MANNING PUBLICATIONS

Introducing OpenCL

Green Paper from

OpenCL in Action

Matthew Scarpino
Softbound print: 2012 | 475 pages
ISBN: 9781617290176

*This green paper is taken from the book **OpenCL in Action** from Manning Publications. The author explains the inner workings of OpenCL and uses a card game analogy to explain OpenCL operation.*

In May of 2005, Intel and AMD released the first multi-core processors for desktop computers: Intel's Pentium D and AMD's Athlon 64 X2. Since then, PC chip designers have embraced the concept of improving performance by adding more cores. Intel has released processors with four and six cores and has announced eight-core devices in the near future. IBM, working with Sony and Toshiba, has released the Cell Broadband Engine, a nine-core processor that serves as the brain of the PlayStation 3. Not to be outdone, AMD has announced development of a multi-core processor with one major core devoted to graphics.

The computing public continues to enjoy the benefits of multi-core devices, but software developers aren't smiling as widely. When it comes to coding, more cores usually means more difficulty, and one of the largest concerns is *portability*. Back in the day of single-core processing, source code written in C could be compiled and executed on nearly every platform under the sun. But, today, code that targets an Intel multi-core processor won't run on a Cell Broadband Engine, and compiling regular C code for a graphics processing unit (GPU) is out of the question. If you want to program the hundreds of cores in one of Nvidia's Fermi processors, you have to learn Nvidia's proprietary language.

This is where Open Computing Language (OpenCL) comes in. OpenCL provides a standard set of routines that can be executed on any OpenCL-compliant device. (And, there are many compliant devices—Intel, AMD, Nvidia, IBM, and Apple are all part of the OpenCL Working Group.) This means that, if you write code in OpenCL, you don't have to worry about which company designed the processor or how many cores it contains. Your high-speed routines will compile and execute on AMD's multi-core processors, IBM's Cell Broadband Engine, and Nvidia's Fermi processors.

The goal of this green paper is to provide a basic overview of OpenCL. The discussion will start by focusing on its advantages and operation and then proceed to a complete application. But first, it's important to understand OpenCL's origin. Corporations have spent a great deal of time developing this language, and once you see why, you'll have a better idea why learning about OpenCL is worth your own time.

The dawn of OpenCL

The x86 architecture enjoys a dominant position in the world of personal computing, but there is no prevailing architecture in the fields of graphical and high-performance computing. Further, despite their common purpose, there is little similarity between Nvidia's Fermi processors, AMD's Evergreen processors, and IBM's Cell Broadband

Engine. Each of these devices has its own instruction set, and before OpenCL, if you wanted to program them, you'd have to learn all three languages.

Enter Apple. As you know, Apple Inc. produces insanely popular consumer electronic products. But, Apple does not make its own processors. Instead, it carefully selects devices from other companies. If Apple chooses a graphics processor from Company A for its new gadget, then Company A will see a tremendous rise in market share and developer interest. This is why everyone is *so* nice to Apple.

Important events in OpenCL and multi-core computing history

2001—IBM releases POWER4, the first multi-core processor.

2005—First multi-core processors for desktop computers released: AMD's Athlon 64 X2 and Intel's Pentium D.

June 2008—The OpenCL Working Group forms as part of the Khronos Group.

December 2008—The OpenCL Working Group releases version 1.0 of the OpenCL specification.

April 2009—Nvidia releases OpenCL SDK for Nvidia graphic cards.

August 2009—ATI (now AMD) releases OpenCL SDK for ATI graphic cards. Apple includes OpenCL support in its Mac OS 10.6 (Snow Leopard) release.

2010—The OpenCL Working Group releases version 1.1 of the OpenCL specification.

In 2008, Apple turned to its vendors and asked, "Why don't we make a common interface so that developers can program these devices without having to learn multiple languages?" If anyone else had raised this question, cutthroat competitors like Nvidia, AMD, Intel, and IBM might have laughed. But no one laughs at Apple. It took some time, but everyone put their heads together and they produced the first draft of OpenCL later that year.

To manage OpenCL's progress and development, Apple and its friends formed the OpenCL Working Group. This is part of the Khronos Group, a consortium of companies whose aim is to advance graphics and graphical media. Since its formation, the OpenCL Working Group has released two formal specifications. OpenCL version 1.0 was released in 2008 and OpenCL version 1.1 was released in 2010. OpenCL 2.0 is planned for 2012.

This section has explained why businesses think highly of OpenCL, but I wouldn't be surprised if you're still on the fence. The next section, however, explains the technical merits of OpenCL in much greater depth. As you read, I hope you'll understand better the advantages of OpenCL as compared to traditional programming languages.

Why OpenCL?

You may hear OpenCL referred to as its own separate language, but this isn't accurate. The OpenCL standard defines a set of data types, data structures, and functions that augment C and C++. Developers have created OpenCL ports for Java and Python, but the standard only requires that OpenCL frameworks provide libraries in C and C++.

So here's the million-dollar question: what can you do with OpenCL that you can't do with regular C and C++? It will take this entire book to answer this question in full, but for now, let's look at three of OpenCL's chief advantages: portability, standardized vector processing, and parallel programming.

Portability

One of the reasons that Java is so popular is its famous motto, "Write once, run everywhere." With Java, you don't have to write code multiple times for different operating systems. As long as the operating system supports a Java Virtual Machine (JVM), your code will run.

OpenCL adopts a similar philosophy, but a more suitable motto would be "Write once, run on anything." Every vendor that provides OpenCL-compliant hardware also provides tools that compile and link OpenCL code to run on the hardware. This means you can build an OpenCL application once and run it on any compliant hardware, whether it's a multi-core processor or a graphics card. This is a great advantage over the regular state of affairs, in which you have to learn multiple vendor-specific languages to program vendor-specific hardware.

But there's more to this advantage than just running on any type of compliant hardware. OpenCL applications commonly execute on multiple devices at once, and these devices don't have to be the same make and model. As long as all the devices are OpenCL compliant, the applications will run. This is simply impossible with regular C/C++ programming, in which an executable can only target one device.

This is a subtle point, so let me give you a concrete example. Let's say you have a multi-core processor from AMD, a graphics card from Nvidia, and a PCI-connected accelerator from IBM. Normally, you'd never be able to build an application that accesses the three systems at once. After all, each requires a separate compiler and linker. But a single OpenCL program can contain executable code for all three devices. This means not only can you get all your hardware working together, but you only need one program to do it. And, if you connect more compliant devices, you'll have to rebuild the program but you won't have to rewrite your code.

Standardized vector processing

Standardized vector processing is one of the greatest advantages of OpenCL, but before I explain why, I need to define precisely what I'm talking about. The term *vector* may be used in one of three different (though essentially similar) ways:

1. *Physical or geometric vector*—An entity with a magnitude and direction. This is used frequently in physics to identify force, velocity, heat transfer, and so on. In graphics, vectors are employed to identify directions.
2. *Mathematical vector*—An ordered, one-dimensional collection of elements. This is distinguished from a two-dimensional collection of elements, called a matrix.
3. *Computational vector*—A data structure that contains multiple elements of the same data type. During a vector operation, each element (called a component) is operated upon in the same clock cycle.

This last usage is important to OpenCL because high-performance processors operate on multiple values at once. If you've heard the terms *superscalar processor* or *vector processor*, this is the type of device being referred to. Nearly all modern processors are capable of processing vectors, but ANSI C/C++ doesn't define any basic vector data types. This may seem odd, but there's a clear problem: vector instructions are usually vendor specific. Intel processors use SSE extensions, Nvidia devices require PTX instructions, and IBM devices rely on AltiVec instructions to process vectors. These instruction sets have nothing in common.

But with OpenCL, you can code your vector routines once and run them on any compliant processor. When you compile your application, Nvidia's OpenCL compiler will produce PTX instructions. An IBM compiler for OpenCL will produce AltiVec instructions. If you intend to make your high-performance application available on multiple platforms, coding with OpenCL will save you a great deal of time.

Parallel programming

If you've ever coded large-scale applications, you're probably familiar with the concept of concurrency, in which a single processing element shares its resources among processes and threads. OpenCL includes aspects of concurrency, but one of its great advantages is that it enables *parallel programming*. Parallel programming assigns tasks to multiple processing elements to be performed at the same time.

In OpenCL parlance, these tasks are called *kernels*. The processing elements that execute kernels are called *devices*, and the devices are accessed through a container called a *context*. Figure 1 shows how hosts interact with kernels and devices.

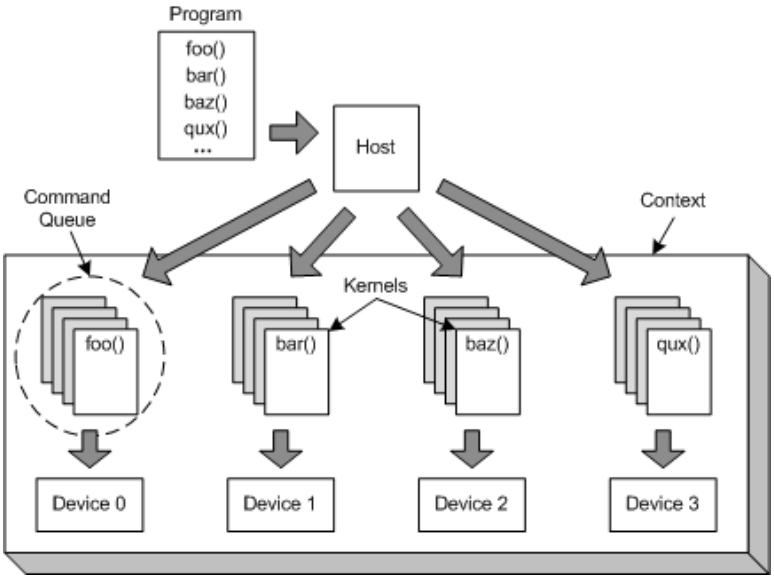

Figure 1 Kernel distribution among OpenCL-compliant devices

Each kernel corresponds to a specifically-annotated OpenCL function, and the host selects these functions from a kernel container called a *program*. Next, the host associates argument data with each kernel and places the kernel or kernels into structures called *command queues*. Each command queue is then sent to its intended device for execution.

An OpenCL application can configure different devices to perform different tasks and each task can operate on different sets of data. In other words, OpenCL provides full *task-parallelism*. This is a clear advantage over other parallel programming toolsets, which only enable *data-parallelism*. In a data-parallel system, each device receives the same instructions but operates on different sets of data.

Figure 1 depicts how OpenCL accomplishes task-parallelism between devices, but it doesn't show what's happening inside each device. Most OpenCL-compliant devices consist of more than one processing element, which means there's an additional level of parallelism internal to each device.

Portability, vector processing, and parallel programming make OpenCL more powerful than regular C and C++, but with this greater power comes greater complexity. In any practical OpenCL application, you have to create a number of different data structures and coordinate their operation. It can be hard to keep everything straight, but the next section presents an analogy that I hope will give you a clearer perspective.

Analogy: OpenCL processing and a game of cards

When I first started learning OpenCL, I was overwhelmed by all the strange data structures: platforms, contexts, devices, programs, kernels, and command queues. I found it hard to remember what these objects do and how they interact, so I created an analogy. The operation of an OpenCL application is like a game of cards (particularly poker). This may seem odd at first, but please allow me to explain.

In a card game, the dealer sits at a table with one or more players and deals a set of cards to each. Once each player has a full hand, they analyze the cards and decide what further actions to take. The players don't interact with each other; instead, they make requests to the dealer for additional cards or an increase in the stakes. The dealer handles each request in turn, and once the game is over, the dealer takes control. That is, they either take the winnings or distribute the pot to the winner.

In this analogy, the dealer represents the OpenCL host, each player represents a device, the card table represents a context, and each card represents a kernel. Each player's hand represents a command queue. Table 1 clarifies how the steps of a card game resemble the operation of an OpenCL application.

Table 1 Comparison of OpenCL operation with a card game

Card game	OpenCL application
The dealer identifies the players at a card table.	The host identifies devices and places them in a context.
The dealer selects cards from a deck and deals them to each player. Each player's cards form a hand.	The host selects kernels from a program. It adds each device's kernels to a command queue and sends each queue to the devices.
Each player looks at their hand and decides what actions to take.	Each device processes the kernel or kernels in its command queue.
The host responds to players' requests during the game.	The host receives events from the devices and invokes event handling routines.
The game ends and the host looks at each player's hand to determine who won.	Once the devices are finished, the host receives and processes the output data.

Just in case the analogy still seems hard to understand, figure 2 depicts a card game with four players, each of whom receives a hand with four cards. If you compare figures 1 and 2, I hope the analogy will become clearer.

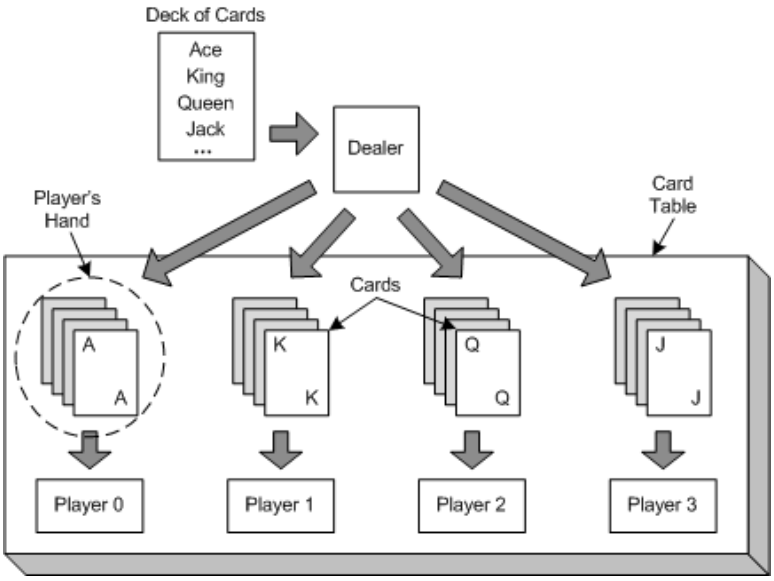

Figure 2 Pictorial representation of a game of cards

This analogy provides an intuitive understanding of OpenCL, but it has a number of flaws. Five of the most significant flaws are as follows:

- The analogy doesn't mention platforms. A platform is a data structure that identifies a vendor's implementation of OpenCL. They're important because you can only access devices through a platform. For example, to access an Nvidia device, you first have to find the Nvidia platform.
- A card dealer doesn't select which players sit at the table. However, an OpenCL host selects which devices should be placed in a context.
- A card dealer can't deal the same card to multiple players, but an OpenCL host can send the same kernel to multiple devices.
- The analogy doesn't mention data or how it's partitioned for OpenCL devices. OpenCL devices usually contain multiple processing elements, and each element may process a subset of the input data. The host

sets the dimensionality of the data and identifies the size of work groups.

- In card games, dealers commonly deal cards in a round-robin fashion. OpenCL sets no constraints on how kernels are distributed to devices.

Writing OpenCL code is the primary goal, and the next section provides a first taste of what OpenCL code looks like.

A first look at an OpenCL application

At this point, you should have a good idea of what OpenCL is intended to accomplish. Hopefully, you also have a basic understanding of how an OpenCL application works. But if you want to know anything substantive about OpenCL, you need to look at source code.

This section will present two OpenCL source files, one for the host processor and one for a device. Both are needed to compute the product of a 4 × 4 matrix and a four-element vector. This operation is central to graphic processing and figure 3 shows what it looks like using matrix-vector notation.

$$\begin{vmatrix} 0.0 & 2.0 & 4.0 & 6.0 \\ 8.0 & 10.0 & 12.0 & 14.0 \\ 16.0 & 18.0 & 20.0 & 22.0 \\ 24.0 & 26.0 & 28.0 & 30.0 \end{vmatrix} \begin{vmatrix} 0.0 \\ 3.0 \\ 6.0 \\ 9.0 \end{vmatrix} = \begin{vmatrix} 84.0 \\ 228.0 \\ 372.0 \\ 516.0 \end{vmatrix}$$

Figure 3 Matrix-vector multiplication

Listing 1 shows what the host code looks like. Notice that the source code is written in the C programming language.

Listing 1 Creating and distributing a matrix-vector multiplication kernel

```
#include <assert.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <CL/cl.h>

int main() {

   cl_platform_id platform;                                    #A
   cl_device_id device;                                        #A
   cl_context context;                                         #A
   cl_command_queue queue;                                     #A
   cl_int error;                                               #A

   cl_program program;                                         #B
   char prog_name[] = "matvec.cl";                             #B
   FILE* prog_handle;                                          #B
   struct stat prog_stat;                                      #B
   cl_kernel kernel;                                           #B
   char kernel_name[] = "matvec_mult";                         #B
   cl_mem mat_buff, vec_buff, res_buff;                        #B

   float mat[16], vec[4];                                      #C
   for(int i=0; i<4; i++) {                                    #C
      vec[i] = i * 3.0;                                        #C
   }                                                           #C
   for(int i=0; i<16; i++) {                                   #C
      mat[i] = i * 2.0;                                        #C
   }                                                           #C

   error = clGetPlatformIDs(1, &platform, NULL);               #D
   assert(error == CL_SUCCESS);
```

```
error = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1,  #E
   &device, NULL);
assert(error == CL_SUCCESS);

context = clCreateContext(NULL, 1, &device, NULL, NULL,  #F
   &error);
assert(error == CL_SUCCESS);

prog_handle = fopen(prog_name, "r");                     #G
assert(prog_handle != NULL);                             #G
stat(prog_name, &prog_stat);                             #G
size_t buff_size = prog_stat.st_size;                    #G
char *buff = (char*)malloc(buff_size);                   #G
fread(buff, buff_size, 1, prog_handle);                  #G
fclose(prog_handle);                                     #G

program = clCreateProgramWithSource(context, 1,          #H
   (const char**)&buff, &buff_size, &error);
assert(error == CL_SUCCESS);
free(buff);

if(clBuildProgram(program, 0, NULL, NULL, NULL,          #I
      NULL) != CL_SUCCESS) {
   char log[1024];
   clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG,
      sizeof(log), log, NULL);
   printf("%s\n", log);
   exit(1);
}

kernel = clCreateKernel(program, kernel_name, &error);     #J
assert(error == CL_SUCCESS);

mat_buff = clCreateBuffer(context, CL_MEM_READ_ONLY |      #K
   CL_MEM_COPY_HOST_PTR, sizeof(float)*16, mat, &error);
assert(error == CL_SUCCESS);

vec_buff = clCreateBuffer(context, CL_MEM_READ_ONLY |      #K
   CL_MEM_COPY_HOST_PTR, sizeof(float)*4, vec, &error);
assert(error == CL_SUCCESS);

res_buff = clCreateBuffer(context, CL_MEM_WRITE_ONLY,      #K
   sizeof(float)*4, NULL, &error);
assert(error == CL_SUCCESS);

error = clSetKernelArg(kernel, 0, sizeof(cl_mem),          #L
   &mat_buff);
assert(error == CL_SUCCESS);
error = clSetKernelArg(kernel, 1, sizeof(cl_mem),          #L
   &vec_buff);
assert(error == CL_SUCCESS);
error = clSetKernelArg(kernel, 2, sizeof(cl_mem),          #L
   &res_buff);
assert(error == CL_SUCCESS);

queue = clCreateCommandQueue(context, device, 0, &error);  #M
assert(error == CL_SUCCESS);

size_t global_size = 4;
size_t local_size = 4;
error = clEnqueueNDRangeKernel(queue, kernel, 1, NULL,     #N
   &global_size, &local_size, 0, NULL, NULL);
assert(error == CL_SUCCESS);

float result[4];
error = clEnqueueReadBuffer(queue, res_buff, CL_TRUE, 0,   #O
   sizeof(float)*4, result, 0, NULL, NULL);
assert(error == CL_SUCCESS);

float correct[4] = {0.0, 0.0, 0.0, 0.0};
for(int i=0; i<4; i++) {
```

```
      correct[0] += mat[i]    * vec[i];
      correct[1] += mat[i+4]  * vec[i];
      correct[2] += mat[i+8]  * vec[i];
      correct[3] += mat[i+12] * vec[i];
   }
   if((result[0] == correct[0]) && (result[1] == correct[1])
      && (result[2] == correct[2]) && (result[3] == correct[3])) {
      printf("Test successful.\n");
   }
   else {
      printf("Test unsuccessful.\n");
   }

   clReleaseMemObject(mat_buff);                              #P
   clReleaseMemObject(vec_buff);                              #P
   clReleaseMemObject(res_buff);                              #P
   clReleaseKernel(kernel);                                   #P
   clReleaseCommandQueue(queue);                              #P
   clReleaseProgram(program);                                 #P
   clReleaseContext(context);                                 #P

   return 0;
}
```

#A Host data structures
#B Program data structures
#C Initialize vector and matrix
#D Identify valid platform
#E Access compliant device
#F Create context
#G Read source file text
#H Create program
#I Build program
#J Create kernel
#K Create buffers to hold input/output
#L Create kernel arguments
#M Create command queue
#N Send command queue to device
#O Obtain result
#P Deallocate resources

This source file is long but straightforward. Most of the code is devoted to creating OpenCL's data structures. Here, the structures obey a simple naming convention: the `cl_context` structure is called `context`, the `cl_platform_id` structure is called `platform`, the `cl_device_id` structure is called `device`, and so on. If you follow this convention, you can copy and paste most of this source file into your own code.

However, the creation of the `cl_program` and the `cl_kernel` structures changes from application to application. In listing 1, the application creates a kernel from a function in the matvec.cl file. More precisely, it reads the characters from the matvec.cl source file into a buffer and calls `clCreateProgramWithSource`, which allocates the `cl_program`. Then it calls `clCreateKernel` to construct the kernel object from one of the functions.

The kernel code in matvec.cl is much shorter than the host code in matvec.c. The single function, `matvec_mult`, performs the matrix-vector multiplication algorithm presented in figure 3.

Listing 2 Performing the dot product on the device

```
__kernel void matvec_mult(__global float4* matrix,
                          __global float4* vector,
                          __global float* result) {

   int i = get_global_id(0);                                  #A
   result[i] = dot(matrix[i], vector[0]);                     #B
}
```

#A Access kernel identifier
#B Perform dot product

Unlike the code in listing 1, this kernel code runs on a device outside the host, specifically a GPU. It contains many terms that aren't defined in the ANSI C standard: keywords like `__kernel` and `__global`, and the `float4` data type. These new features are defined in the OpenCL standard, and the next section explains more about this standard.

The OpenCL standard and extensions

If you search through the primary OpenCL web site at www.khronos.org/opencl, you can download a file called opencl-1.1.pdf. This PDF contains the OpenCL 1.1 standard, which holds a wealth of information about OpenCL. This document not only defines OpenCL's functions and data structures, but also the capabilities required by a vendor's development tools. It sets processing criteria that all compliant devices must meet.

But compliant software and hardware can provide capabilities beyond those defined in the standard. These additional features are made available to OpenCL applications through OpenCL's extension mechanism. There are two types of extensions: those that relate to a vendor's software package (called a platform) and those that relate to a specific device.

Every OpenCL extension has a name whose form depends on the extension's level of acceptance. If an extension has been approved by the OpenCL working group, its name will take the form `cl_khr_<name>`. If it has been released by a vendor but not approved by that working group, the extension's name will be `cl_<vendor>_<name>`.

Let's look at an example. My Linux system supports the platform extension `cl_khr_icd`. This extension relates to software. In particular, it makes it possible for build tools to find the names of vendor-specific OpenCL libraries installed on a system. ICD stands for Installable Client Driver.

Frameworks and software development kits (SDKs)

The code in matvec.c and matvec.cl may look impressive, but the two source files aren't useful unless you can compile them into an OpenCL application. To do this, you need to access the tools in a framework. As defined in the OpenCL standard, a framework consists of three parts:

- *Platform layer*—Makes it possible to access devices and form contexts
- *Runtime*—Enables host applications to send kernels and command queues to devices in the context
- *Compiler*—Builds programs that contain executable kernels

The OpenCL Working Group doesn't provide any frameworks of its own. Instead, vendors of OpenCL-compliant devices have released frameworks as part of their SDKs. At the time of this writing, only two companies provide OpenCL SDKs: Nvidia and AMD. In both cases, the development kits are free and contain the libraries and tools that you'll need to build OpenCL applications. Whether you're targeting Nvidia or AMD devices, installing an SDK is a straightforward process.

Summary

OpenCL is a new, powerful toolset for building parallel programs to run on high-performance processors. With OpenCL, you don't have to learn device-specific languages; you can write your code once and run it on any OpenCL-compliant hardware.

Besides portability, OpenCL provides the advantages of vector processing and parallel programming. In high-performance computing, a vector is a data structure comprising multiple values of the same data type. But unlike other data structures, when a vector is operated upon, all of its values are operated upon at the same time. Parallel programming means that one application controls processing on multiple devices at once. OpenCL can send different tasks to different devices, and this is called task-parallel programming. If used effectively, vector processing and task-parallel programming provide dramatic improvements in computational performance over scalar, single-processor systems.

OpenCL code consists of two parts: code that runs on the host and code that runs on one or more devices. Host code is written in regular C or C++ and is responsible for creating the data structures that manage the host-device communication. The host selects functions, called kernels, to be placed in command queues and sent to the devices. Kernel code, unlike the host code, uses the new data types and functions defined in the OpenCL standard.

With so many new data structures and operations, OpenCL may seem daunting at first. But as you start writing your own code, you'll see that it's not much different from regular C and C++. And once you harness the power of vector-based parallel programming in your own applications, you'll never want to go back to traditional single-core computing.