Chapter No. 1
"Using OpenCL"
In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.1 "Using OpenCL"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Raymond Tay has been a software developer for the past decade and his favorite programming languages include Scala, Haskell, C, and C++. He started playing with GPGPU technology since 2008, first with the CUDA toolkit by NVIDIA and OpenCL toolkit by AMD, and then Intel. In 2009, he decided to submit a GPGPU project on which he was working to the editorial committee working on the "GPU Computing Gems" to be published by Morgan Kauffmann. And though his work didn't make it to the final published work, he was very happy to have been short-listed for candidacy. Since then, he's worked on projects that use GPGPU technology and techniques in CUDA and OpenCL. He's also passionate about functional programming paradigms and their applications in cloud computing which has led him investigating on various paths to accelerate applications in the cloud through the use of GPGPU technology and the functional programming paradigm. He is a strong believer of continuous learning and hopes to be able to continue to do so for as long as he possibly can.

For More Information:

This book could not have been possible without the support of foremost, my wife and my family, as I spent numerous weekends and evenings away from them so that I could get this book done and I would make it up to them soon. Packt Publishing for giving me the opportunity to be able to work on this project and I've received much help from the editorial team and lastly to the reviewing team, and I would also like to thank Darryl Gove – The senior principal software engineer at Oracle and Oleg Strikov – the CPU Architect at NVIDIA, who had rendered much help for getting this stuff right with their sublime and gentle intellect, and lastly to my manager, Sau Sheong, who inspired me to start this. Thanks guys.
OpenCL Parallel Programming Development Cookbook

Welcome to the OpenCL Parallel Programming Development Cookbook! Whew, that was more than a mouthful. This book was written by a developer, that's me, and for a developer, hopefully that's you. This book will look familiar to some and distinct to others. It is a result of my experience with OpenCL, but more importantly in programming heterogeneous computing environments. I wanted to organize the things I've learned and share them with you, the reader, and decided upon taking an approach where each problem is categorized into a recipe. These recipes are meant to be concise, but admittedly some are longer than others. The reason for doing that is because the problems I've chosen, which manifest as chapters in this book describe how you can apply those techniques to your current or future work. Hopefully it can be a part of the reference, which rests on your desk among others. I certainly hope that understanding the solution to these problems can help you as much as they helped me.

This book was written keeping a software developer in mind, who wishes to know not only how to program in parallel but also think in parallel. The latter is in my opinion more important than the former, but neither of them alone solves anything. This book reinforces each concept with code and expands on that as we leverage upon more recipes.

This book is structured to ease you gently into OpenCL by getting you to be familiar with the core concepts of OpenCL, and then we'll take deep dives by applying that newly gained knowledge into the various recipes and general parallel computing problems you'll encounter in your work.

To get the most out of this book, it is highly recommended that you are a software developer or an embedded software developer, and is interested in parallel software development but don't really know where/how to start. Ideally, you should know some C or C++ (you can pick C up since its relatively simple) and comfortable using a cross-platform build system, for example, CMake in Linux environments. The nice thing about CMake is that it allows you to generate build environments for those of you who are comfortable using Microsoft's Visual Studio, Apple's XCode, or some other integrated development environment. I have to admit that the examples in this book used neither of these tools.

For More Information:
What This Book Covers

Chapter 1, Using OpenCL, sets the stage for the reader by establishing OpenCL in its purpose and motivation. The core concepts are outlined in the recipes covering the intrinsics of devices and their interactions and also by real working code. The reader will learn about contexts and devices and how to create code that runs on those devices.

Chapter 2, Understanding OpenCL Data Transfer and Partitioning, discusses the buffer objects in OpenCL and strategies on how to partition data amongst them. Subsequently, readers will learn what work items are and how data partitioning can take effect by leveraging OpenCL abstractions.

Chapter 3, Understanding OpenCL Data Types, explains the two general data types that OpenCL offers, namely scalar and vector data types, how they are used to solve different problems, and how OpenCL abstracts native vector architectures in processors. Readers will be shown how they can effect programmable vectorization through OpenCL.

Chapter 4, Understanding OpenCL Functions, discusses the various functionalities offered by OpenCL in solving day-to-day problems, for example, geometry, permuting, and trigonometry. It also explains how to accelerate that by using their vectorized counterparts.

Chapter 5, Developing a Histogram OpenCL program, witnesses the lifecycle of a typical OpenCL development. It also discusses about the data partitioning strategies that rely on being cognizant of the algorithm in question. The readers will inadvertently realize that not all algorithms or problems require the same approach.

Chapter 6, Developing a Sobel Edge Detection Filter, will guide you in how to build an edge detection filter using the Sobel's method. They will be introduced into some mathematical formality including convolution theory in one-dimension and two-dimensions and its accompanying code. And finally, we introduce how profiling works in OpenCL and its application in this recipe.

Chapter 7, Developing the Matrix Multiplication with OpenCL, discusses parallelizing the matrix multiplication by studying its parallel form and applying the transformation from sequential to parallel. Next, it'll optimize the matrix multiplication by discussing how to increase the computation throughput and warming the cache.

For More Information:
Chapter 8, Developing the Sparse Matrix-Vector Multiplication with OpenCL, discusses the context of this computation and the conventional method used to solve it, that is, the conjugate gradient through enough math. Once that intuition is developed, readers will be shown how various storage formats for sparse matrices can affect the parallel computation and then the readers can examine the ELLPACK, ELLPACK-R, COO, and CSR.

Chapter 9, Developing Bitonic Sort Using OpenCL, will introduce readers to the world of sorting algorithms, and focus on the parallel sorting network also known as bitonic sort. This chapter works through the recipes, as we did in all other chapters by presenting the theory and its sequential implementation, and extracting the parallelism from the transformation, and then developing the final parallel version.

Chapter 10, Developing the Radix Sort with OpenCL, will introduce a classic example of non-comparison based sorting algorithms, for example, QuickSort where it suits a GPU architecture better. The reader is also introduced to another core parallel programming technique known as reduction, and we developed the intuition of how reduction helps radix sort perform better. The radix sort recipe also demonstrates multiple kernel programming and highlights the advantages as well as the disadvantages.

For More Information:
In this chapter, we will cover the following recipes:

- Querying OpenCL platforms
- Querying OpenCL devices on your platform
- Querying for OpenCL device extensions
- Querying OpenCL contexts
- Querying an OpenCL program
- Creating OpenCL kernels
- Creating command queues and enqueueing OpenCL kernels

**Introduction**

Let's start the journey by looking back into the history of computing and why OpenCL is important from the respect that it aims to unify the software programming model for heterogeneous devices. The goal of OpenCL is to develop a royalty-free standard for cross-platform, parallel programming of modern processors found in personal computers, servers, and handheld/embedded devices. This effort is taken by "The Khronos Group" along with the participation of companies such as Intel, ARM, AMD, NVIDIA, QUALCOMM, Apple, and many others. OpenCL allows the software to be written once and then executed on the devices that support it. In this way it is akin to Java, this has benefits because software development on these devices now has a uniform approach, and OpenCL does this by exposing the hardware via various data structures, and these structures interact with the hardware via **Application Programmable Interfaces (APIs)**. Today, OpenCL supports CPUs that includes x86s, ARM and PowerPC and GPUs by AMD, Intel, and NVIDIA.

For More Information:

Developers can definitely appreciate the fact that we need to develop software that is cross-platform compatible, since it allows the developers to develop an application on whatever platform they are comfortable with, without mentioning that it provides a coherent model in which we can express our thoughts into a program that can be executed on any device that supports this standard. However, what cross-platform compatibility also means is the fact that heterogeneous environments exists, and for quite some time, developers have to learn and grapple with the issues that arise when writing software for those devices ranging from execution model to memory systems. Another task that commonly arose from developing software on those heterogeneous devices is that developers were expected to express and extract parallelism from them as well. Before OpenCL, we know that various programming languages and their philosophies were invented to handle the aspect of expressing parallelism (for example, Fortran, OpenMP, MPI, VHDL, Verilog, Cilk, Intel TBB, Unified parallel C, Java among others) on the device they executed on. But these tools were designed for the homogeneous environments, even though a developer may think that it's to his/her advantage, since it adds considerable expertise to their resume. Taking a step back and looking at it again reveals that is there is no unified approach to express parallelism in heterogeneous environments. We need not mention the amount of time developers need to be productive in these technologies, since parallel decomposition is normally an involved process as it's largely hardware dependent. To add salt to the wound, many developers only have to deal with homogeneous computing environments, but in the past few years the demand for heterogeneous computing environments grew.

The demand for heterogeneous devices grew partially due to the need for high performance and highly reactive systems, and with the "power wall" at play, one possible way to improve more performance was to add specialized processing units in the hope of extracting every ounce of parallelism from them, since that's the only way to reach power efficiency. The primary motivation for this shift to hybrid computing could be traced to the research headed entitled Optimizing power using Transformations by Anantha P. Chandrakasan. It brought out a conclusion that basically says that many-core chips (which run at a slightly lower frequency than a contemporary CPU) are actually more power-efficient. The problem with heterogeneous computing without a unified development methodology, for example, OpenCL, is that developers need to grasp several types of ISA and with that the various levels of parallelism and their memory systems are possible. CUDA, the GPGPU computing toolkit, developed by NVIDIA deserves a mention not only because of the remarkable similarity it has with OpenCL, but also because the toolkit has a wide adoption in academia as well as industry. Unfortunately CUDA can only drive NVIDIA's GPUs.

The ability to extract parallelism from an environment that's heterogeneous is an important one simply because the computation should be parallel, otherwise it would defeat the entire purpose of OpenCL. Fortunately, major processor companies are part of the consortium led by The Khronos Group and actively realizing the standard through those organizations. Unfortunately the story doesn't end there, but the good thing is that we, developers, realized that a need to understand parallelism and how it works in both homogeneous and heterogeneous environments. OpenCL was designed with the intention to express parallelism in a heterogeneous environment.

For More Information:
For a long time, developers have largely ignored the fact that their software needs to take advantage of the multi-core machines available to them and continued to develop their software in a single-threaded environment, but that is changing (as discussed previously). In the many-core world, developers need to grapple with the concept of concurrency, and the advantage of concurrency is that when used effectively, it maximizes the utilization of resources by providing progress to others while some are stalled.

When software is executed concurrently with multiple processing elements so that threads can run simultaneously, we have parallel computation. The challenge that the developer has is to discover that concurrency and realize it. And in OpenCL, we focus on two parallel programming models: task parallelism and data parallelism.

Task parallelism means that developers can create and manipulate concurrent tasks. When developers are developing a solution for OpenCL, they would need to decompose a problem into different tasks and some of those tasks can be run concurrently, and it is these tasks that get mapped to processing elements (PEs) of a parallel environment for execution. On the other side of the story, there are tasks that cannot be run concurrently and even possibly interdependent. An additional complexity is also the fact that data can be shared between tasks.

When attempting to realize data parallelism, the developer needs to readjust the way they think about data and how they can be read and updated concurrently. A common problem found in parallel computation would be to compute the sum of all the elements given in an arbitrary array of values, while storing the intermediary summed value and one possible way to do this is illustrated in the following diagram and the operator being applied there, that is, \( \oplus \) is any binary associative operator. Conceptually, the developer could use a task to perform the addition of two elements of that input to derive the summed value.
Whether the developer chooses to embody task/data parallelism is dependent on the problem, and an example where task parallelism would make sense will be by traversing a graph. And regardless of which model the developer is more inclined with, they come with their own sets of problems when you start to map the program to the hardware via OpenCL. And before the advent of OpenCL, the developer needs to develop a module that will execute on the desired device and communication, and I/O with the driver program. An example example of this would be a graphics rendering program where the CPU initializes the data and sets everything up, before offloading the rendering to the GPU. OpenCL was designed to take advantage of all devices detected so that resource utilization is maximized, and hence in this respect it differs from the "traditional" way of software development.

Now that we have established a good understanding of OpenCL, we should spend some time understanding how a developer can learn it. And not to fret, because every project you embark with, OpenCL will need you to understand the following:

- Discover the makeup of the heterogeneous system you are developing for
- Understand the properties of those devices by probing it
- Start the parallel program decomposition using either or all of task parallelism or data parallelism, by expressing them into instructions also known as kernels that will run on the platform
- Set up data structures for the computation
- Manipulate memory objects for the computation
- Execute the kernels in the order that's desired on the proper device
- Collate the results and verify for correctness

Next, we need to solidify the preceding points by taking a deeper look into the various components of OpenCL. The following components collectively make up the OpenCL architecture:

- **Platform Model**: A platform is actually a host that is connected to one or more OpenCL devices. Each device comprises possibly multiple compute units (CUs) which can be decomposed into one or possibly multiple processing elements, and it is on the processing elements where computation will run.

- **Execution Model**: Execution of an OpenCL program is such that the host program would execute on the host, and it is the host program which sends kernels to execute on one or more OpenCL devices on that platform.

For More Information:

When a kernel is submitted for execution, an index space is defined such that a work item is instantiated to execute each point in that space. A work item would be identified by its global ID and it executes the same code as expressed in the kernel. Work items are grouped into work groups and each work group is given an ID commonly known as its work group ID, and it is the work group's work items that get executed concurrently on the PEs of a single CU.

That index space we mentioned earlier is known as NDRange describing an N-dimensional space, where N can range from one to three. Each work item has a global ID and a local ID when grouped into work groups, that is distinct from the other and is derived from NDRange. The same can be said about work group IDs. Let's use a simple example to illustrate how they work.

Given two arrays, A and B, of 1024 elements each, we would like to perform the computation of vector multiplication also known as dot product, where each element of A would be multiplied by the corresponding element in B. The kernel code would look something as follows:

```c
__kernel void vector_multiplication(__global int* a,
                                  __global int* b,
                                  __global int* c) {
    int threadId = get_global_id(0); // OpenCL function
    c[i] = a[i] * b[i];
}
```

In this scenario, let's assume we have 1024 processing elements and we would assign one work item to perform exactly one multiplication, and in this case our work group ID would be zero (since there's only one group) and work items IDs would range from {0 ... 1023}. Recall what we discussed earlier, that it is the work group's work items that can executed on the PEs. Hence reflecting back, this would not be a good way of utilizing the device.

In this same scenario, let's ditch the former assumption and go with this: we still have 1024 elements but we group four work items into a group, hence we would have 256 work groups with each work group having an ID ranging from {0 ... 255}, but it is noticed that the work item's global ID still would range from {0 ... 1023} simply because we have not increased the number of elements to be processed. This manner of grouping work items into their work groups is to achieve scalability in these devices, since it increases execution efficiency by ensuring all PEs have something to work on.

For More Information:
The NDRange can be conceptually mapped into an N-dimensional grid and the following diagram illustrates how a 2DRange works, where WG-X denotes the length in rows for a particular work group and WG-Y denotes the length in columns for a work group, and how work items are grouped including their respective IDs in a work group.

Before the execution of the kernels on the device(s), the host program plays an important role and that is to establish context with the underlying devices and laying down the order of execution of the tasks. The host program does the context creation by establishing the existence (creating if necessary) of the following:

- All devices to be used by the host program
- The OpenCL kernels, that is, functions and their abstractions that will run on those devices
- The memory objects that encapsulated the data to be used / shared by the OpenCL kernels.
- Once that is achieved, the host needs to create a data structure called a command queue that will be used by the host to coordinate the execution of the kernels on the devices and commands are issued to this queue and scheduled onto the devices. A command queue can accept: kernel execution commands, memory transfer commands, and synchronization commands. Additionally, the command queues can execute the commands in-order, that is, in the order they've been given, or out-of-order. If the problem is decomposed into independent tasks, it is possible to create multiple command queues targeting different devices and scheduling those tasks onto them, and then OpenCL will run them concurrently.

For More Information:
Memory Model: So far, we have understood the execution model and it's time to introduce the memory model that OpenCL has stipulated. Recall that when the kernel executes, it is actually the work item that is executing its instance of the kernel code. Hence the work item needs to read and write the data from memory and each work item has access to four types of memories: global, constant, local, and private. These memories vary from size as well as accessibilities, where global memory has the largest size and is most accessible to work items, whereas private memory is possibly the most restrictive in the sense that it's private to the work item. The constant memory is a read-only memory where immutable objects are stored and can be shared with all work items. The local memory is only available to all work items executing in the work group and is held by each compute unit, that is, CU-specific.

The application running on the host uses the OpenCL API to create memory objects in global memory and will enqueue memory commands to the command queue to operate on them. The host’s responsibility is to ensure that data is available to the device when the kernel starts execution, and it does so by copying data or by mapping/unmapping regions of memory objects. During a typical data transfer from the host memory to the device memory, OpenCL commands are issued to queues which may be blocking or non-blocking. The primary difference between a blocking and non-blocking memory transfer is that in the former, the function calls return only once (after being queued) it is deemed safe, and in the latter the call returns as soon as the command is enqueued.

Memory mapping in OpenCL allows a region of memory space to be available for computation and this region can be blocking or non-blocking and the developer can treat this space as readable or writeable or both.

Hence forth, we are going to focus on getting the basics of OpenCL by letting our hands get dirty in developing small OpenCL programs to understand a bit more, programmatically, how to use the platform and execution model of OpenCL.

The OpenCL specification Version 1.2 is an open, royalty-free standard for general purpose programming across various devices ranging from mobile to conventional CPUs, and lately GPUs through an API and the standard at the time of writing supports:

- Data and task based parallel programming models
- Implements a subset of ISO C99 with extensions for parallelism with some restrictions such as recursion, variadic functions, and macros which are not supported
- Mathematical operations comply to the IEEE 754 specification
- Porting to handheld and embedded devices can be accomplished by establishing configuration profiles
- Interoperability with OpenGL, OpenGL ES, and other graphics APIs

Throughout this book, we are going to show you how you can become proficient in programming OpenCL.

For More Information:
Using OpenCL

As you go through the book, you'll discover not only how to use the API to perform all kinds of operations on your OpenCL devices, but you'll also learn how to model a problem and transform it from a serial program to a parallel program. More often than not, the techniques you'll learn can be transferred to other programming toolsets.

In the toolsets, I have worked with OpenCL™, CUDA™, OpenMP™, MPI™, Intel thread building blocks™, Cilk™, CilkPlus™, which allows the developer to express parallelism in a homogeneous environment and find the entire process of learning the tools to application of knowledge to be classified into four parts. These four phases are rather common and I find it extremely helpful to remember them as I go along. I hope you will be benefited from them as well.

- **Finding concurrency**: The programmer works in the problem domain to identify the available concurrency and expose it to use in the algorithm design
- **Algorithm structure**: The programmer works with high-level structures for organizing a parallel algorithm
- **Supporting Structures**: This refers to how the parallel program will be organized and the techniques used to manage shared data
- **Implementation mechanisms**: The final step is to look at specific software constructs for implementing a parallel program.

Don't worry about these concepts, they'll be explained as we move through the book.

The next few recipes we are going to examine have to do with understanding the usage of OpenCL APIs, by focusing our efforts in understanding the platform model of the architecture.

### Querying OpenCL platforms

Before you start coding, ensure that you have installed the appropriate OpenCL development toolkit for the platform you are developing for. In this recipe, we are going to demonstrate how you can use OpenCL to query its platform to retrieve simple information about the compliant devices it has detected and its various properties.

#### Getting ready

In this first OpenCL application, you'll get to query your computer for the sort of OpenCL platform that's installed. In the setup of your computer, you could have a configuration where both NVIDIA and AMD graphic cards are installed, and in this case you might have installed both the AMD APP SDK and NVIDIA's OpenCL toolkit. And hence you would have both the platforms installed.

The following code listing is extracted from Ch1/platform_details/platform_details.c.

For More Information:

How to do it...

Pay attention to the included comments, as they would help you to understand each individual function:

```c
#include <stdio.h>
#include <stdlib.h>

#ifdef APPLE
#include <OpenCL/cl.h>
#else
#include <CL/cl.h>
#endif

void displayPlatformInfo(cl_platform_id id,
    cl_platform_info param_name,
    const char* paramNameAsStr) {
    cl_int error = 0;
    size_t paramSize = 0;

    error = clGetPlatformInfo( id, param_name, 0, NULL,
        &paramSize );
    char* moreInfo = (char*)alloca( sizeof(char) * paramSize);
    error = clGetPlatformInfo( id, param_name, paramSize,
        moreInfo, NULL );
    if (error != CL_SUCCESS ) {
        perror("Unable to find any OpenCL platform information");
        return;
    }
    printf("%s: %s\n", paramNameAsStr, moreInfo);
}

int main() {
    /* OpenCL 1.2 data structures */
    cl_platform_id* platforms;
    /* OpenCL 1.1 scalar data types */
    cl_uint numOfPlatforms;
    cl_int error;

    /* Get the number of platforms
     * Remember that for each vendor's SDK installed on the
     * Computer, the number of available platform also
```

For More Information:
Using OpenCL

increased.
*/
error = clGetPlatformIDs(0, NULL, &numOfPlatforms);
if(error < 0) {
    perror("Unable to find any OpenCL platforms");
    exit(1);
}
// Allocate memory for the number of installed platforms.
// alloca(...) occupies some stack space but is
// automatically freed on return
platforms = (cl_platform_id*) alloca(sizeof(cl_platform_id)
    * numOfPlatforms);
printf("Number of OpenCL platforms found: %d\n",
    numOfPlatforms);

    // We invoke the API 'clPlatformInfo' twice for each
    // parameter we're trying to extract
    // and we use the return value to create temporary data
    // structures (on the stack) to store
    // the returned information on the second invocation.
    for(cl_uint i = 0; i < numOfPlatforms; ++i) {
        displayPlatformInfo( platforms[i],
      CL_PLATFORM_PROFILE,
      "CL_PLATFORM_PROFILE" ) ;

        displayPlatformInfo( platforms[i],
      CL_PLATFORM_VERSION,
      "CL_PLATFORM_VERSION" ) ;

        displayPlatformInfo( platforms[i],
      CL_PLATFORM_NAME,
      "CL_PLATFORM_NAME" ) ;

        displayPlatformInfo( platforms[i],
      CL_PLATFORM_VENDOR,
      "CL_PLATFORM_VENDOR" ) ;

        displayPlatformInfo( platforms[i],
      CL_PLATFORM_EXTENSIONS,
      "CL_PLATFORM_EXTENSIONS" ) ;
    }
    return 0;
}
To compile it on the UNIX platform, you would run a compile command similar to the following:

```
gcc -std=c99 -Wall -DUNIX -g -DDEBUG -DAPPLE -arch i386 -o platform_details platform_details.c -framework OpenCL
```

When that happens, you would have a binary executable named `platform_details`.

To run the program, simply execute the `platform_details` program, and a sample output will be an OSX:

```
Number of OpenCL platforms found: 1
CL_PLATFORM_PROFILE: FULL_PROFILE
CL_PLATFORM_VERSION: OpenCL 1.0 (Dec 23 2010 17:30:26)
CL_PLATFORM_NAME: Apple
CL_PLATFORM_VENDOR: Apple
CL_PLATFORM_EXTENSIONS:
```

**How it works...**

When you first learn to program OpenCL, it can be a daunting task but it does get better as we move along. So, let's decipher the source code that we've just seen. The file is a C source file and what you'll notice is that it's arranged such that the system header files are almost always placed right near the top:

```
| #include <stdlib.h>
| #include <stdio.h>
```

Next is what the C programmers would call as the platform-dependent code:

```
| #ifdef APPLE
| #include <OpenCL/cl.h>
| #else
| #include <CL/cl.h>
| #endif
```

The OpenCL header files are needed for the program to be compiled because they contain the method signatures. Now, we will try to understand what the rest of the code is doing. In OpenCL, one of the code conventions is to have data types be prefixed by `cl_` and you'll find data types for each of the platform, device and context as `cl_platform_XX`, `cl_device_XX`, `cl_context_XX`, and APIs prefixed in a similar fashion by `cl` and one such API is `clGetPlatformInfo`.

In OpenCL, the APIs do not assume that you know exactly how many resources (for example platforms, devices, and contexts) are present or are needed when you write the OpenCL code. And in order to write portable code, the developers of the language have figured out a clever way to present the API such that you use the same API to pose a general question and based on the results of that question, request more information via the same API. Let me illustrate with an example.

For More Information:

In the code, you will notice that `clGetPlatformInfo()` was invoked twice. The first invocation was to query the number of platforms that were installed on the machine. Based on the results of that query, we invoked `clGetPlatformInfo` again, but this time we passed context-sensitive information, for example, obtaining the name of the vendor. You'll find this pattern recurring when programming with OpenCL and the cons, I can think of is that it makes the API rather cryptic at times, but the nice thing about it is that it prevents the proliferation of APIs in the language.

Admittedly, this is rather trivial when it comes to the entire ecosystem of programming OpenCL, but subsequent chapters will show how you can transform sequential code to parallel code in OpenCL.

Next, let's build on the code and query OpenCL for the devices that are attached to the platform.

**Querying OpenCL devices on your platform**

We'll now query OpenCL devices that are installed on your platforms.

**Getting ready**

The code listing discussed in the *How to do it...* section presents an abbreviated portion of the code in `Ch1/device_details/device_details.c`. This code demonstrates how you can obtain the types of devices installed on your platform via `clGetDeviceIDs`. You'll use that information to retrieve detailed data about the device by passing it to `clGetDeviceInfo`.

**How to do it...**

For this recipe, you need to completely reference the appropriate chapter code. Pay attention to the included comments, as they would help you understand each individual function. We've included the main part of this recipe with highlighted commentary:

```c
/* C-function prototype */
void displayDeviceDetails(cl_device_id id, cl_device_info param_name, const char* paramNameAsStr) ;

... 
void displayDeviceInfo(cl_platform_id id, cl_device_type dev_type) {
  /* OpenCL 1.1 device types */
  cl_int error = 0;
  cl_uint numOfDevices = 0;
  /* Determine how many devices are connected to your platform */
```

For More Information:
error = clGetDeviceIDs(id, dev_type, 0, NULL, 
&numOfDevices);
if (error != CL_SUCCESS ) {
    perror("Unable to obtain any OpenCL compliant device 
     info");
    exit(1);
}

cl_device_id* devices = (cl_device_id*)
    alloca(sizeof(cl_device_id) * numOfDevices);

/* Load the information about your devices into the 
   variable 'devices' */
error = clGetDeviceIDs(id, dev_type, numOfDevices, devices, 
    NULL);
if (error != CL_SUCCESS ) {
    perror("Unable to obtain any OpenCL compliant device 
     info");
    exit(1);
}

printf("Number of detected OpenCL devices: 
    %d\n",numOfDevices);

/*
 * We attempt to retrieve some information about the 
 * devices.
 */

for(int i = 0; i < numOfDevices; ++ i ) {
    displayDeviceDetails( devices[i], CL_DEVICE_TYPE, "CL_DEVICE_ 
        TYPE" );
    displayDeviceDetails( devices[i], CL_DEVICE_VENDOR_ID, "CL_ 
        DEVICE_VENDOR_ID" );
    displayDeviceDetails( devices[i], CL_DEVICE_MAX_COMPUTE_UNITS, 
        "CL_DEVICE_MAX_COMPUTE_UNITS" );
    displayDeviceDetails( devices[i], CL_DEVICE_MAX_WORK_ITEM_ 
        DIMENSIONS, "CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS" );
    displayDeviceDetails( devices[i], CL_DEVICE_MAX_WORK_ITEM_ 
        SIZES, "CL_DEVICE_MAX_WORK_ITEM_SIZES" );
    displayDeviceDetails( devices[i], CL_DEVICE_MAX_WORK_GROUP_ 
        SIZE, "CL_DEVICE_MAX_WORK_GROUP_SIZE" );
}

For More Information:
www.packtpub.com/opencl-parallel-programming-development-
cookbook/book
void displayDeviceDetails(cl_device_id id,
                        cl_device_info param_name,
                        const char* paramNameAsStr) {

    cl_int error = 0;
    size_t paramSize = 0;

    error = clGetDeviceInfo( id, param_name, 0, NULL, &paramSize );
    if (error != CL_SUCCESS ) {
        perror("Unable to obtain device info for param\n");
        return;
    }

    /*
     * The cl_device_info are preprocessor directives defined in cl.h
     */

    switch (param_name) {
        case CL_DEVICE_TYPE: {
            cl_device_type* devType = (cl_device_type*)
            alloca(sizeof(cl_device_type) * paramSize);
            error = clGetDeviceInfo( id, param_name, paramSize,
            devType, NULL );
            if (error != CL_SUCCESS ) {
                perror("Unable to obtain device info for param\n");
                return;
            }

            switch (*devType) {
                case CL_DEVICE_TYPE_CPU :
                    printf("CPU detected\n");break;
                case CL_DEVICE_TYPE_GPU :
                    printf("GPU detected\n");break;
                case CL_DEVICE_TYPE_DEFAULT :
                    printf("default detected\n");break;
            }
        }break;
        // omitted code – refer to source "device_details.c"
    } //end of switch

On UNIX platforms, you can compile device_details.c by running this command on your terminal:

    gcc -std=c99 -Wall -DUNIX -g -DDEBUG -DAPPLE -arch i386 -o device_details
device_details.c   -framework OpenCL

For More Information:
And a binary executable named `device_details` should be deposited locally on your machine.

When you execute the binary executable depending on your machine's setup, you will see varying results. But on my OSX platform here is the output when executed on a machine with Intel Core i5 processor with a NVIDIA mobile GPU GT330m (extensions are highlighted):

```
Number of OpenCL platforms found: 1
CL_PLATFORM_PROFILE: FULL_PROFILE
CL_PLATFORM_VERSION: OpenCL 1.0 (Dec 23 2010 17:30:26)
CL_PLATFORM_NAME: Apple
CL_PLATFORM_VENDOR: Apple
CL_PLATFORM_EXTENSIONS:
Number of detected OpenCL devices: 2
GPU detected
  VENDOR ID: 0x1022600
  Maximum number of parallel compute units: 6
  Maximum dimensions for global/local work-item IDs: 3
  Maximum number of work-items in each dimension: 512
  Maximum number of work-items in a work-group: 512
CPU detected
  VENDOR ID: 0x1020400
  Maximum number of parallel compute units: 4
  Maximum dimensions for global/local work-item IDs: 3
  Maximum number of work-items in each dimension: 1
  Maximum number of work-items in a work-group: 1
```

Don't worry too much if the information doesn't seem to make sense right now, the subsequent chapters will reveal all.

**How it works...**

Leveraging the work we did in the previous section, now we have made use of the platform via `clGetPlatformInfo`, that was detected to query for the devices attached. This time, we used new API functions, `clGetDeviceIDs` and `clGetDeviceInfo`. The former attempts to uncover all the basic information about the devices attached to the given platform, and we use `clGetDeviceInfo` to iterate through the results to understand more about their capabilities. This information is valuable when you are crafting your algorithm and is not very sure about what device it's going to be run on. Considering that OpenCL supports various processors, it's a good way to write portable code.

There is actually a lot more information you can derive from your device and I'd strongly suggest you to refer [http://www.khronos.org/registry/cl/sdk/2.0/docs/man/xhtml/](http://www.khronos.org/registry/cl/sdk/2.0/docs/man/xhtml/) and look at the main page for `clGetDeviceInfo`.

**For More Information:**

Now that we've understood how to query the platform and the attached devices, we should take a look at how to query OpenCL extensions. The extensions allow the vendor to define additional capabilities that's delivered with the OpenCL compliant device, which in turn allows you, the programmer, to utilize them.

**Querying for OpenCL device extensions**

The extensions in OpenCL allow the programmer to leverage on additional capabilities provided by the vendor of the device, and hence they're optional. However, there are extensions that are recognized by OpenCL and purportedly supported by major vendors.

Here's a partial list of the approved and supported extensions in OpenCL 1.2. If you wish to discover the entire list of extensions that adopters of OpenCL have made public (some are given in the table), please refer to the PDF document via this link: [http://www.khronos.org/registry/cl/specs/opencl-1.2-extensions.pdf](http://www.khronos.org/registry/cl/specs/opencl-1.2-extensions.pdf).

<table>
<thead>
<tr>
<th>Extension name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cl_khr_fp64</td>
<td>This expression gives a double precision floating-point</td>
</tr>
<tr>
<td>cl_khr_int64_base_atomics</td>
<td>This expression gives 64-bit integer base atomic operations, provides atomic</td>
</tr>
<tr>
<td></td>
<td>operations for addition, subtraction, exchange, increment/decrement, and CAS</td>
</tr>
<tr>
<td>cl_khr_int64_extended_atomics</td>
<td>This expression gives 64-bit integer extended atomic operations, provides</td>
</tr>
<tr>
<td></td>
<td>atomic operations for finding the minimum, maximum, and boolean operations</td>
</tr>
<tr>
<td></td>
<td>such as and, or, and xor</td>
</tr>
<tr>
<td>cl_khr_3d_image_writes</td>
<td>This expression writes to 3D image objects</td>
</tr>
<tr>
<td>cl_khr_fp16</td>
<td>This expression gives a halfly precised floating point</td>
</tr>
<tr>
<td>cl_khr_global_int32_base_atomics</td>
<td>This expression gives atomics for 32-bit operands</td>
</tr>
<tr>
<td>cl_khr_global_int32_extended_atomics</td>
<td>This expression gives more atomic functionality for 32-bit operands</td>
</tr>
<tr>
<td>cl_khr_local_int32_base_atomics</td>
<td>This expression gives atomics for 32-bit operands in shared memory space</td>
</tr>
<tr>
<td>cl_khr_local_int32_extended_atomics</td>
<td>This expression gives more atomic functionality for 32-bit operands in</td>
</tr>
<tr>
<td></td>
<td>shared memory space</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Extension name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cl_khr_byte_addressable_store</td>
<td>This expression allows memory writes to bytes less than a 32-bit word</td>
</tr>
<tr>
<td>cl_APPLE_gl_sharing</td>
<td>This expression provides MacOSX OpenGL sharing, and also allows applications to use the OpenGL buffer, texture, and render buffer objects as OpenCL memory objects</td>
</tr>
<tr>
<td>cl_khr_gl_sharing</td>
<td>This expression provides OpenGL sharing</td>
</tr>
<tr>
<td>cl_khr_gl_event</td>
<td>This expression retrieves CL event objects from GL sync objects</td>
</tr>
<tr>
<td>cl_khr_d3d10_sharing</td>
<td>This expression shares memory objects with Direct3D 10</td>
</tr>
</tbody>
</table>

Next, let's find out how we can determine what extensions are supported and available on your platform by leveraging the previous code we've worked on.

**Getting ready**

The listing below only shows the interesting portion of the code found in Ch1/device_extensions/device_extensions.c. Various devices that are OpenCL compliant will have different capabilities, and during your application development you definitely want to make sure certain extensions are present prior to making use of them. The code discussed in the *How to do it...* section of this recipe shows you how to retrieve those extensions.

**How to do it...**

We've included the main querying function, which allows you to implement this particular recipe:

```c
void displayDeviceDetails(cl_device_id id,
                           cl_device_info param_name,
                           const char* paramNameAsStr) {

    cl_int error = 0;
    size_t paramSize = 0;

    error = clGetDeviceInfo( id, param_name, 0, NULL, &paramSize );
    if (error != CL_SUCCESS ) {
        perror("Unable to obtain device info for param\n");
        return;
    }

    /* the cl_device_info are preprocessor directives defined in cl.h */
```
Using OpenCL

```c
/*
   switch (param_name) {
     // code omitted – refer to "device_extensions.c"
     case CL_DEVICE_EXTENSIONS :
       size_t* ret = (size_t*) alloc(sizeof(size_t) * paramSize);
       error = clGetDeviceInfo( id, param_name, paramSize, ret, NULL );
       char* extension_info = (char*)malloc(sizeof(char) * (*ret));
       error = clGetDeviceInfo( id, CL_DEVICE_EXTENSIONS, sizeof(extension_info), extension_info, NULL);
       printf("\tSupported extensions: %s\n", extension_info);
     } break;
  } //end of switch
}
```

To compile the code, do as you did before by running a similar command on your terminal like this:

```
gcc -std=c99 -Wall -DUNIX -g -DDEBUG -DAPPLE -arch i386 -o device_extensions device_extensions.c -framework OpenCL
```

On a UNIX platform, here's what we got when executed on an Intel Core i5 processor with an NVIDIA mobile GPU GT330m (extensions are highlighted):

```
Number of OpenCL platforms found: 1
CL_PLATFORM_PROFILE: FULL_PROFILE
CL_PLATFORM_VERSION: OpenCL 1.0 (Dec 23 2010 17:30:26)
CL_PLATFORM_NAME: Apple
CL_PLATFORM_VENDOR: Apple
CL_PLATFORM_EXTENSIONS:
Number of detected OpenCL devices: 2
GPU detected
  VENDOR ID: 0x1022600
  Maximum number of parallel compute units: 6
  Maximum dimensions for global/local work-item IDs: 3
  Maximum number of work-items in each dimension: ( 512 512 64 )
  Maximum number of work-items in a work-group: 512
  Supported extensions: cl_khr_byte_addressable_store cl_khr_global_int32_base_atomics
                      cl_khr_global_int32_extended_atomics cl_APPLE_gl_sharing cl_APPLE_SetMemObjectDestructor
                      cl_APPLE_ContextLoggingFunctions cl_khr_local_int32_base_atomics cl_khr_local_int32_extended_atomics
CPU detected
  VENDOR ID: 0x1020400
```

For More Information:

Maximum number of parallel compute units: 4  
Maximum dimensions for global/local work-item IDs: 3  
Maximum number of work-items in each dimension: ( 1 1 l )  
Maximum number of work-items in a work-group: 1  
Supported extensions: cl_khr_fp64 cl_khr_global_int32_base_atomics  
cl_khr_global_int32_extended_atomics cl_khr_local_int32_base_atomics  
cl_khr_local_int32_extended_atomics cl_khr_byte_addressable_store  
cl_APPLE_gl_sharing cl_APPLE_SetMemObjectDestructor cl_APPLE_ContextLoggingFunctions

How it works...

When we examine the work we just did, we simply leveraged on the existing code and added the needed functionality where it was required, namely by adding code to handle the case where CL_DEVICE_EXTENSIONS was being passed in. We created an array of a fixed size on the stack and passed that array to clGetDeviceInfo, where the API will eventually store the information into the array. Extracting the information is as simple as printing out the array. For advanced usage, you might want to deposit that information into a global table structure where the other parts of the application can make use of it.

To understand what those extensions mean and how you can take advantage of them, I'd suggest that you refer to the Khronos register for OpenCL: http://www.khronos.org/registry/cl/.

We won’t dwell too much on each extension that we've seen so far. Let’s move on to understanding the OpenCL contexts.

Querying OpenCL contexts

An OpenCL context is created with one or more devices. Contexts are used by the OpenCL runtime for managing objects such as command queues (the object that allows you to send commands to the device), memory, program, and kernel objects, and for executing kernels on one or more devices specified in the context.

In more detail, OpenCL contexts can be created by associating a collection of devices that are available for the platform via clCreateContext or by associating it with a particular type of device, for example, CPU, GPUs, and so on, via clCreateContextFromType. However, in either way you cannot create contexts that are associated with more than one platform. Let’s use the example of vector multiplication in the Introduction section to demonstrate these concepts. The problem of vector multiplication or dot product can be solved using: pen and paper, CPU, GPU, or GPU + CPU. Obviously, the first option doesn't quite scale when we have a little more than 20 elements and with OpenCL you have more options. The first thing you need to decide is which platform it should be run, and in OpenCL it means deciding whether to use the AMD, NVIDIA, Intel, and so on. And what comes next is to decide whether to run the dot product on all of the devices listed for that platform or only some of it.

For More Information:  
So, let’s assume that the platform reports one Intel Core i7 and 3 AMD GPUs and the developer could use the `clCreateContextFromType` to restrict execution to either CPUs or GPUs, but when you use `clCreateContext`, you can list all the four devices to be executed against, theoretically speaking (however, in practice it’s hard to use all CPUs and GPUs effectively because the GPU can push more threads for execution than the CPU). The following diagram illustrates the options available to the developer to create contexts assuming the host environment is installed with both Intel and AMD’s OpenCL platform software. The configuration gets a little more interesting when you consider the Ivy Bridge Intel processor, which includes an HD Graphics co-processor that allows a context that’s both CPU and GPU aware.

Contexts have another interesting property, that is, it retains a reference count so that third-party libraries can refer to it and hence utilize the devices. For example, if the `cl_khr_d3d10_sharing` extension is available on your device, you can actually interoperate between OpenCL and Direct3D 10, and treat Direct3D 10 resources similar to memory objects as OpenCL memory objects that you can read from or write to. However, we will not demonstrate the capability with this extension in this book and will instead leave it to the reader to engage themselves in further exploration.

**Getting ready**

The code listing given in the *How to do it...* section is extracted from `Ch1/context_query/context_details.c`, and it illustrates how to create and release OpenCL contexts.

**How to do it...**

To query an OpenCL context, you need to include a function similar to the following in your code. You should reference the full code listing alongside this recipe:

```c
void createAndReleaseContext(cl_platform_id id,
                            cl_device_type dev_type) {
```

For More Information:

/* OpenCL 1.1 scalar types */
cl_int error = 0;
cl_uint numOfDevices = 0;

/* Determine how many devices are connected to your platform */
error = clGetDeviceIDs(id, dev_type, 0, NULL, &numOfDevices);
if (error != CL_SUCCESS ) {
    perror("Unable to obtain any OpenCL compliant device info");
    exit(1);
}
cl_device_id* devices = (cl_device_id*)
    alloca(sizeof(cl_device_id) * numOfDevices);

/*
Load the information about your devices into the variable
'devices'
*/
error = clGetDeviceIDs(id, dev_type, numOfDevices, devices, NULL);
if (error != CL_SUCCESS ) {
    perror("Unable to obtain any OpenCL compliant device info");
    exit(1);
}
printf("Number of detected OpenCL devices: %d\n",
numOfDevices);

/*
We attempt to create contexts for each device we find, 
report it and release the context. Once a context is 
created, its context is implicitly 
retained and so you don't have to invoke 
'clRetainContext'
*/

for(int i = 0; i < numOfDevices; ++ i ) {
    cl_context context = clCreateContext(NULL, 1,
        &devices[i],
        NULL, NULL,
        &error);

    cl_uint ref_cnt = 0;
    if (error != CL_SUCCESS) {
        perror("Can't create a context");
        exit(1);

For More Information:
Using OpenCL

```c
error = clGetContextInfo(context,
    CL_CONTEXT_REFERENCE_COUNT,
    sizeof(ref_cnt), &ref_cnt,
    NULL);

if (error != CL_SUCCESS) {
    perror("Can't obtain context information");
    exit(1);
}
printf("Reference count of device is %d\n", ref_cnt);
// Release the context
clReleaseContext(context);
```

On UNIX platforms, you can compile and build the program by typing the following command:

```
gcc -std=c99 -Wall -DUNIX -g -DDEBUG -DAPPLE -arch i386 -o context_details context_details.c -framework OpenCL
```

On the test machine, we have two OpenCL compliant devices. The first is the Intel Core i5 CPU, and the second is the NVIDIA mobile GT330m GPU. And the following is the output:

- Number of OpenCL platforms found: 1
- Number of detected OpenCL devices: 2
- Reference count of device is 1
- Reference count of device is 1

**How it works...**

If you have been following the book, you should realize that we didn't do anything special other than leverage on the previous exercises where we discover the sort of platforms installed, and with that uncover the devices and finally use that information to create the relevant contexts. Finally, with those relevant contexts we can query them. What you will notice is that the context's reference count is one in both cases, which indicates that a memory object is currently referencing it and the fact that we passed in `CL_CONTEXT_REFERENCE_COUNT` reflects this. This counter is only good when you want to detect if the application is experiencing a context leak, which actually means a memory leak. For OpenCL devices such as the CPU or GPU, the problem might not sound as a big deal. But for mobile processors, it would pose quite a serious problem since memory leaks, in general, wastes resources and the ultimately depleting battery life.

For More Information:  
There are actually more details where you can query the context via `clGetContextInfo` by passing in various `cl_context_info` types. Here's a list of them:

<table>
<thead>
<tr>
<th><code>cl_context_info</code></th>
<th>Return type</th>
<th>Information returned in <code>param_name</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>CL_CONTEXT_REFERENCE_COUNT</td>
<td>cl_uint</td>
<td>This variable returns the context reference count</td>
</tr>
<tr>
<td>CL_CONTEXT_NUM_DEVICES</td>
<td>cl_uint</td>
<td>This variable returns the number of devices in context</td>
</tr>
<tr>
<td>CL_CONTEXT_DEVICES</td>
<td>cl_device_id[]</td>
<td>This variable returns a list of devices in context</td>
</tr>
<tr>
<td>CL_CONTEXT_PROPERTIES</td>
<td>cl_context_properties</td>
<td>This variable returns the properties argument specified in <code>clCreateContext</code> or <code>clCreateContextFromType</code></td>
</tr>
</tbody>
</table>

Now that we've understood the basics of querying the platform, devices, extensions, and contexts I think it's time to take a look at OpenCL kernels and how you can program them.

### Querying an OpenCL program

In OpenCL, kernels refer to a function declared in a program. A program in OpenCL consists of a set of kernels that are functions declared with the `__kernel` qualifier in the code. Such a program encapsulates a context, a program source or binary, and the number of kernels attached. The following sections explain how to build the OpenCL program and finally load the kernels for execution on the devices.

### Getting ready

In order to run OpenCL kernels, you need to have a program (source or binary). Currently, there are two ways to build a program: from source files and other from binary objects via `clCreateProgramWithSource` and `clCreateProgramWithBinary` respectively (clever names). These two APIs return a program object represented by the OpenCL type, `cl_program` when successful. Let's examine the method signatures to understand it better:

```c
cl_program clCreateProgramWithSource(cl_context context,
                                      cl_uint count,
                                      const char** strings,
                                      const size_t* lengths,
                                      cl_int* errcode_ret)
```

For More Information:

Using OpenCL

If you read the signature carefully, you’ll notice that the OpenCL context needs to be created prior to building our program from source. Next the `strings` and `lengths` arguments hold the various (kernel) filenames and their respective file lengths, and the last argument, `errcode_ret` reflects the presence of errors while building the program:

```
cl_program clCreateProgramWithBinary(cl_context context,
   cl_uint num_devices,
   const cl_device_id* device_list,
   const size_t* lengths,
   const unsigned char** binaries,
   cl_int* binary_status,
   cl_int* errcode_ret)
```

Examine the signature and you can quickly realize that the `binaries` and `lengths` arguments hold the pointers to the program binaries and their respective lengths. All the binaries are loaded into the devices represented by the `device_list` argument through the context. Whether the program was loaded onto the device successfully is reflected in the `binary_status` argument. The developer would find this manner of program creation useful when the binary is the only artifact that can be exposed to customers or even during system integration tests.

For a developer to be able to create a valid OpenCL program by pulling offline binaries using `clCreateProgramWithBinary`, he needs to generate the offline binaries in the first place using the platform’s compiler and this process is unfortunately vendor specific. If you are using the AMD APP SDK, then you would need to enable the `cl_amd_offline_devices` AMD extension, and when you create the context, you need to pass in the `CL_CONTEXT_OFFLINE_DEVICES_AMD` property. If you are developing for the Intel or Apple OpenCL platforms, we would recommend you to consult the documentation at their websites.

Next, we need to build the program by invoking `clBuildProgram` passing it the created `cl_program` object from `clCreateProgramFromSource` and during program creation, the developer can provide additional compiler options to it (just as you perform when compiling C/C++ programs). Let’s see an example of how you might do this in the code given in the `How to do it...` section, abbreviated from `Ch1/build_opencl_program/build_opencl_program.c` and the OpenCL kernel files are listed in `Ch1/build_opencl_program/ {simple.cl, simple_2.cl`.

### How to do it...

To query an OpenCL program, you need to include a function similar to the following in your code. You should refer to the complete code listing alongside this recipe:

```
int main(int argc, char** argv) {
    // code omitted - refer to "build_opencl_program.c"
    ...
    // Search for a CPU/GPU device through the installed
```

For More Information:

// platform. Build a OpenCL program and do not run it.
for(cl_uint i = 0; i < numOfPlatforms; i++) {
    // Get the GPU device
    error = clGetDeviceIDs(platforms[i],
        CL_DEVICE_TYPE_GPU, 1,
        &device, NULL);
    if(error != CL_SUCCESS) {
        // Otherwise, get the CPU
        error = clGetDeviceIDs(platforms[i],
            CL_DEVICE_TYPE_CPU,
            1, &device, NULL);
    }
    if(error != CL_SUCCESS) {
        perror("Can't locate any OpenCL compliant device");
        exit(1);
    }
    /* Create a context */
    context = clCreateContext(NULL, 1, &device, NULL, NULL,
        &error);
    if(error != CL_SUCCESS) {
        perror("Can't create a valid OpenCL context");
        exit(1);
    }
    /* Load the two source files into temporary
datastores */
    const char *file_names[] = {
        "simple.cl",
        "simple_2.cl"};
    const int NUMBER_OF_FILES = 2;
    char* buffer[NUMBER_OF_FILES];
    size_t sizes[NUMBER_OF_FILES];
    loadProgramSource(file_names, NUMBER_OF_FILES, buffer,
        sizes);
    /* Create the OpenCL program object */
    program = clCreateProgramWithSource(context,
        NUMBER_OF_FILES,
        (const
        char**)buffer,
        sizes, &error);
    if(error != CL_SUCCESS) {
        perror("Can't create the OpenCL program object");
        exit(1);
    }
}
Using OpenCL

```c
/*
   Build OpenCL program object and dump the error
   message, if any
*/
char *program_log;
const char options[] = "-cl-finite-math-only \
                       -cl-no-signed-zeros"
size_t log_size;
error = clBuildProgram(program, 1, &device, options,
                        NULL, NULL);
// Uncomment the line below, comment the line above;
// build the program to use build options dynamically
// error = clBuildProgram(program, 1, &device, argv[1],
//                        NULL, NULL);

if(error != CL_SUCCESS) {
   // If there's an error whilst building the program,
   // dump the log
   clGetProgramBuildInfo(program, device,
                          CL_PROGRAM_BUILD_LOG, 0,
                          NULL,
                          &log_size);
   program_log = (char*) malloc(log_size+1);
   program_log[log_size] = '\0';
   clGetProgramBuildInfo(program, device,
                          CL_PROGRAM_BUILD_LOG,
                          log_size+1, program_log,
                          NULL);
   printf("\n=== ERROR ===
\n%s
====
", program_log);
   free(program_log);
   exit(1);
}
/* Clean up */
for(i=0; i< NUMBER_OF_FILES; i++) { free(buffer[i]); }
clReleaseProgram(program);
clReleaseContext(context);
}
```

Similar to what you did previously, the compilation command won't be too far off:

```bash
gcc -std=c99 -Wall -DUNIX -g -DDEBUG -DAPPLE -arch i386 -o build_opencl_ 
    program build_opencl_program.c -framework OpenCL
```

For More Information:
www.packtpub.com/opencl-parallel-programming-development-
cookbook/book
You'll find the executable file named `build_opencl_program` deposited on the filesystem.

There are two ways to run the program, depending on how you compile it. If you reexamine the code snippet shown earlier, you would notice that the compiler options is defined in the source code and hence it's static, but there's another dynamic way in which the compiler options can be passed during compilation and the following are those two simple approaches as follows:

If you chose the option of defining the build options statically, that is, if you have the following lines:

```c
const char options[] = "-cl-nosigned-zeros -cl-finite-math-only";
error = clBuildProgram(program, 1, &device, options, NULL, NULL);
```

OpenCL will simply build the program based on those build options you provided. This is rather suitable as the shipped application will have consistent results when running across different customer's setups.

To run the program, simply click on the `build_opencl_program` executable.

However, if you chose the other option of allowing your users to pass in options of their choice (largely depending on your algorithm design), that is, if you have something like this:

```c
error = clBuildProgram(program, 1, &device, argv[1], NULL, NULL);
```

In place of options, we have the array of pointers to strings, traditionally used to pass in arguments to the program via the command line (conveniently known to the C programmer as `argv`), then you would have allowed the user to pass in multiple build options.

To run the program, you would enter a command similar to this where you quote the multiple options (enclosed with quotes) you wish to pass to the program via `-D`:

```bash
./build_opencl_program -D"-cl-finite-math-only -cl-no-signed-zeros"
```

**How it works...**

The code example in this section is a little more involved than what we've been doing so far. What we did was to build an OpenCL program with two files: `simple.cl` and `simple_2.cl` which contains two simple OpenCL kernels via this (earlier) code snippet.

```c
const char *file_names[] = {"simple.cl",
                        "simple_2.cl"};
```

We demonstrated on to create the necessary data structures to store the contents of both files and the length of their program in two variables, `buffer` and `sizes`. 

---

**For More Information:**

Next, we demonstrated how you built an OpenCL program using the `cl_program` object that's returned by `clCreateProgramWithSource` with build options that are either pre or user defined. We've also learnt how to use the `clGetProgramInfo` to query the program object for the result of the build. Also, the host application has the capability to dump any build errors from this process.

Finally, we released the data structures associated with the program and contexts in reverse order of their creation. In OpenCL 1.2, there is another new manner in which you can build a OpenCL program object but you would need to use both of the new APIs: `clCompileProgram` and `clLinkProgram`. The rationale behind them is to facilitate separation, compilation, and linkage.

The build options deserved a further mention here, as there are in general four groups of options available to the OpenCL programmer. Go through the following for more information.

There are, in general, three groups of options available when you wish to build the OpenCL program: options to control behavior in math, optimizations, and miscellaneous.

The following table presents the math options available:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-cl-single-precision-constant</code></td>
<td>This option treats double precision floating point as a single precision constant.</td>
</tr>
<tr>
<td><code>-cl-denorms-are-zero</code></td>
<td>This option controls how single and double precision denormalized numbers are handled. The compiler can choose to flush these numbers to zero. See <a href="http://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/">http://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/</a>.</td>
</tr>
<tr>
<td><code>-cl-fp32-correctly-rounded-divide-sqrt</code></td>
<td>This option can be passed to <code>clBuildProgram</code> or <code>clCompileProgram</code>, which allows an application to specify that a single precision floating point divide (x / y and 1 / x) and sqrt used in the program source are correctly rounded.</td>
</tr>
</tbody>
</table>

The following table highlights the optimization options available:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-cl-opt-disable</code></td>
<td>This option disables all optimizations. Optimizations are enabled by default</td>
</tr>
<tr>
<td><code>-cl-mad-enable</code></td>
<td>This option allows a * b + c to be computed with reduced accuracy</td>
</tr>
<tr>
<td><code>-cl-unsafe-math-optimizations</code></td>
<td>This option combines the <code>-cl-mad-enable</code> and <code>-cl-no-signed-zeros</code> options</td>
</tr>
<tr>
<td><code>-cl-no-signed-zeros</code></td>
<td>This option allows floating point arithmetic to ignore the signedness of zero, since according to IEEE 754, there's a difference between +0.0 and -0.0</td>
</tr>
</tbody>
</table>

For More Information:
Chapter 1

- **-cl-finite-math-only**  This option allows optimizations to assume no floating point argument to take a NaN or an infinite value

- **-cl-fast-relaxed-math**  This option combines the **-cl-unsafe-math-optimizations** and the **-cl-finite-math-only** options

The following table here highlights the miscellaneous options available:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-w</td>
<td>This option prevents all warning messages</td>
</tr>
<tr>
<td>-Werror</td>
<td>This option turns all warning messages into errors</td>
</tr>
<tr>
<td>-cl-std=VERSION</td>
<td>This option builds the program based on the version of the OpenCL compiler (VERSION=CL1.1)</td>
</tr>
</tbody>
</table>

Let's move on to a bigger example where we create and query OpenCL kernels and eventually place them on a command queue for a device.

**Creating OpenCL kernels**

So far, we've managed to create a program from the source files. These source files are actually the OpenCL kernel code. Here's an example of how they look like:

```c
__kernel void simpleAdd(__global float *a,
                         __global float *b,
                         __global float *c) {
  int gid = get_global_id(0);
  c[gid] = a[gid] + b[gid];
}
```

The kernels are identified by **__kernel** qualified to the C-like function. The **__global** qualifiers refer to the memory space in which the variables reside. We'll have more to say about this in later chapters.

But this program cannot execute on the device even though we have created the program objects, as described previously. Recall that a program can reference several kernels and we need to hold on to those kernels, because it is the kernel that gets scheduled for execution on the devices and not the program object. OpenCL gives us the function to extract those kernels via **clCreateKernel** or **clCreateKernelsInProgram**. Let's take a close look at them:

```c
cl_kernel clCreateKernel(cl_program program,
                          const char* kernel_name,
                          cl_int* errcode_ret)
```

**For More Information:**

By looking at this code, you'll notice that in order to create the kernel we first need to create the program object, the name of the kernel function plus the capture of the return status. This API returns a `cl_kernel`, which represents the kernel object when successful. This API provides the programmer with an option of not transforming every kernel function in the program into actual OpenCL kernel objects ready for execution.

But if you wish to simply transform all kernel functions in the program into kernel objects, then `clCreateKernelsInProgram` is the API to use:

```c
cl_int clCreateKernelsInProgram(cl_program program,
                                cl_uint num_kernels,
                                cl_kernel* kernels,
                                cl_uint* num_kernels_ret)
```

You use this API to ask OpenCL to create and load the kernels into the `kernels` argument, and you hint to the OpenCL compiler how many kernels you're expecting with the `num_kernels` argument.

**Getting ready**

The complete code can be found in `ch1/kernel_query/kernel_query.c`. An abbreviated code is shown in the code snippet discussed in the *How to do it...* section of this recipe to keep us focused on the key APIs. This code requires one or more OpenCL source files, that is, `*.cl` and once you've placed them together you need to change the program's variables, `file_names` and `NUMBER_OF_FILES` to reflect the files accordingly.

**How to do it ...**

To query an OpenCL kernel, you'll need to include a function similar to the following in your code. You should reference the full code listing alongside this recipe:

```c
/*
Query the program as to how many kernels were detected *
*/
cl_uint numOfKernels;
error = clCreateKernelsInProgram(program, 0, NULL,
                                 &numOfKernels);
if (error != CL_SUCCESS) {
    perror("Unable to retrieve kernel count from program");
    exit(1);
}
cl_kernel* kernels = (cl_kernel*)
    alloca(sizeof(cl_kernel) *
           numOfKernels);
```

For More Information:

error = clCreateKernelsInProgram(program, numOfKernels, kernels, NULL);

for(cl_uint i = 0; i < numOfKernels; i++) {
    char kernelName[32];
    cl_uint argCnt;
    clGetKernelInfo(kernels[i],
        CL_KERNEL_FUNCTION_NAME,
        sizeof(kernelName),
        kernelName, NULL);
    clGetKernelInfo(kernels[i], CL_KERNEL_NUM_ARGS,
        sizeof(argCnt), &argCnt, NULL);
    printf("Kernel name: \%s with arity: \%d\n",
        kernelName, argCnt);
}

/* Release the kernels */
for(cl_uint i = 0; i < numOfKernels; i++)
    clReleaseKernel(kernels[i]);

The compilation is very similar to that of build_opencl_program.c illustrated in the previous section, so we’re skipping this step. When this application is run with two OpenCL source files, the output we will get is:

Number of OpenCL platforms found: 1
Kernel name: simpleAdd with arity: 3
Kernel name: simpleAdd_2 with arity: 3

The two source files, each defined a simple kernel function that adds its two arguments and stores the result into the third argument; and hence the arity of the function is 3.

**How it works...**

The code invokes clCreateKernelsInProgram twice. If you recall, this pattern recurs for many of the OpenCL APIs, where the first call would query the platform for certain details, which in this case is the number of kernels detected in the program. The subsequent calls would ask OpenCL to deposit the kernel objects into the storage referenced by kernels.

Finally, we invoke clGetKernelInfo, passing to it the retrieved kernel objects, and printing out some information about the kernel functions, such as the kernel function's name and the arity of the function through the CL_KERNEL_FUNCTION_NAME and CL_KERNEL_NUM_ARGS variables.
A complete list of details that can be queried from the kernel objects is reflected in the following table:

<table>
<thead>
<tr>
<th>cl_kernel_info</th>
<th>Return Type</th>
<th>Information returned in param_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL_KERNEL_FUNCTION_NAME</td>
<td>char[]</td>
<td>This variable returns the kernel function's name</td>
</tr>
<tr>
<td>CL_KERNEL_NUM_ARGS</td>
<td>cl_uint</td>
<td>This variable returns the number of arguments to kernel</td>
</tr>
<tr>
<td>CL_KERNEL_REFERENCE_COUNT</td>
<td>cl_uint</td>
<td>This variable returns the kernel reference count</td>
</tr>
<tr>
<td>CL_KERNEL_CONTEXT</td>
<td>cl_context</td>
<td>This variable returns the associated context for this kernel</td>
</tr>
<tr>
<td>CL_KERNEL_PROGRAM</td>
<td>cl_program</td>
<td>This variable returns the program object, that will be bound to the kernel object</td>
</tr>
</tbody>
</table>

Now that we've figured out how to create kernel objects, we should take a look at how to create command queues and start enqueuing our kernel objects and data for execution.

**Creating command queues and enqueuing OpenCL kernels**

This section will show you how to enqueue OpenCL kernel objects on the device. Before we do that, let's recall that we can create kernels without specifying an OpenCL device and the kernels can be executed on the device via the command queue.

At this point, we probably should spend some time talking about in-order execution and how they can be compared with out-of-order execution, though this subject is complex but intriguing as well. When a program is to be executed, the processor has the option of processing the instructions in the program in-order or out-of-order; a key difference between these two schemes is that in-order results in an execution order that is static, while out-of-order allows instructions to be scheduled dynamically. Out-of-order execution typically involves reordering the instructions, so that all computation units in the processors are utilized and driven by the goal of minimizing the stalling of the computation.

However, kernels are not the only objects that can be queued on the command queue. A kernel needs data so that it can perform its operations and data needs to be transferred to the device for consumption, and these data could be OpenCL buffer / sub-buffer or image objects. The memory objects that encapsulate the data need to be transported into the device and you have to issue memory commands to the command queue for that to occur; and in many use cases, it is common to hydrate the device with data prior to computation.
The following diagram highlights this use case where a kernel is scheduled for in-order execution, assuming that the kernel needs the data to be copied explicitly or memory-mapped, and upon completion of computation, the data is copied from the device's memory to host memory.

Also multiple command queues can be created and enqueued with commands and the reason for their existence is because the problem you wish to solve might involve some, if not all of the heterogeneous devices in the host. And they could represent independent streams of computation where no data is shared, or dependent streams of computation where each subsequent task depends on the previous task (often, data is shared). Take care that these command queues will execute on the device without synchronization, provided that no data is shared. If data is shared, then the programmer needs to ensure synchronization of the data through synchronization commands provided by the OpenCL specification.

For More Information:
Be aware that even though by default, commands enqueued in the command queue execute in-order, you can enable out-of-order execution by passing the `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE` flag when creating the command queue.

An example of out-of-order execution is shown in the following diagram, and let's assume that our problem is decomposed into three interdependent kernels, where each kernel will consume and process the data and then pass it to the next phase. Let's assume further that the execution of the kernels is out-of-order. What would happen next is mayhem and that's probably why this option is never the default.

However, the reader should be aware about CPUs from AMD and Intel.

When you start working on the kernels, you might discover that certain kernels seem to have better performance than others. And you can profile the kernel while you are fine-tuning it by passing the `CL_QUEUE_PROFILING_ENABLE` flag when creating the command queue.

**Getting ready**

Without repeating too much of the previous code, here's the relevant code that is derived from `ch1/kernel_queue/kernel_queue.c`. This code listing would need valid OpenCL kernel file(s) with distinct kernel function names (function overloading is disallowed) and valid function parameters. In `ch1/kernel_queue/hello_world.cl` you can see an example of such a function or kernel otherwise.

```c
__kernel void hello(__global char* data) {
}
```

For More Information:

You should reference the full code listing alongside this recipe:

```c
cl_kernel* kernels = (cl_kernel*) alloca(sizeof(cl_kernel) *
    numOfKernels);
error = clCreateKernelsInProgram(program, numOfKernels,
    kernels, NULL);
for(cl_uint i = 0; i < numOfKernels; i++) {
    char kernelName[32];
    cl_uint argCnt;
    clGetKernelInfo(kernels[i], CL_KERNEL_FUNCTION_NAME,
        sizeof(kernelName), kernelName, NULL);
    clGetKernelInfo(kernels[i], CL_KERNEL_NUM_ARGS,
        sizeof(argCnt),
        &argCnt, NULL);
    printf("Kernel name: %s with arity: %d\n", kernelName,
        argCnt);
    printf("About to create command queue and enqueue this
    kernel...");

    /* Create a command queue */
    cl_command_queue cQ = clCreateCommandQueue(context,
        device,
        0,
        &error);
    if (error != CL_SUCCESS) {
        perror("Unable to create command-queue");
        exit(1);
    }
    /* Create a OpenCL buffer object */
    cl_mem strObj = clCreateBuffer(context, CL_MEM_READ_ONLY |
        CL_MEM_COPY_HOST_PTR,
        sizeof(char) * 11,
        "dummy value", NULL);

    /*
    * Let OpenCL know that the kernel is suppose to receive an
    * Argument
    */
    error = clSetKernelArg(kernels[i],
        0,
        sizeof(cl_mem),
        &strObj);
```
```c
if (error != CL_SUCCESS) {
    perror("Unable to create buffer object");
    exit(1);
}

/* Enqueue the kernel to the command queue */
error = clEnqueueTask(cQ, kernels[i], 0, NULL, NULL);

if (error != CL_SUCCESS) {
    perror("Unable to enqueue task to command-queue");
    exit(1);
}

printf("Task has been enqueued successfully!\n");

}/* Release the command queue */
clReleaseCommandQueue(cQ);

/* Clean up */
for(cl_uint i = 0; i < numOfKernels; i++) {
    clReleaseKernel(kernels[i]);
}
```

As before, the compilation steps are similar to that in kernel_query.c with a command like:

```bash
gcc -std=c99 -Wall -DUNIX -g -DDEBUG -DAPPLE -arch i386 -o kernel_queue
kernel_queue.c   -framework OpenCL
```

Here's the sample output when I execute the application on my machine:

**Number of OpenCL platforms found:** 1

**Kernel name:** hello with arity: 1

About to create command queue and enqueue this kernel...

**Task has been enqueued successfully!**

From the output, you can tell that the task has been enqueued onto a command queue successfully!

### How it works...

Following from the previous section where we successfully queried the OpenCL kernel objects for information, we leverage on that code to create a command queue via `clCreateCommandQueue`, enqueue the kernel into the queue via `clEnqueueTask`, but not before setting the data needed for the kernel via `clSetKernelArg` and `clCreateBuffer`. You can ignore these two APIs for now, until we explain them in a later chapter.
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.