# Chapter 3

# Introduction to OpenCL

This chapter introduces OpenCL and describes the anatomy and architecture of the OpenCL API.

## 3.1 What is OpenCL?

The Open Computing Language (OpenCL) is an open and royalty-free parallel computing API designed to enable GPUs and other coprocessors to work in tandem with the CPU, providing additional raw computing power. As a standard, OpenCL 1.0 was released on December 8, 2008, by The Khronos Group, an independent standards consortium.

Developers have long sought to divide computing problems into a mix of concurrent subsets, making it feasible for a GPU to be used as a math coprocessor working with the CPU to better handle general problems. The potential of this heterogeneous computing model was encumbered by the fact that programmers could only choose proprietary programming languages, limiting their ability to write vendor-neutral, cross-platform applications. Proprietary implementations such as NVIDIA's CUDA limited the hardware choices of developers wishing to run their application on another system without having to retool it.

### 3.1.1 Benefits of OpenCL

A primary benefit of OpenCL is substantial acceleration in parallel processing. OpenCL takes all computational resources, such as multi-core CPUs and GPUs, as peer computational units and correspondingly allocates different levels of memory, taking advantage of the resources available in the system. OpenCL also complements the existing OpenGL® visualization API by sharing data structures and memory locations without any copy or conversion overhead.

A second benefit of OpenCL is cross-vendor software portability. This low-level layer draws an explicit line between hardware and the upper software layer. All the hardware implementation specifics, such as drivers and runtime, are invisible to the upper-level software programmers through the use of high-level abstractions, allowing the developer to take advantage of the best hardware without having to reshuffle the upper software infrastructure. The change from proprietary programming to open standard also contributes to the acceleration of general computation in a cross-vendor fashion.

# 3.2 Anatomy of OpenCL

The OpenCL development framework is made up of three main parts:

- Language specification
- Platform layer API
- Runtime API

## 3.2.1 Language Specification

The language specification describes the syntax and programming interface for writing kernel programs that run on the supported accelerator (GPU, multi-core CPU, or DSP). Kernels can be precompiled or the developer can allow the OpenCL runtime to compile the kernel program at runtime.

The OpenCL programming language is based on the ISO C99 specification with added extensions and restrictions. Additions include vector types and vector operations, optimized image access, and address space qualifiers. Restrictions include the absence of support for function pointers, bit-fields, and recursion. The C language was selected as the first base for the OpenCL language due to its prevalence in the developer community. To ensure consistent results across different platforms, OpenCL C also provides a well-defined IEEE 754 numerical accuracy for all floating point operations and a rich set of built-in functions. For complete language details on the language, see the OpenCL specification.

## 3.2.2 Platform API

The platform-layer API gives the developer access to software application routines that can query the system for the existence of OpenCL-supported devices. This layer also lets the developer use the concepts of device context and work-queues to select and initialize OpenCL devices, submit work to the devices, and enable data transfer to and from the devices.

## 3.2.3 Runtime API

The OpenCL framework uses contexts to manage one or more OpenCL devices. The runtime API uses contexts for managing objects such as command queues, memory objects, and kernel objects, as well as for executing kernels on one or more devices specified in the context.

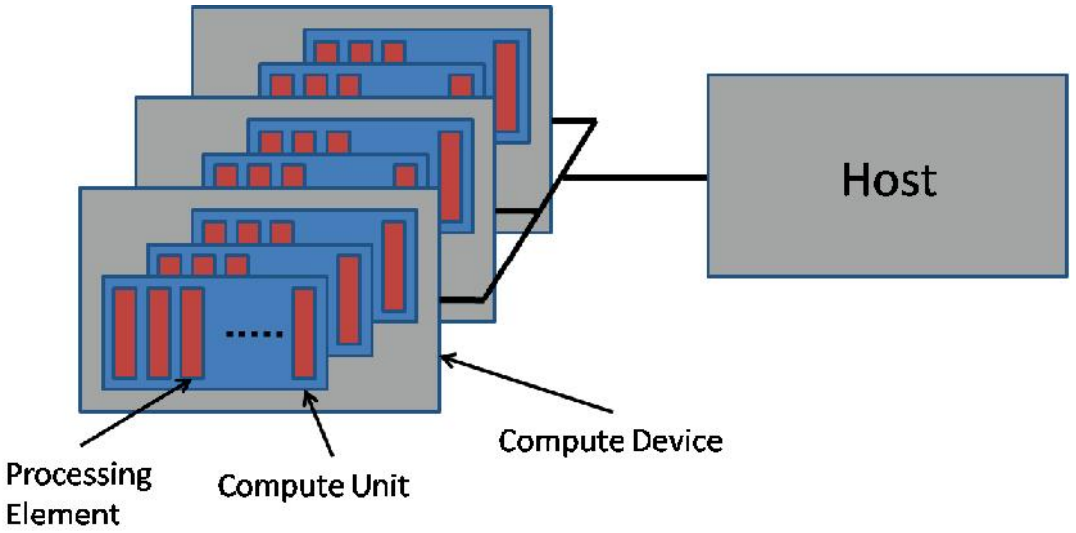
# 3.3 OpenCL Architecture

### 3.3.1 The Platform Model

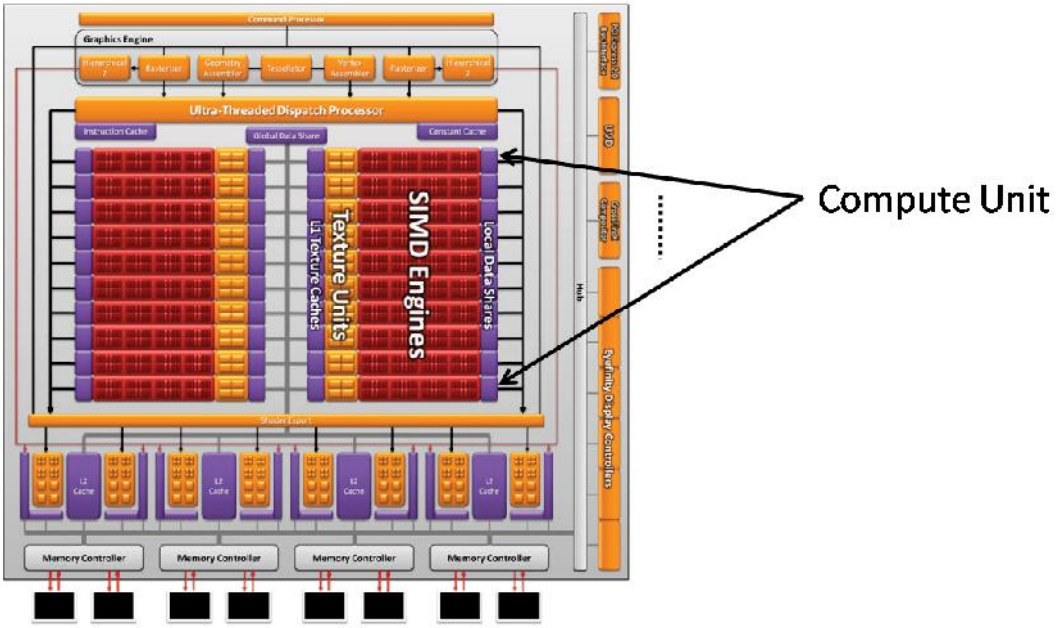
The OpenCL platform model is defined as a host connected to one or more OpenCL devices. Figure 3–1 OpenCL Platform Model shows the platform model comprising one host plus multiple compute devices, each having multiple compute units, each of which have multiple processing elements.

A host is any computer with a CPU running a standard operating system. OpenCL devices can be a GPU, DSP, or a multi-core CPU. An OpenCL device consists of a collection of one or more compute units (cores). A compute unit is further composed of one or more processing elements. Processing elements execute instructions as SIMD (Single Instruction, Multiple Data) or SPMD (Single Program, Multiple Data). SPMD instructions are typically executed on general purpose devices such as CPUs, while SIMD instructions require a vector processor such as a GPU or vector units in a CPU.

Figure 3–1 OpenCL Platform Model



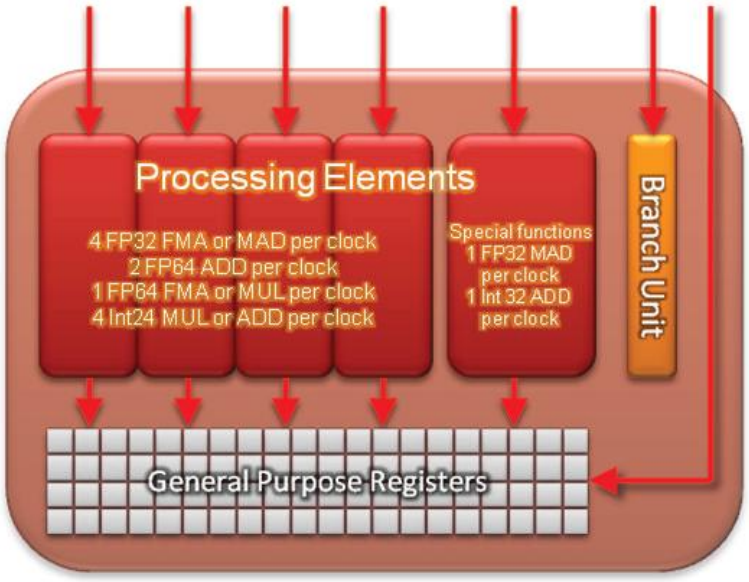
The following depiction of the ATI Radeon™ HD 5870 GPU architecture illustrates a compute device construct. The ATI Radeon HD 5870 GPU is made up of 20 SIMD units, which translates to 20 compute units in OpenCL:





Each SIMD unit contains 16 stream cores, and each stream core houses five processing elements. Thus, each compute unit in the ATI Radeon HD 5870 has 80 (16 × 5) processing elements.

Figure 3–2 Stream Core Housing Five Processing Elements




## 3.3.2 The Execution Model

The OpenCL execution model comprises two components: kernels and host programs. Kernels are the basic unit of executable code that runs on one or more OpenCL devices. Kernels are similar to a C function that can be data- or task-parallel. The host program executes on the host system, defines devices context, and queues kernel execution instances using command queues. Kernels are queued in-order, but can be executed in-order or out-of-order.

### 3.3.2.1 Kernels

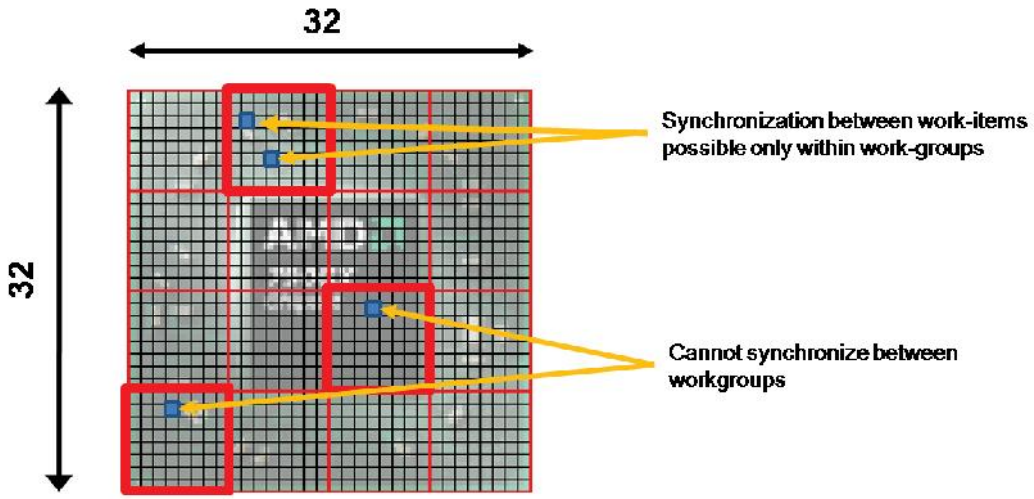
OpenCL exploits parallel computation on compute devices by defining the problem into an N-dimensional index space. When a kernel is queued for execution by the host program, an index space is defined. Each independent element of execution in this index space is called a work-item. Each work-item executes the same kernel function but on different data. When a kernel command is placed into the command queue, an index space must be defined to let the device keep track of the total number of work-items that require execution. The N-dimensional index space can be N=1, 2, or 3. Processing a linear array of data would be considered N=1; processing an image would be N=2, and processing a 3D volume would be N=3.

Processing a 1024x1024 image would be handled this way: The global index space comprises a 2-dimensional space of 1024 by 1024 consisting of 1 kernel execution (or work-item) per pixel with a total of 1,048,576 total executions. Within this index space, each work-item is assigned a unique global ID. The work-item for pixel x=30, y=22 would have global ID of (30,22).

OpenCL also allows grouping of work-items together into work-groups, as shown in the following figure. The size of each work-group is defined by its own local index space. All work-items in the same work-group are executed together on the same

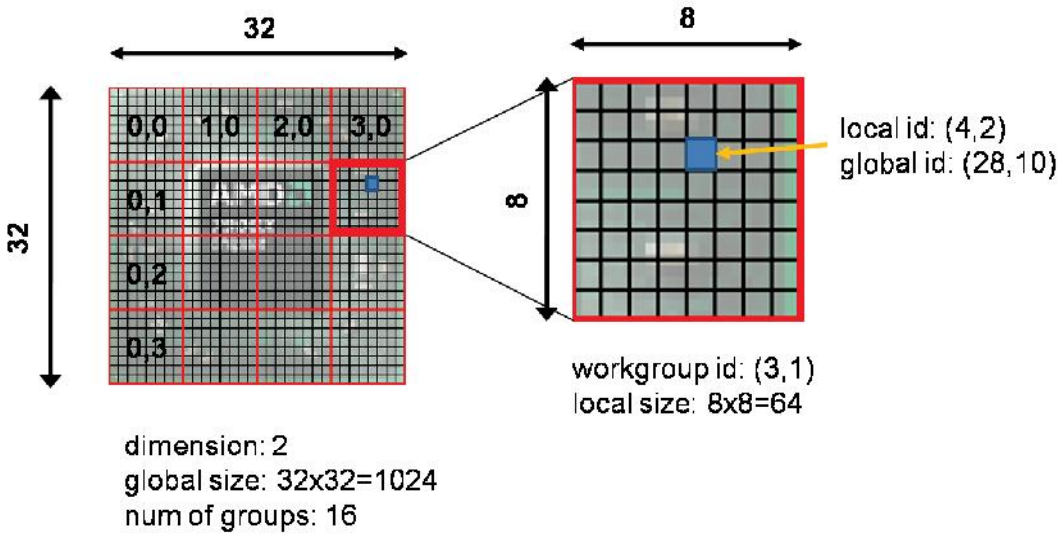
device. The reason for executing on one device is to allow work-items to share local memory and synchronization. Global work-items are independent and cannot by synchronized. Synchronization is only allowed between the work-items in a work-group.

Figure 3–3 Grouping Work-items Into Work-groups




The following example shows a two-dimensional image with a global size of 1024 (32x32). The index space is divided into 16 work-groups. The highlighted work-group has an ID of (3,1) and a local size of 64 (8x8). The highlighted work-item in the work-group has a local ID of (4,2), but can also be addressed by its global ID of (28,10).

Figure 3–4 Work-group Example



The following example illustrates how a kernel in OpenCL is implemented. In this example, each element is squared in a linear array. Normally, a scalar function would be required with a simple *for* loop iterating through the elements in the array and then squaring it. The data-parallel approach is to read an element from the array *in parallel*, perform the operation in parallel, and write it to the output. Note that the code segment on the right does not have a *for* loop: It simply reads the index value for the particular kernel instance, performs the operation, and writes the output.

Table 3–1 Simple Example of Scalar Versus Parallel Implementation

| Scalar C Function | Data-Parallel Function |
|---|---|
| `void square(int n, const float *a, float *result)`<br>`{`<br>`  int i;`<br>`  for (i=0; i<n; i++)`<br>`  result[i] = a[i]*a[i];`<br>`}` | `kernel void dp_square`<br>`  (global const float *a, global float *result)`<br>`{`<br>`  int id= get_global_id(0);`<br>`  result[id] = a[id]*a[id];`<br>`}`<br>`// dp_square execute over "n" work-items` |

The OpenCL execution model supports two categories of kernels: OpenCL kernels and native kernels.

OpenCL kernels are written in the OpenCL C language and compiled with the OpenCL compiler. All devices that are OpenCL-compliant support execution of OpenCL kernels.

Native kernels are extension kernels that could be special functions defined in application code or exported from a library designed for a particular accelerator. The OpenCL API includes functions to query capabilities of devices to determine if native kernels are supported.

If native kernels are used, developers should be aware that the code may not work on other OpenCL devices.

### 3.3.2.2 Host Program

The host program is responsible for setting up and managing the execution of kernels on the OpenCL device through the use of context. Using the OpenCL API, the host can create and manipulate the context by including the following resources:

- **Devices** — A set of OpenCL devices use by the host to execute kernels.
- **Program Objects** — The program source or program object that implements a kernel or collection of kernels.
- **Kernels** — The specific OpenCL functions that execute on the OpenCL device.
- **Memory Objects** — A set of memory buffers or memory maps common to the host and OpenCL devices.

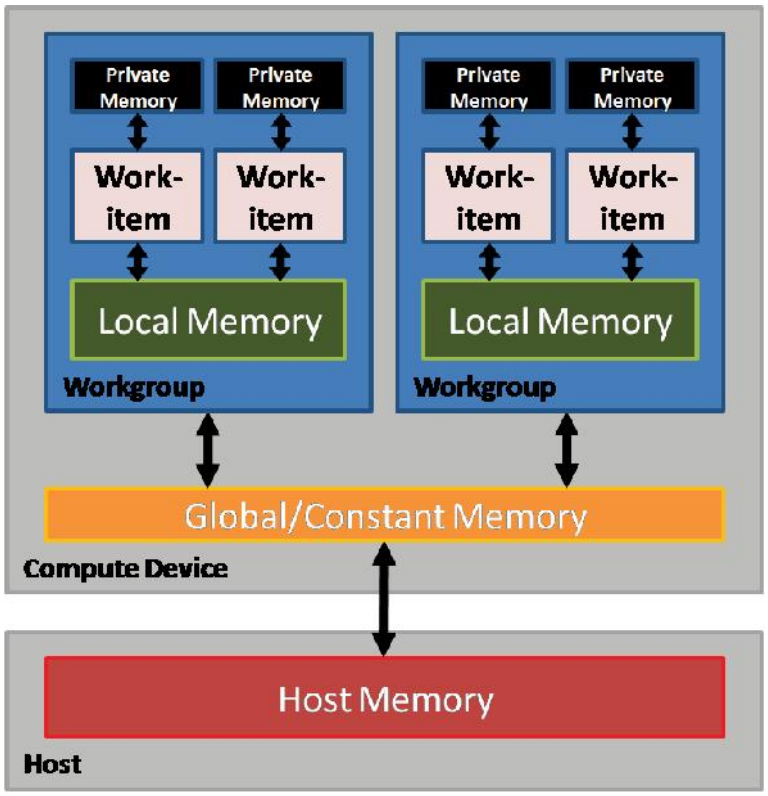
After the context is created, command queues are created to manage execution of the kernels on the OpenCL devices that were associated with the context. Command queues accept three types of commands:

- **Kernel execution commands** run the kernel command on the OpenCL devices.
- **Memory commands** transfer memory objects between the memory space of the host and the memory space of the OpenCL devices.
- **Synchronization commands** define the order in which commands are executed.

Commands are placed into the command queue in-order and execute either in-order or out-of-order. In in-order mode, the commands are executed serially as they are placed onto the queue. In out-of-order mode, the order the commands execute is based on the synchronization constraints placed on the command.

### 3.3.3 The Memory Model

Since common memory address space is unavailable on the host and the OpenCL devices, the OpenCL memory model defines four regions of memory accessible to work-items when executing a kernel. The following figure shows the regions of memory accessible by the host and the compute device:



Global memory is a memory region in which all work-items and work-groups have read and write access on both the compute device and the host. This region of memory can be allocated only by the host during runtime.

Constant memory is a region of global memory that stays constant throughout the execution of the kernel. Work-items have only read access to this region. The host is permitted both read and write access.

Local memory is a region of memory used for data-sharing by work-items in a work-group. All work-items in the same work-group have both read and write access.

Private memory is a region that is accessible to only one work-item.

In most cases, host memory and compute device memory are independent of one another. Thus, memory management must be explicit to allow the sharing of data between the host and the compute device. This means that data must be explicitly moved from host memory to global memory to local memory and back. This process works by enqueuing read/write commands in the command queue. The commands placed into the queue can either be blocking or non-blocking. Blocking means that the host memory command waits until the memory transaction is complete before continuing. Non-blocking means the host simply puts the command in the queue and continues, not waiting until the memory transaction is complete.