics problem which is expressible by graphics programming APIs for the GPU. CUDA first gave a user friendly small extension of C language to write code for the GPU. But it was a proprietary framework from NVIDIA and was supposed to work on NVIDIA’s GPU only.

With the growing popularity of such a framework, the requirement for an open standard architecture that would be able to support different kinds of devices from various vendors was becoming strongly perceivable. In June 2008, the Khronos compute working group was formed and they published OpenCL1.0 specification in December 2008. Multiple vendors gradually provided a tool-chain for OpenCL programming including NVIDIA OpenCL Drivers and Tools, AMD APP SDK, Intel® SDK for OpenCL application, IBM Server with OpenCL development Kit, and so on. Today OpenCL supports multi-core programming, GPU programming, cell and DSP processor programming, and so on.

In this book we discuss OpenCL with a few examples.

What This Book Covers

Chapter 1, Hello OpenCL, starts with a brief introduction to OpenCL and provides hardware architecture details of the various OpenCL devices from different vendors.

Chapter 2, OpenCL Architecture, discusses the various OpenCL architecture models.

For More Information:
www.packtpub.com/opencl-programming-by-example/book
Chapter 3, *OpenCL Buffer Objects*, discusses the common functions used to create an OpenCL memory object.

Chapter 4, *OpenCL Images*, gives an overview of functions for creating different types of OpenCL images.

Chapter 5, *OpenCL Program and Kernel Objects*, concentrates on the sequential steps required to execute a kernel.

Chapter 6, *Events and Synchronization*, discusses coarse grained and fine-grained events and their synchronization mechanisms.

Chapter 7, *OpenCL C Programming*, discusses the specifications and restrictions for writing an OpenCL compliant C kernel code.

Chapter 8, *Basic Optimization Techniques with Case Studies*, discusses various optimization techniques using a simple example of matrix multiplication.

Chapter 9, *Image Processing and OpenCL*, discusses Image Processing case studies. OpenCL implementations of Image filters and JPEG image decoding are provided in this chapter.

Chapter 10, *OpenCL-OpenGL Interoperation*, discusses OpenCL and OpenGL interoperation, which in its simple form means sharing of data between OpenGL and OpenCL in a program that uses both.

Chapter 11, *Case studies – Regressions, Sort, and KNN*, discusses general algorithm-like sorting. Besides this, case studies from Statistics (Linear and Parabolic Regression) and Machine Learning (K Nearest Neighbourhood) are discussed with their OpenCL implementations.

For More Information:
www.packtpub.com/opencl-programming-by-example/book
Heterogeneous computing is all about exploiting computing resources in a platform to maximize performance. Many developers have begun to realize that heterogeneous multi-core computer systems can provide significant performance opportunities to a range of applications. OpenCL specification targets expert programmers who want to run their code on various heterogeneous platforms. Unlike NVIDIA® CUDA framework, which is capable of running only on NVIDIA devices, library writers can provide acceleration on any parallel hardware device using OpenCL. Thus OpenCL provides a low-level hardware abstraction and a programming model to support a variety of hardware architectures.

OpenCL describes a hierarchy of models to describe the OpenCL programming framework:

- Platform model
- Memory model
- Execution model
- Programming model

For More Information:
www.packtpub.com/opencl-programming-by-example/book
**Platform model**

In heterogeneous computing, knowledge about the architecture of the targeted device is critical to reap the full benefits of the hardware. We had discussed the hardware architectures from AMD, Intel, and NVIDIA in Chapter 1, *Hello OpenCL*. Though we will briefly discuss about the hardware from different vendors, we suggest you to take a deeper look at the underlying platform on which you will be working. In this section we will describe the OpenCL Platform model and map the AMD, NVIDIA, and Intel hardware architectures to the OpenCL Platform definitions.

An OpenCL Platform model consists of a host connected to one or more devices like CPU's, GPU's or hardware accelerators like DSP's. Each OpenCL device consists of one or more compute units, which in turn is further divided into one-to-many processing elements. Computations on a device that is the actual kernel (work item) execution occurs within these processing elements. We just coined the term **work item**. This we will discuss later in this chapter when we discuss about the OpenCL Execution model.

We will now discuss the four different architectures from different device vendors and try to map their architecture to the OpenCL Platform model. In the next diagram we have shown four different OpenCL architectures and their mappings to the Platform models.

---

**For More Information:**

AMD A10 5800K APUs

A10 5800K APU has four AMD x86_64 processor cores, which forms the host. Its graphics processor includes as many as six SIMD engines, each with four texture units and sixteen thread processors. There are four ALUs in each thread processor, adding up to 384 total shader cores or processing elements. The following diagram shows the relation of the Trinity APU to the OpenCL Platform model:

![Diagram showing the relation of the Trinity APU to the OpenCL Platform model.](image)

This platform has two devices, the CPU device and the GPU device. The x86 CPU device is also the host. The OpenCL Platform model can be mapped as having four compute units and each having one processing element. The graphics processor connected to the host CPU also forms an OpenCL device of type GPU. The six SIMD engines form the six GPU device compute units in the platform. Each of the six compute elements have sixteen thread processors, each having four processing elements. In all there are 384 processing elements or shader cores in this platform for the GPU device.

For More Information:
www.packtpub.com/opencl-programming-by-example/book
**AMD Radeon™ HD 7870 Graphics Processor**

HD 7870 discrete card is a graphics processor based on the AMD GCN architecture. This compute device can be connected to any x86/x86_64 platform. The CPU forms the host and the GPU forms the device in the OpenCL platform. AMD Radeon HD 7870 GPU has a total of twenty compute units. With each compute unit having 64 shader cores a total of 1280 processing elements are there.

![AMD Radeon™ HD 7870 Architecture diagram](https://example.com)

**NVIDIA® GeForce® GTX 680 GPU**

The NVIDIA GTX 680 graphics card architecture diagram is shown as follows. There are eight blocks of compute units in this graphics processor. Also referred to as the Kepler Architecture, the compute units are called the Streaming Multiprocessors-X (SMX). This SMX compute unit is an advance over previous architectures and has 192 CUDA cores or processing elements. This is shown in the following diagram:

![NVIDIA GeForce GTX 680 Architecture diagram](https://example.com)

For More Information:  
www.packtpub.com/opencl-programming-by-example/book
Intel® IVY bridge

The IVY bridge architecture is very similar to the sandy bridge architecture discussed in Chapter 1, Hello OpenCL. The CPU device can be mapped as any x86 CPU as discussed in the AMD A10 5800K APU’s section. In the case of Intel hardware’s, the GPU device offers what is called as the Execution Units (EUs). These numbers vary across different SOC solutions provided by Intel. In Intel HD 4000 there are sixteen EUs. These sixteen EUs form the processing elements or sixteen compute unit, that is each execution unit is a compute unit.

For More Information:
www.packtpub.com/opencl-programming-by-example/book
For all the preceding OpenCL hardware architectures, which we have discussed till now an OpenCL application consists of a host program that runs according to the models native to the host platform. The host application submits commands to the device to which executes the OpenCL kernels on the processing elements in a compute device. The OpenCL specification describes the functions to create memory objects called buffers and run OpenCL kernels on an OpenCL device. The host queues the thread launch. Before processing the data the host application writes to device, and finally after processing it reads from device. It would be good if the data transfer bandwidth between the host and the device is good enough to hide the data transfer bottleneck with the highly parallel computing power of the device. Some computers may use a shared memory architecture between the host computer (CPU) and the OpenCL device (say a GPU). In such cases the memory transfer bottlenecks may be minimal.

**Platform versions**

The OpenCL is designed to support devices with different capabilities under a single platform. This includes devices which conform to different versions of the OpenCL specification. While writing an OpenCL based application one needs to query the implementation about the supported version in the platform. There are mainly two different types of version identifiers to consider:

- **Platform Version**: Indicates the version of the OpenCL runtime supported.
- **Device Version**: Indicates the device capabilities and attributes. The conformant version info provided cannot be greater than platform version.

**Query platforms**

Now let’s write an OpenCL program to get the platform details. Use the get_platform_property example in this chapter.

The OpenCL standard specifies API interfaces to determine the platform configuration. To query the platform versions and details of the OpenCL implementation, the following two APIs are used:

```c
cl_int clGetPlatformIDs (cl_uint num_entries,
                         cl_platform_id *platforms,
                         cl_uint *num_platforms);
cl_int clGetPlatformInfo(cl_platform_id platform,
                         cl_platform_info param_name,
                         size_t param_value_size,
                         void *param_value);
```

For More Information:
www.packtpub.com/opencl-programming-by-example/book
void *param_value,
    size_t *param_value_size_ret);

clGetPlatformIDs is used to obtain the total number of platforms available in the system. There can be more than one platform. If you install two OpenCL runtimes, one from AMD APP SDK and the other Intel OpenCL runtime for the CPU, you should be able to see two platforms in the system. Usually you don't want to pre-allocate the memory for storing the platforms. Before getting the actual platform, an application developer should query for the number of OpenCL implementations available in the platform. This is done using the following OpenCL call:

    clError = clGetPlatformIDs(0, NULL, &num_platforms);

This call returns the total number of available platforms. Once we have obtained the number of available platforms we can allocate memory and query for the platform IDs for the various OpenCL implementations as follows:

    platforms = (cl_platform_id *)malloc
               (num_platforms*sizeof(cl_platform_id));
    clError = clGetPlatformIDs (num_platforms, platforms, NULL);

Once the list of platforms is obtained, you can query for the platform attributes in a loop for each platform. In the example we have queried the following parameters using the API clGetPlatformInfo:

    CL_PLATFORM_NAME
    CL_PLATFORM_VENDOR
    CL_PLATFORM_VERSION
    CL_PLATFORM_PROFILE
    CL_PLATFORM_EXTENSIONS

Example:

    clError = clGetPlatformInfo (platforms[index], CL_PLATFORM_NAME, 1024,
                                 &queryBuffer, NULL);

In the get_device_property example where we get device properties, we default to the first available platform and query the device property for all the devices in default platform obtained. Take a look at the get_device_property example for this chapter.

    clError = clGetPlatformIDs(1, &platform, &num_platforms);

Note the difference in the calls to clGetPlatformIDs in the two examples discussed.
In this section we just wrote a small program to print the platform details. Take a look at how we allocate memory for platforms and how we get the details of the platform. As an exercise try to install multiple OpenCL implementations in your platform and see how many OpenCL platforms are enumerated by the function \texttt{clGetPlatformIDs}.

Multiple OpenCL implementations can be installed in the platform. You would question how would the application pick the appropriate runtime. The answer is OpenCL \textbf{Installable Client Driver (ICD)}. We will study this more in a later section.

**Query devices**

We shall now continue with getting the attributes and resource limitations of an OpenCL device. In the last program we were able to print all the platform information available. In this example we shall try to enhance the existing code to print some basic device attributes and resource information for the first available platform. We will implement a function \texttt{PrintDeviceInfo()}, which will print the device specific information. The following two OpenCL APIs are used in the example:

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

cl_int clGetDeviceInfo (cl_device_id device,
    cl_device_info param_name,
    size_t param_value_size,
    void *param_value,
    size_t *param_value_size_ret);
```

In the same way as we did for platforms, we first determine the number of devices available, and then allocate memory for each device found in the platform.

```c
clError = clGetDeviceIDs (platform,
    CL_DEVICE_TYPE_ALL,
    0, NULL, &num_devices);
```

The above call gives the number of available device of \texttt{CL\_DEVICE\_TYPE\_ALL}. You can otherwise use \texttt{CL\_DEVICE\_TYPE\_CPU} or \texttt{CL\_DEVICE\_TYPE\_GPU}, if you want to list the number of available CPU or GPU devices.
To understand better we we have added the `PrintDeviceInfo` function:

```c
void PrintDeviceInfo(cl_device_id device)
{
    char queryBuffer[1024];
    int queryInt;
    cl_int clError;
    clError = clGetDeviceInfo(device, CL_DEVICE_NAME,
                                sizeof(queryBuffer),
                                &queryBuffer, NULL);
    printf("CL_DEVICE_NAME: %s\n", queryBuffer);
    queryBuffer[0] = '\0';
    clError = clGetDeviceInfo(device, CL_DEVICE_VENDOR,
                                sizeof(queryBuffer), &queryBuffer,
                                NULL);
    printf("CL_DEVICE_VENDOR: %s\n", queryBuffer);
    queryBuffer[0] = '\0';
    clError = clGetDeviceInfo(device, CL_DRIVER_VERSION,
                                sizeof(queryBuffer), &queryBuffer,
                                NULL);
    printf("CL_DRIVER_VERSION: %s\n", queryBuffer);
    queryBuffer[0] = '\0';
    clError = clGetDeviceInfo(device, CL_DEVICE_VERSION,
                                sizeof(queryBuffer), &queryBuffer,
                                NULL);
    printf("CL_DEVICE_VERSION: %s\n", queryBuffer);
    queryBuffer[0] = '\0';
    clError = clGetDeviceInfo(device, CL_DEVICE_MAX_COMPUTE_UNITS,
                                sizeof(int), &queryInt, NULL);
    printf("CL_DEVICE_MAX_COMPUTE_UNITS: %d\n", queryInt);
}
```

Note that each of the `param_name` associated with `clGetDeviceInfo` returns a different data type. In the routine `PrintDeviceInfo` you can see that the `CL_DEVICE_MAX_COMPUTE_UNITS` `param_name` returns an integer type The `CL_DRIVER VERSION` `param_name` returns a character buffer.

The preceding function prints the following information about the device:

```
CL_DEVICE_NAME
CL_DEVICE_VENDOR
CL_DRIVER_VERSION
CL_DEVICE_VERSION
CL_DEVICE_MAX_COMPUTE_UNITS
```

For More Information:
www.packtpub.com/opencl-programming-by-example/book
OpenCL Architecture

Following is the maximum number of compute units for different types of platforms when you query for the GPU type device:

For APU like processors:

- **AMD A10 5800K** - 6

AMD trinity has 6 SIMD engines (compute units) and each has 64 processing elements.

- **INTEL HD 4000** - 16

Intel HD 4000 has 16 compute units and each is a single thread processor.

For discrete graphics:

- **NVIDIA GTX 680** - 8

The NVIDIA GTX 680 has a total of eight Compute units; each compute unit has 192 processing elements.

- **AMD Radeon HD 7870** - 32

The AMD Radeon HD 7870 GPU has 32 compute units and each has 64 processing elements.

It is not the case that if you have more compute units in the GPU device type, the faster the processor is. The number of compute units varies across different computer architectures and across different hardware vendors. Sometimes even within the vendors there are different families like the NVIDIA Kepler and Fermi architectures or the AMD Radeon HD 6XXX and Radeon HD 7XXX Architecture. The OpenCL specification is targeted at programming these different kinds of devices from different vendors. As an enhancement to the sample program print all the device related attributes and resource sizes for some of the `param_name` instances listed as follows:

- `CL_DEVICE_TYPE`
- `CL_DEVICE_MAX_CLOCK_FREQUENCY`
- `CL_DEVICE_IMAGE_SUPPORT`
- `CL_DEVICE_SINGLE_FP_CONFIG`

Besides these there are many more device attributes which can be queried. Take a look at the different `param_name` instances provided in the OpenCL specification 1.2, table 4.3. You should try out all the `param_name` instances and try to understand each device property.
### Execution model

The two main execution units in OpenCL are the kernels and the host program. The kernels execute on the so-called OpenCL device and the host program runs on the host computer. The main purpose of the host program is to create and query the platform and device attributes, define a context for the kernels, build the kernel, and manage the execution of these kernels.

On submission of the kernel by the host to the device, an $N$ dimensional index space is created. $N$ is at least 1 and not greater than 3. Each kernel instance is created at each of the coordinates of this index space. This instance is called as the "work item" and the index space is called as the **NDRange**. In the following screenshot we have shown the three scenarios for 1, 2 and 3 dimensional NDRange:

![Diagram of NDRange](image)

In the **saxpy** example which we discussed in the previous chapter, we have taken a global size of 1024 and a local size of 64. Each work item computes the corresponding:

\[
C[\text{local id}] = \alpha A[\text{local id}] + B[\text{local id}];
\]

For More Information:

A total of sixteen work groups are spawned. When the `clEnqueueNDRange` function is executed, a 1 Dimensional NDRange is created for the `saxpy_kernel` function. The explanation of `clEnqueueNDRange` function is given in the next section. Since in `saxpy` every data can be calculated independently, all the work items can run in a parallel way. We divide the problem of 1024 element saxpy into work groups, so that a group of contiguous elements can work on a separate OpenCL capable compute unit.

**NDRange**

An NDRange is the kernel execution index in an N-dimensional index space. The values which N can take are 1, 2, or 3. An NDRange value is given by an array of integers of length N specifying the index's extent in each dimension. Starting OpenCL 1.2 an offset index value can also be specified for each dimension, which gives the starting offset for an NDRange. If this offset is not specified then its value is 0 by default in each dimension. The extent of a work group is specified by `local_work_size` in the `clEnqueueNDRangeKernel` function below. Global ID and Local ID are N tuple values. The `global_work_size` function defines the total number of work items, which can be spawned for the OpenCL kernel. The global ID components are values in the range from offset X, to X plus the `global_work_size` function in their corresponding dimensions.

A group of work items are organized in OpenCL work groups. Take a look at the following diagram of a 2D NDRange. The work groups provide a coarse-grained decomposition of the index space. Each work group is assigned a unique ID with the same number of dimensions as the global index space used to define the work items. Every work item in a work group is assigned a unique local ID, so that every work item can be uniquely addressed during the kernel execution within a work group scope. Similarly work items are also assigned a unique global ID in the NDRange and can be uniquely addressed during the kernel execution.

Work groups are also assigned work group IDs. This is also an array of N integers, where each integer defines the number of work groups in each dimension. The work groups' IDs are assigned in the same manner as it is done for assigning global IDs. See equation 2 later in the section. Every work item has an associated work group ID and a local ID. It's easy to calculate the global ID of the work item, when we are given a work group ID and a local-ID. See equation 1 later in this section. Each work item can be identified in two ways; global index, and work group index plus a local index in each of its respective dimensions.

Let's explain the following with an equation: $N=2$ NDRange:
We will be using the following terms for defining the Execution model:

- **work-item**: It is the individual kernel execution instance
- **work-group**: It is a group of work items form a work group
- **global-id**: A unique global ID given to each work item in the global NDRange
- **local-id**: A unique local ID given to each work item within a work group

Consider a \((12,12)\) NDRange as shown in the following figure. Each of the smallest box is a work item. As you can see there are twelve of them in each row and there are twelve such rows.

In the preceding diagram the global size is defined by \((12,12) \sim (G_x, G_y)\). The extent of \(G_x\) and \(G_y\) is 0 to 11. The total number of work items is given by the product of \(G_x\) and \(G_y\), which amounts to a total of 144 work items.

The size of each work group is \((4, 4) \sim (S_x, S_y)\). The extent of \(S_x\) and \(S_y\) is 0 to 3. The total number of work items in a work group is given by the product of \(S_x\) and \(S_y\). In this example there are sixteen work items in the work group.

For More Information:  
www.packtpub.com/opencl-programming-by-example/book
OpenCL Architecture

From the extent of the global work items \((G_x, G_y)\) and the extent of the local work items \((S_x, S_y)\), we can compute the number of work groups \((W_x, W_y)\) in the NDRange.

Each work item is identified by its global ID \((g_x, g_y)\) or local ID \((s_x, s_y)\). The work items in a work group belong to a work group ID \((w_x, w_y)\) defined in the following equation 3. Similarly the global ID can be computed using a combination of local ID \((s_x, s_y)\) and work group ID \((w_x, w_y)\), as shown in the equation:

\[
(g_x, g_y) = (w_x \cdot S_x + s_x, w_y \cdot S_y + s_y) \quad (1)
\]

The number of work groups can be computed using the equation:

\[
(W_x, W_y) = \left(\frac{G_x}{S_x}, \frac{G_y}{S_y}\right) \quad (2)
\]

The work-group ID for a work item is computed the using equation:

\[
(w_x, w_y) = \left(\frac{(g_x - s_x)}{S_x}, \frac{(g_y - s_y)}{S_y}\right) \quad (3)
\]

Till now we have discussed about the work item, work group, local ID, and global ID. All these values can be determined inside a kernel execution at runtime using the built-in functions, which are listed as follows:

- `get_global_id(int dim);`
- `get_local_id(int dim);`
- `get_num_groups(int dim);`
- `get_group_size(int dim);`
- `get_group_id(int dim);`

The NDRange execution space is defined by the OpenCL API. The associated parameters should all be created in an OpenCL context as follows:

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

For More Information:

This function enqueue's a command to execute a kernel on the device associated with the command_queue function. Of all the OpenCL functions that run on the host, clEnqueueNDRangeKernel is the most important to understand. Not only does it deploys kernels to devices, it also specifies how many work items should be generated to execute the kernel (global_work_size) and the number of work items in each work group (local_work_size). The following list represents certain objects:

- **command_queue**: Every command_queue is associated with one device. kernel will be enqueued for execution on this device. The command_queue object is created using the clCreateCommandQueue function.
- **kernel**: It refers to an OpenCL kernel object. This kernel object would have been created using the OpenCL program object.
- **work_dim**: It specifies the dimension of the NDRange (index space). The value can be 1, 2 or 3.
- **global_work_offset**: This is a size_t pointer to the work_dim elements. If set to NULL all the values in each dimension take the default value as 0. Otherwise this is used to calculate the global ID of a work item.
- **global_work_size**: This is a size_t pointer to the work_dim elements, which specifies the extent of the global work items in every dimensions.
- **local_work_size**: This is also a size_t pointer to the work_dim elements and specifies the extent of local work items in every dimension.
- **event_wait_list and num_events_in_wait_list**: The event_wait_list object contains handles to events, which an OpenCL implementation will wait for before enqueueing this command.
- **event**: Every enqueued command returns an OpenCL event object that is the reference to the command in the queue. Here the kernel's execution handle is returned in the event pointer. This cl_event object can be used later on for reference to the execution status.

The OpenCL supports two of these execution models; the data parallel programming model and the task parallel programming model. The clEnqueueNDRangeKernel function is a kind of data parallel execution model, the task parallel programming model will be discussed in Chapter 5, OpenCL Program and Kernel Objects.

We just coined the term "enqueues a command", let's explain what a queue has to do with the OpenCL. Before that, let's discuss the OpenCL context.
OpenCL context

A context defines the entire OpenCL environment, including the devices, the program objects, the OpenCL kernels, memory objects, command queues, and so on. A context can be associated with multiple devices or with only one device. The OpenCL context associated with command queue and the kernel should be the same. They cannot be from different contexts.

Before we can create a context we must first query the OpenCL runtime to determine which vendor platforms are available in the system. After you have selected a vendor platform, the first step is to initialize the OpenCL implementation in order to create a context. The rest of the OpenCL work like creating devices and memory, compiling, and running programs is performed within this context. A context can have a number of associated devices, which can be either of CPU or GPU or both, and, within a context. Contexts in the OpenCL are referenced by a `cl_context` object, which must be initialized using the following OpenCL API:

```c
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)
```

The following is the list of few contexts of the OpenCL along with its description:

- **properties**: It is a list of name and its corresponding value. The name is the context property name like `CL_CONTEXT_PLATFORM` and this is followed by the property value. An example of the same is as follows:

  ```c
  cl_context_properties props[3] =
  {
    CL_CONTEXT_PLATFORM,
    (cl_context_properties)platforms,
    0
  };
  ```

  One can add more property values based on the requirements of the application.
• **num_devices**: It is the number of devices one wants to associate with the context. The devices pointer should have at least num_devices, cl_device_id instance.

• **devices**: It is a pointer to a num_devices list of cl_device_id instances, which will be associated with the context.

• **errcode_ret**: The error code returned by the OpenCL implementation associated with a call to this function.

• **pfn_notify**: It is a function pointer to the callback function, which an application can register. The underlying OpenCL implementation will call this function to asynchronously report errors for context creation. If set to NULL then no callback function is registered. The prototype of a callback function is as follows:

  ```c
  void OpenCL_Context_Callback(const char *errinfo,
                             const void *private_info,
                             size_t cb, void *user_data);
  ```

• **user_data**: This is the pointer to the data, which will be passed to the callback function if registered by the application. If no callback function is registered this should be set to NULL.

---

**OpenCL command queue**

The OpenCL command queue is an object where OpenCL commands are queued to be executed by the device. The command queue is created for every usable OpenCL device for kernel execution. One can have multiple command queues for different tasks in applications. This way an application developer can run tasks independently on different command queues. We will discuss about the various synchronization mechanisms using multiple command queues in Chapter 6, Events and Synchronization. The following code snippet creates a command queue and a write (clEnqueueWriteBuffer), and NDRange execution of the kernel commands are queued on to the device:

```c
cl_command_queue command_queue = clCreateCommandQueue(context, device_list[0], 0, &clStatus);
clStatus = clEnqueueWriteBuffer(command_queue, A_clmem, CL_TRUE, 0,
                               VECTOR_SIZE * sizeof(float), A, 0, NULL, NULL);
clStatus = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, &global_size, &local_size, 0, NULL, NULL);
```
OpenCL Architecture

The host program creates this command queue. The snapshot of the queue anytime shall give you the list of enqueued commands. These commands can be of data transfer, or kernel execution commands or barriers within the command queue. The host enqueues these commands to the command queue. Each command or task is associated with an OpenCL event. These events can be used as a synchronization mechanism to coordinate execution between the host and the device.

There can be multiple queues associated within a context. They can dispatch commands independently and concurrently with no explicit mechanism to synchronize between them.

Queues can be in-order of the execution queues. The commands are dequeued in **first in first out (FIFO)** manner. Hence application can send commands to the queue and be ensured that they execute in order.

Out of order command queues are also supported by the OpenCL. The commands are issued in order, but do not wait for the previous command to complete before the next command executes. We will discuss more about this in Chapter 5, *OpenCL Program and Kernel Objects*.

**Memory model**

The OpenCL Memory model guarantees a relaxed memory consistency between devices. This means that different work items may see a different view of global memory as the computation progresses. This leads to a bigger challenge for the developers to partition data and splitting computation tasks into different work items. Synchronization is required to ensure data consistency within the work items of a work group. One needs to make sure that the data the work item is accessing is always correct. This makes the application developers task a little complicated to write applications with relaxed consistency, and hence explicit synchronization mechanisms are required.

The x86/x86_64 CPU cache coherent architecture is different from the OpenCL relaxed memory architecture. In cache coherent systems, data that resides in the local processor caches is guaranteed to be consistent across processors. The programmer need not worry about the data partitioning in cache coherent architectures. This results in a lot of memory bandwidth at the back of the cache, and makes the task of an application programmer easier. The OpenCL Memory model scales well across cache coherent memory architectures also. An OpenCL programmer must have knowledge of partitioning the data across his application work load, to achieve the highest performance in massively parallel heterogeneous systems. The standard defines four distinct memory regions. Each region can be accessed by the work items executing a kernel. The following are the different types of memory.

---

**For More Information:**

Global memory

Every OpenCL device has an associated global memory. This is the largest size memory subsystem. Memory accesses can be coalesced, to ensure contiguous memory reads and thereby increasing the performance. All the work items in all the work groups can read or write into this memory buffer. This memory can be cached depending on the OpenCL device. Take a look at the following OpenCL kernel prototype:

```c
__kernel
void histogram_kernel(__global const uint* data,
                      __local uchar* sharedArray,
                      __global uint* binResultR,
                      __global uint* binResultG,
                      __global uint* binResultB)
```

The `__global` or `global` keyword identifies this buffer region. This memory region is device wide and changes made in this region are visible to all the work items in the NDRange.

Constant memory

An OpenCL device has a region of memory called the constant memory, which is initialized by the host. This is similar to creating an OpenCL buffer with `CL_MEM_READ_ONLY` flag. This is the region of memory that remains constant throughout the execution time of the kernel.

Local memory

For high performance every OpenCL device has an associated local memory. This is the memory closest to the OpenCL processing element. Every work item in a work group can use this buffer and is shared amongst them that is if one work item modifies a local memory then the changes are visible across all the work items in a work group. As shown in the diagram the local memory is associated with one OpenCL compute unit. This means that the work items in a work group should all run on one compute unit. The `__local` or `local` keyword identifies this memory region.

For More Information:
www.packtpub.com/opencl-programming-by-example/book
Private memory

Memory region or processing element scratch registers are all referred to as the private region. This region of memory is used by the OpenCL device compiler to allocate all the local variables in the kernel code. Any modifications done to this memory region are not visible to the other work items. As shown in the following diagram every processing element has a private memory. This is the default memory attribute in an OpenCL kernel:

Based on the underlying architecture the work items in a given work group execute concurrently on the processing elements of a single compute unit. This means that one work group is associated with one compute unit of the hardware in OpenCL. This is because most of the hardware architectures have high speed memory local to the compute unit. In the context of OpenCL we refer to private memory as high speed memory.
The private memory can be shared among all the work items in the work group. For example in some graphics architectures, every compute unit has a large private memory say of the size 64 KB. When all the work items in the work group run on the device this 64 KB is shared among all the work items. For example a work group of size 64 work items will allocate 1 KB of private memory for each work item. This makes the application programmer create the OpenCL kernels, which use small number of registers and the hardware scheduler should be able to launch many work items or wave fronts at a time.

**OpenCL ICD**

The OpenCL function `clGetPlatformIDs` is used to determine the different OpenCL implementations available in the platform. There can be multiple OpenCL implementations installed in the system. Let's define an OpenCL platform.

An OpenCL platform is a host computing machine and a collection of heterogeneous devices managed by OpenCL implementations, which allow an application to share hardware resources and execute kernels on different devices in the platform. Devices from different vendors will have their own OpenCL runtimes in the platform. Let's consider a system with an AMD graphics card and an NVIDIA graphics card. Now an AMD OpenCL implementation is not going to work on NVIDIA OpenCL devices. Remember only the code is portable not the underlying OpenCL runtime. So how does an application solve this problem of using the multiple OpenCL runtimes or use multiple platforms. The answer is OpenCL ICD.

For More Information:

**What is an OpenCL ICD?**

The OpenCL **Installable Client Driver (ICD)** is a means of allowing multiple OpenCL implementations to co-exist and applications to select between them at runtime. With this it is now the applications responsibility for querying which OpenCL platform is present in the system and which one the application should use, instead of just requesting the default like we did in our first few example wherein we chose the first available platform as default.

---

In the preceding diagram, an OpenCL application is linked with the OpenCL ICD library. At runtime this ICD shared library (*OpenCL.dll* in windows and *libOpenCL.so* in Linux) will query the registry and load the appropriate shared library as selected by the application. An application may want to use both the platforms available. The application developer can create a context for each device in the platform, and appropriately execute his algorithm on the device. It is not possible to pass device buffers between two different OpenCL contexts. It is the host applications responsibility to share, transfer, and synchronize data consumption between two contexts.

---

For More Information:

Application scaling

A multithreaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more cores will automatically execute the program in less time than a GPU with fewer cores. This is important since we can see here two levels of nested data parallelism or data parallelism nested within task parallelism. The upper level parallelism partitions a given problem into blocks of threads. Each block of thread will run on a compute unit, for example, a SIMD engine in the AMD APUs. Beyond this high level parallelism there is lower level parallelism, where a group of threads run cooperatively within the thread block. Each of these threads runs on the processing elements of the compute unit.

Less cores more time, Courtesy NVIDIA®

For More Information:
www.packtpub.com/opencl-programming-by-example/book
OpenCL Architecture

**Summary**

In this chapter we started with the discussion of OpenCL Platform model, and briefly explained the various hardware from different vendors and tried to map the OpenCL terminologies to the devices which we discussed. We also discussed the Execution and Memory model of OpenCL. This chapter forms the foundation for any OpenCL programmer.

Till now we discussed about the OpenCL architecture and the OpenCL ecosystem in general. From here on we will study the OpenCL objects such as, buffers, images, programs, kernels, and so on in detail. In the next chapter we will start our discussion about the OpenCL buffers, and the mechanisms of data transfer between the host and the OpenCL devices.

For More Information:  
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.