OpenCL Programming Guide for Mac OS X

Developer

Contents

Figures and Listings

Introduction

OpenCL (Open Computing Language) is an open standard for leveraging the computing power of GPUs and multi-core CPUs. OpenCL makes it possible for you to write cross-platform code that can execute on a variety of CPUs and GPUs without having to use the language of a specific vendor or to map general-purpose code to a 3D graphics API such as OpenGL or DirectX. OpenCL provides an abstraction layer that allows your general-purpose code to run in parallel on all the GPU and CPU cores on a system. The OpenCL technology is specifically designed for use with applications that need extra computing power, that must be portable, and that need to respond readily to evolving improvements in hardware. OpenCL can benefit scientific applications, image processing, signal processing, and other programs that require large amounts of parallel processing.

OpenCL Programming Guide for Mac OS X explains how to use OpenCL to exploit the computing power that resides in the many processing cores in a Macintosh computer's graphics processing unit (GPU) and central processing unit (CPU).

Who Should Read This Document?

Read "OpenCL Overview" (page 7) to find out exactly what OpenCL is and whether it will provide benefits to your application. After that, if you want to learn how to use OpenCL, this document will provide some context, basic concepts, and programming samples that will make it easier to learn how to use the OpenCL framework to create an OpenCL-enabled application. The document provides techniques and tips for establishing a workflow and achieving the best performance possible on the Mac OS X platform.

This guide assumes that you know C and have access to *The OpenCL Specification* . Although this guide does discuss many key OpenCL API functions, it does not provide detailed information on the OpenCL API or the OpenCL-C programming language.

Organization of This Document

This programming guide contains the following chapters:

- "OpenCL Overview" (page 7) provides definitions of OpenCL terminology and discusses fundamental concepts for understanding how to use the OpenCL framework.
- "OpenCL on the Mac Platform" (page 14) describes how OpenCL is implemented on Mac OS X.

- "The OpenCL Development Process and Workflow" (page 16) describes from start to finish the steps involved in making an OpenCL-enabled application.
- "OpenCL Memory Objects" (page 37) explains how to package application data in memory objects for later processing in kernels.
- "Basic Programming Sample" (page 47) provides a programming sample that illustrates all the essential steps in using OpenCL.
- "Programming with OpenCL-C" provides a brief introduction to the use of the OpenCL-C language. To be provided.
- "Improving Performance" offers tips and tricks for obtaining the best performance for an OpenCL program. To be provided.
- "Using OpenCL with OpenGL" describes how to share data structures between OpenCL and OpenGL. To be provided.

See Also

Please see the following for more information on OpenCL:

- *The OpenCL Specification* , available from the Khronos Group at http://www.khronos.org/registry/cl/.

OpenCL Overview

Introduced with Mac OS X v10.6, OpenCL is a Mac OS X framework as well as an open standard for writing applications that make use of GPUs and multi-core CPUs. Using OpenCL you can make your applications faster by moving the most time-consuming routines to a separate device within the system (most commonly, a GPU). OpenCL abstracts the nuances of the particular hardware so you don't need to write vendor-specific code in order to offload computation.

This chapter briefly summarizes the architecture of OpenCL and relates it to the OpenCL API. For detailed information about the OpenCL architecture and API, see *The OpenCL Specification* , available from the Khronos Group at http://www.khronos.org/registry/cl/.

OpenCL Terminology

All OpenCL applications make use of the same set of elements: hosts, devices, compute units, contexts, command queues, program objects, kernel functions, kernel objects, and memory objects. If you have past experience with parallel computing environments, some of these terms may seem familiar. However, understanding the nuances of these terms will help you get the most out of OpenCL on the Macintosh platform.

Devices

The OpenCL specification refers to any computational device on a computer system as a **device**. Within each device there are one or more units—referred to as **compute units**—that handle the actual computation. A compute unit is a hardware unit capable of performing computation by interpreting an instruction set. A device such as a central processing unit (CPU) may have one or more compute units. A CPU with multiple compute units is known as a *multi-core CPU* . Generally speaking, the number of compute units corresponds to the number of independent instructions that a device can execute at the same time. In the case of a dual-core CPU, for example, it is possible for it to execute two distinct instructions simultaneously.

CPUs commonly contain two to eight compute units, with the maximum increasing year-to-year. A graphics processing unit (GPU) typically contains many compute units—the GPUs in current Macintosh systems feature tens of compute units, and future GPUs may contain hundreds. As used by OpenCL, a CPU with 8 compute units is considered a single device, as is a GPU with 100 compute units.

Kernels

When you write a set of instructions in the OpenCL-C language intended for compilation and execution on a device, you're creating an OpenCL **kernel** (also called a *kernel function* or a *compute kernel*). A kernel is essentially a function written in a language that enables it to be compiled for execution on any device that supports OpenCL. Although kernels are enqueued for execution by host applications written in C, C++, or Objective C, a kernel must be compiled separately in order to be customized for the device on which it is going to run. You can write your OpenCL kernel source code in a separate file or include it inline in your host application source code. You can compile OpenCL kernels at runtime when launching the host application, or you can use a previously-built binary.

Kernel Objects

A **kernel object** encapsulates a specific kernel declared in a program, along with the argument values to use when executing this kernel.

Programs

An OpenCL **program** is a set of OpenCL kernels, auxiliary functions called by the kernels, and constants used by the kernels.

Contexts

The **context** is the environment in which OpenCL kernels execute. The context includes a set of devices, the memory accessible to those devices, and one or more command queues used to schedule execution of one or more kernels. A context is needed to share memory objects between devices.

Program Objects

An OpenCL **program object** is a data type that represents your OpenCL program. It encapsulates the following data:

- a reference to a context for the program (which is needed to know on which devices the program can run)
- the program source code or binary
- the latest successfully built OpenCL program executable
- the list of devices for which the program executable was built, the build options used, and a build log

Command Queues

OpenCL **command queues** are used for submitting work to a device. They order the execution of kernels on a device and manipulate memory objects. OpenCL executes the commands in the order that you enqueue them. .

Hosts

The program that calls OpenCL functions to set up the context in which kernels run and enqueue the kernels for execution is known as the **host application**. The device on which the host application executes is known as the **host device**. Before kernels can be run, the host application must complete the following steps:

1. determine what compute devices are available
2. select compute devices appropriate for the application
3. create command queues for selected compute devices
4. allocate the memory objects needed by the kernels for execution

Note that the host device (the CPU) can itself be an OpenCL device and can be used to execute kernel instances.

Memory Objects

A **memory object** is a handle to a region of global memory (see "Memory Model" (page 11)). You can create memory objects to reserve memory on a device to store your application data. There are two types of memory objects used in OpenCL: **buffer objects** and **image objects**, where buffer objects can contain any type of data and image objects are specifically for representing images. The host application can enqueue commands to read from and write to memory objects.

OpenCL Operation Model

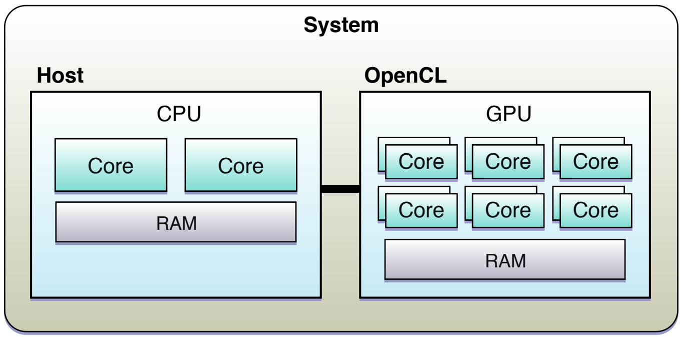
The operation of OpenCL can be described in terms of four interrelated models:

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

Platform Model

As shown in Figure 1-1 (page 10), the OpenCL device communicates with the host device—that is, the device on which the controlling application is running. Normally, the throughputs of the compute device's internal busses are much faster than the throughput of the external bus between the compute device and the host. In that case, of course, the speed of an external bus is not an issue. Because the data transfer takes a long time, you need to do enough computations in each kernel to make sure you are not limited by this latency. Note that the host, normally a CPU, can also be an OpenCL device.

Figure 1-1 OpenCL Platform Model



Execution Model

As described in "OpenCL Terminology" (page 7), execution of an OpenCL program involves simultaneous execution of multiple instances of a kernel on one or more OpenCL devices as queued and controlled by the host application. Each instance of a kernel is known as a **work-item**. Each work item executes the same code, but on different parts of the data. Each work-item runs on a single core of a multiprocessor. When you submit a kernel to execute on a device, you define the number of work-items that you need to completely process all of your data. This is known as an **index space**. OpenCL supports index spaces of 1, 2, or 3 dimensions. For example, if you have a kernel that changes the color value of a single pixel and you have an image that is 64 pixels wide by 64 pixels high, you might want to define a 2-dimensional index space of 64 by 64 so that there is a work-item for each pixel in the image. The total number of work-items is practically unlimited; use the number that maps best to your algorithm. OpenCL takes care of distributing the work-items among the available processors.

Work-items can be organized into **work-groups**. OpenCl supports synchronization of computation between work-items in a work-group using barriers and memory fences, but does not permit such synchronization between different work-groups or work-items in separate work-groups.

Each work-item has a unique **global ID**, which is the location of the work-item in the index space. For example, the work item in a 2-dimensional index space that is number 23 on the X axis and number 6 on the Y axis (counting from 0) would have the global ID (23,6). Each work-group has a unique **work-group ID**, which is similar to the work-item global ID in that it specifies the work-group's position in the index space. The number of work-groups in each dimension must divide evenly into the number of work-items in that dimension. For example, if your global work size is 64 by 144, then your work-groups could each contain 8 x 24 work items so that the work-group array is 8 x 6. (8 work groups in the X dimension x 8 work-items per work group = 64 work-items in the X direction. 6 work-groups in the Y dimension x 24 work-items per work group = 144 work-items in the Y dimension.) . The work-group ID of the work group that is at index 3 on the X axis and index 8 on the Y axis (again, counting from 0) is (3,8).

A work-item can also be located by a combination of the index of that item within its work-group (its **local ID**) plus the work-group ID of its work-group. For example, the work item in the preceding example with the global ID (23,6) would be the last work-item in the third work-group in the X dimension (local index 7, work-group index 2) and the sixth work-item in the first work-group in the Y dimension (local index 5, work-group index 0).

In order to generalize your OpenCL application to run on a variety of hardware, you cannot hard-code into your program the number of work-items and work-groups to be used. Use the `clGetDeviceInfo` with the selector `CL_DEVICE_MAX_WORK_ITEM_SIZES`) and the `clGetKernelWorkgroupInfo` function with the selector `CL_KERNEL_WORK_GROUP_SIZE` to determine the maximum work-group size for a given device and kernel. This number changes based on the device and kernel.

The host application sets up the context in which the kernels run, including allocation of memory of various types (see the following section), transfer of data among memory objects, and creation of command queues used to control the sequence in which commands—including the commands that execute kernels—are run. You are responsible for synchronizing any necessary order of execution. The OpenCL API includes synchronization commands for this purpose.

You can use the OpenCL API to query the OpenCL runtime about its status and find out when your kernel has finished executing. Once it's done you can retrieve the results of the computation from OpenCL and repeat as necessary.

Memory Model

OpenCL generalizes the different types of memory available into global memory, constant memory, local memory, and private memory, as follows:

- **Global memory** is available for reading and writing by all work-items in all work-groups. It is the device memory in the platform model.
- **Constant memory** is a region of global memory that is reserved for read-only access by work-items and is held constant during the execution of a kernel. It can be written and read by the host application.
- **Local memory** is available for reading and writing by a specific work-group and can be used to hold values that are shared by all work-items in a work-group.

- **Private memory** is available to only a single work-item.

When writing kernels in the OpenCL language, you must declare your memory with certain address space qualifiers to indicate whether the data resides in global, constant, local, or private memory, or it will default to private within a kernel.

Programming Models

OpenCL supports **data parallel** and **task parallel** programming models:

As used in OpenCL, data parallel processing refers to many instances of the same kernel being executed simultaneously, with each instance operating on its own set of data. Each set of data is associated with a point in a 1-, 2-, or 3-dimensional index space.

Task parallel programming is similar to the familiar process of spawning multiple threads, each performing different tasks. In OpenCL terms, task-parallel programming involves enqueuing many kernels, and letting OpenCL run them in parallel using available processors.

OpenCL Architecture

The OpenCL framework includes three main components:

The OpenCL

- compiler,
- API, and
- runtime.

The OpenCL-C language is based on the ISO/IEC 9899:1999 C language specification (also known as the C99 specification) with specific extensions and restrictions. This is a C-like programming language that can compile and run natively on any processing unit that supports the OpenCL standard. The OpenCL-C language extensions include mathematical functions that make it easier to implement graphics and numerical-method algorithms to solve problems in engineering and science. You can use this language to implement computation-intensive portions of your code as OpenCL programs that can process many data sets in parallel.

The OpenCL compiler is included in the framework so that your application can compile your OpenCL program during execution. Therefore, you don't need to compile and distribute different versions of your application for every possible Open-CL enabled device on the market. Instead, your application can compile the OpenCL source code for the specific OpenCL devices on the system the first time your application runs. You can then cache the compiled code so that your application does not need to recompile it each time it runs.

The OpenCL runtime abstracts the underlying hardware from the operating system on which OpenCL is running.

OpenCL on the Mac Platform

Mac OS X v10.6 and later ships with a native implementation of the OpenCL 1.0 specification.

OpenCL for Mac OS X is implemented as a framework that contains the OpenCL application programming interface (API), the OpenCL runtime engine, and an OpenCL compiler.

Structure of OpenCL in Mac OS X

The following sections provide an overview of the OpenCL components on the Mac OS X platform.

OpenCL Framework & Runtime

The Open CL framework (`OpenCL.framework`) provides all the necessary headers for compiling OpenCL source code as well as interfacing with the OpenCL runtime. The OpenCL API is available in the C programming language and you can include it in any C, C++, or Objective-C project.

Using the OpenCL framework API you can create OpenCL contexts (see "Contexts" (page 8)), find out what compute devices (such as CPUs and GPUs) are available on the machine, issue commands for OpenCL to perform, and query the status of the running commands. You use the OpenCL-C language to write a kernel ("Kernels" (page 8)), which defines the specific task you'd like to perform in parallel, and then use the compiler to compile those instructions. In order to run the compiled code, the system on which it runs must include the OpenCL runtime.

Apple's implementation of OpenCL is built on top of Grand Central Dispatch (GCD). For more information on GCD, see *Concurrency Programming Guide*.

OpenCL Compiler

Mac OS X's OpenCL compiler uses the clang and LLVM technologies to compile the kernels written in OpenCL and translate those instructions into optimized machine code targeted for the hardware on the host machine. Mac OS X v10.5 and later uses LLVM technology in other environments as well, such as for compiling OpenGL code. When the compiler converts your OpenCL code it first translates your instructions from OpenCL to an intermediate representation (IR). After that, LLVM does its best to optimize the IR before passing it along to the code generators for the devices that the code must execute on. The importance of this pipeline is that it makes it possible to write a single OpenCL program that can execute natively on various machine architectures—such as those of the various CPUs and GPUs available on your system. Although LLVM is fast, compiling is always expensive and the resulting programs should be cached by your application to avoid unnecessary re-compilation.

Multithreading on the Host

The Mac OS X OpenCL framework is thread-safe for all functions that create, retain, or release objects such as command queue objects, memory objects, program and kernel objects. Any functions that enqueue commands to the command queue, or change the state of one of the aforementioned objects is *not* thread-safe.

OpenCL should be able to create multiple command queues for a given OpenCL context, and multiple OpenCL contexts in an application running on the host processor.

If you enqueue commands for multiple threads, or access information from multiple threads, your application must do its own locking to prevent concurrent access. Although you can have multiple command queues and safely submit to them from separate threads, each command queue uses resources that may make this more expensive than implementing the correct locks.

The OpenCL Development Process and Workflow

Every application that uses OpenCL has to go through a pre-defined series of steps in order to efficiently offload computation to an OpenCL device. This chapter describes the structure of an OpenCL-enabled application and shows the steps the application must complete in order to use OpenCL.

Essential Development Tasks

OpenCL enables you to write source code that will run on any system with one or more OpenCL devices. To make this possible, the OpenCL framework includes a compiler and linker and the OpenCL API includes functions you can call from within your application to determine what OpenCL devices are available and to compile and link your OpenCL source code on the system on which the program is to run. The OpenCL development process includes these major steps:

1. Determine what you want to parallelize.

   The first step when deciding to use OpenCL is to examine the computational tasks your program performs and identify which of those tasks most easily lends itself to parallelism. Determining how to parallelize your program effectively is often the hardest part of developing an OpenCL program. See "Identifying Parallelizable Routines" (page 17).

2. Write your kernels and auxiliary OpenCL routines.

   To perform general purpose computation, you must write kernels. A kernel is a function that you write in the OpenCL-C language that gets encapsulated and compiled into an OpenCL program. See "Writing Kernels" (page 20). These first two steps usually represent the majority of the work in adapting an algorithm to run using OpenCL.

3. Set up the context.

   Using OpenCL framework functions you can query to find out what OpenCL devices are available on the system and then create an OpenCL context, including memory objects and one or more command queues used to schedule execution of your kernels. See "Querying Available Compute Devices" (page 21) and "Initializing and Creating OpenCL Contexts" (page 22).

4. Write code to compile and build your OpenCL program.

   After you've identified the available OpenCL devices and prepared an OpenCL context, you write code in your application that compiles and builds your source code and extracts kernel objects from the compiled code. Here's the sequence of commands you must follow:

   a. Call the `clCreateProgramWithSource` function to create your program from OpenCL-C source code or, if you have compiled code available (cached from an earlier run of the program, for example, or from an external source such as a library), call the `clCreateProgramWithBinary` function. These functions link all your kernels and auxiliary routines into a single program and return a program object.

   b. Call the `clBuildProgram` function to compile the program object for the specific devices on the system.

   c. Call the `clCreateKernel` function for each kernel, or call the `clCreateKernelsInProgram` function to create kernel objects for all the kernels in the OpenCL program.

5. Create memory objects to hold input and output data and write input data to the input objects.

6. Enqueue commands to the command queues to control the sequence and synchronization of kernel execution, reading and writing of data, and manipulation of memory objects.

   To execute a kernel function, you must do the following:

   a. Call the `clSetKernelArg` function to set the parameter value for each parameter in the kernel function definition.

   b. Determine the work-group size and index space to use to execute the kernel.

   c. Enqueue the kernel for execution in the command queue.

   For more information on executing kernels, see "Executing Kernels" (page 29).

7. Enqueue commands to read the output of the work-items into host memory.

Identifying Parallelizable Routines

The first step in using OpenCL to improve your application's performance is to identify what portions of your application are appropriate for parallelization. Whereas in a general application you can spawn a separate thread for a task as long as the functions in the thread are reentrant and you're careful about how you synchronize data access, to achieve the level of parallelism for which OpenCL is ideal, it is much more important for the work-items to be independent of each other. Although work-items in the same work-group can share local data, they execute synchronously and so neither work-item's calculations depend on the result from another work-item. Parallelization works only when the tasks that you run in parallel do not depend on each other.

For example, assume that you are writing a simple application that keeps track of the grades for students in a class. The application consists of two main tasks:

1. Compute the final grade for each student, assuming the final grade is the average of all the student's grades.
2. Obtain a class average by averaging the final grades of all students.

You cannot perform these two tasks in parallel because they are not independent of each other: in order to calculate the class average, you must have already calculated the final grade for each student.

Despite the fact that you cannot perform task 1 and task 2 simultaneously, there is still an opportunity for parallelization. To see how it can be broken down, it helps to look at a basic pseudocode example for computing the final grade for each student serially.

Listing 3-1 Pseudocode for computing the final grade for each student in a class

```
// assume 'class' is a collection of 'student' objects
foreach(student in class)
{
    // assume getGrades() returns a collection of integer grades
    grades = student.getGrades();

    sum = 0;
    count = 0;

    // iterate through each grade, adding it to sum
    foreach(grade in grades)
    {
        sum += grade;
        count++;
    }

    // cache the average grade in the student object
    student.averageGrade = sum / count;
}
```

The pseudocode in Listing 3-1 (page 18) proceeds through each student in the class, one by one, calculating the average of each student's grades and caching it in the student object. Although this example computes each grade average one at a time, there's no reason that the grade averages for all the students couldn't be calculated at the same time. Because the grades of one student do not affect the grades of another, you can calculate the grade averages for all the students at the same time instead of looping through the same set of instructions for each student, one at a time. This is the idea behind data parallelism.

Data parallelism consists of taking a single task (in this case, calculating a student's average grade), and repeating it over multiple sets of data. Students' grades do not affect each other, therefore you can process them in parallel. To express this programmatically, you must first separate your task (calculating the grade average of a student) from your data (the students in the class). Listing 3-2 shows how you can isolate the grade-averaging task.

Listing 3-2 The isolated grade average task

```
task calculateAverageGradeForStudent( student )
{
    // assume getGrades() returns a collection of integer grades
    grades = student.getGrades();

    sum = 0;
    count = 0;

    // iterate through each grade, adding it to sum
    foreach(grade in grades)
    {
        sum += grade;
        count++;
    }

    // store the average grade in the student object
    student.averageGrade = sum / count;
}
```

Now that you have the task isolated, you need to apply it to all students in the class in parallel. Because OpenCL has native support for parallel computing, you can re-write the task shown in Listing 3-2 (page 19) as an OpenCL kernel function. Using the OpenCL framework API you can enqueue this kernel to run on a device where each compute unit on the device can apply an instance of the kernel (that is, a work-item) to a different set of data.

The challenge in parallelizing your application is identifying the tasks that you can distribute across multiple compute units. Sometimes, as in this example, the identification is relatively trivial and requires few algorithmic changes. Other times, it may require designing a new algorithm from scratch that lends itself more readily to parallelization. Although there is no universal rule for parallelizing your application, there are a few tips you can keep in mind:

- Pay attention to loops.

  Often the opportunities for parallelization lie within a subroutine that is repeated over a range of results.
- Nested loops might be restructured as multi-dimensional parallel tasks.
- Find as many tasks as possible that do not depend on each other.

  Finding a group of routines that do not share memory or depend on each other's results is usually a good indicator that you can perform them in parallel. If you have enough such tasks, you can consider writing a task-parallel OpenCL program.
- Due to the overhead of setting up a context and transferring data over a PCI bus, you must be processing a fairly large data set before you see any benefits from using OpenCL. The exact point at which you start to see benefits depends on the OpenCL implementation and the hardware being used, so you will have to experiment to see how fast you can get your algorithm to execute. In general, a high ratio of computation to data access and lots of mathematical computations are good for OpenCL programs.

Writing Kernels

Kernels are written in the OpenCL-C language, which uses a C-like syntax with certain restrictions and a set of built-in functions. For example, Listing 3-3 shows one kernel function from the *OpenCL Parallel Prefix Sum (aka Scan) Example* sample code. Note that this kernel function calls several other auxiliary routines also in the OpenCL program but external to this particular kernel function.

Listing 3-3 Sample kernel function

```
__kernel void                                                  // [1]
PreScanKernel(                                                 // [2]
    __global float *output_data,
    __global const float *input_data,
    __local float* shared_data,
    const uint  group_index,
    const uint  base_index,
    const uint  n)
{
    const uint group_id = get_global_id(0) / get_local_size(0);   // [3]
    const uint group_size = get_local_size(0);

    uint local_index = (base_index == 0) ? mul24(group_id,         // [4]
                               (group_size << 1)) : base_index;

    uint4 address_pair = GetAddressMapping(local_index);          // [5]
    LoadLocalFromGlobal(shared_data, input_data, address_pair, n);
    PreScanGroup(shared_data, group_index);
    StoreLocalToGlobal(output_data, shared_data, address_pair, n);
}
```

Notes:

1. A kernel function is declared with the `__kernel` keyword.
2. Before you enqueue a kernel for execution, you call the `clSetKernelArg` function to pass values to the kernel's parameters. When you call that function, you identify each parameter by the order in which it's declared in the kernel definition (starting with `0` for the first parameter). See Listing 3-12 (page 32) for an example of the use of this function.
3. `get_global_id` and `get_local_size` are work-item functions you can use to get information about a work-item while it's executing.
4. `mul24` is a built-in math function; it multiplies two 24-bit integer values. The OpenCL-C language includes dozens of high-performance built-in math functions for scalar and vector types.
5. This call and the following three function calls are to auxiliary routines, also written in OpenCL-C, that are defined in the OpenCL program and that can therefore be called by any of the kernel functions.

Querying Available Compute Devices

Every OpenCL program requires an OpenCL context, including a list of the OpenCL devices available on the system. Use the `clGetDeviceIDs` function to query OpenCL for a list of devices on the machine that support OpenCL. You can restrict your search to a particular type of device or to any combination of device types. You must also specify the maximum number of device IDs that you want returned.

For example, suppose you know that you want your code to execute on a GPU and that, regardless of how many GPUs are actually available, you need only one to execute your program. If you pass `CL_DEVICE_TYPE_GPU` as the `device_type` parameter for `clGetDeviceIDs` and `1` for the `num_entries` parameter, OpenCL returns an ID for the first OpenCL GPU on that machine that it finds. The code is shown in Listing 3-4.

Listing 3-4 Checking for a GPU

```
// declare memory to hold the device id
cl_device_id device_id;

// ask OpenCL for exactly 1 GPU
cl_int err = clGetDeviceIDs(
        NULL                    // platform ID
        CL_DEVICE_TYPE_GPU,     // look only for GPUs
        1,                      // return an ID for only one GPU
        &device_id,             // on return, the device ID
        NULL);                  // don't return the number of devices

// make sure nothing went wrong
if (err != CL_SUCCESS)
{ ... }
```

Initializing and Creating OpenCL Contexts

Once you have determined the available OpenCL devices on the machine and have obtained at least one valid device ID, you are ready to create an OpenCL context. The context serves to group devices together to enable sharing of memory objects across different compute devices. You can create one from scratch for a specific device, or you can also create one from a preexisting OpenGL context if you are using OpenCL and OpenGL together, and would like to be able to share memory between the two. The following sections show you how to create OpenCL contexts.

Creating an OpenCL Context From Scratch

To create an OpenCL context, you must first determine which devices you would like to make this context for (using the `clGetDeviceIDs` function as shown in "Debugging" (page 36)), and then pass this device information to the `clCreateContext` function. Listing 3-5 shows a call to this function.

Listing 3-5 Creating a compute context

```
cl_context clCreateContext (
   0,               // a reserved variable
   numDevices,      // the number of devices in the devices parameter
   &devices,       // a pointer to the list of device IDs from clGetDeviceIDs
   NULL,           // a pointer to an error notice callback function (if any)
   NULL            // data to pass as a param to the callback function
   &err            // on return, points to a result code
);
```

You can specify an error notification callback function when creating an OpenCL context. See *The OpenCL Specification* for details.

Providing an error notification callback function is optional. Leaving this parameter as a NULL value results in no error notification being registered. Providing a callback function can be useful if you want to receive runtime errors for the particular OpenCL context. However, you should be aware that if you do supply a callback function, OpenCL may call it asynchronously throughout execution. It is therefore your responsibility to ensure that your callback function is thread-safe. Note that Apple has provided callback functions in the file OpenCL.framework/cl_ext.h for your use, including versions that write errors to standard out, standard error, and the system log.

Creating a Command Queue

After creating your OpenCL context, you must create a command queue. You create command queues using the clCreateCommandQueue function, as shown in Listing 3-6.

Listing 3-6 Calling the clCreateCommandQueue function

```
ComputeCommands = clCreateCommandQueue (
        ComputeContext,        // a valid OpenCL context
        ComputeDeviceId,       // a device associated with the context
        0,                     // a bit field to specify properties    [1]
        &err,                  // on return, points to a result code
);
```

Notes:

1. You can specify that profiling be enabled to measure execution time of commands. Specify 0 if you don't want to use this option.

The command queue issues commands to a specific compute device, and ensures that a set of operations occur in a particular order. By enqueuing commands, you request that the device execute your operations in the order that you enqueue them. Your application can even have several command queues and handle multiple sets of independent commands at the same time. As long as your command queues do not issue commands involving the same objects, you do not have to worry about synchronization. You are responsible for synchronizing any necessary order of execution. The OpenCL API includes synchronization commands for this purpose. Note that a command is not independent if it writes a memory object read by another command.

If you have multiple OpenCL devices (such as a CPU and a GPU), you must create a command queue for each and you have to divide up the work and submit commands separately to each.

Creating Program Objects

An OpenCL program consists of a set of kernel functions. Kernel functions are functions that you write in the OpenCL programming language in order to have them execute on a particular compute device. All kernel functions must be identified in the program source with the `__kernel` qualifier. OpenCL programs can also consist of helper functions that you can call from your kernel functions. However, these helper functions may not serve as entry points from the OpenCL API—that is, you can only enqueue kernels declared as such. A program object encapsulates your OpenCL program source, along with the latest successfully built program executable, as well as the build options, build log, and list of devices for which the program was built.

You can create an program object directly from the source code of an OpenCL program and compile it at application runtime. Alternatively, you can build a program object by using a previously built binary to avoid compilation at runtime. To build a program object from source, you use the `clCreateProgramWithSource` as shown in Listing 3-7.

Listing 3-7 Calling the `clCreateProgramWithSource` function

```
// OpenCL kernel that computes the square of an input array

const char *KernelSource = "\n" \
"__kernel square(                                                   \n" \
"   __global float* input,                                          \n" \
"   __global float* output,                                         \n" \
"   const unsigned int count)                                       \n" \
"{                                                                  \n" \
"   int i = get_global_id(0);                                       \n" \
"   if(i < count)                                                   \n" \
"       output[i] = input[i] * input[i];                              \n" \
"}                                                                    \n" \
"\n";

ComputeProgram = clCreateProgramWithSource (
        ComputeContext,                // a valid OpenCL context
        1,                             // the number of strings
        (const char**) &KernelSource,  // an array of strings            [1]
        NULL,                          // array of string lengths        [2]
        &err,                          // on return, points to a result code
);
```

Notes:

1. The kernel source code is included in the application as an array of strings. Each string can optionally be terminated with a `NULL` character.
2. If you don't end each string with a NULL, you must specify the length of each string. In this example, the strings are null terminated, so this parameter is `NULL`.

As you can see in Listing 3-7 (page 24), you literally pass each line of your OpenCL program source code as a C string to the `clCreateProgramWithSource` function, and it in turn creates a program object that contains your source code. At this point the program object is just holding your program source code—it is not executable until you compile and link it, which is covered in "Building a Program Executable" (page 26).

Once you have generated the program binary, you can use the `clGetProgramInfo` function to obtain the binary. If you cache the binary, the next time your program is run you can use the binary rather than the source to create the program object. Doing so will significantly reduce the initialization time for the application after the first time the application is run on a particular device.

Creating a program object from a binary is similar to doing so from source code, except you must manually supply the binary for each device on which you intend to execute your kernel. You can use the `clCreateProgramWithBinary` function to do this. Listing 3-8 shows the parameters for `clCreateProgramWithBinary`.

Listing 3-8 Calling the `clCreateProgramWithBinary` function

```
ComputeProgram = clCreateProgramWithBinary (
     ComputeContext,            // a valid OpenCL context
```

```
    numDevices,                // the number of devices                       [1]
    &devices,                  // a pointer to the list of device IDs         [2]
    (const size_t*) &lengths,  // an array of the sizes of the binaries       [3]
    (const void**) &binaries,  // an array of program binaries
    &binaryStatus,             // on return, points to result for each binary [4]
    &err,                      // on return, points to a result code
);
```

Notes:

1. See "Querying Available Compute Devices" (page 21).
2. As returned by `clGetDeviceIDs`.
3. Once you have generated the program binaries (see "Building a Program Executable" (page 26)), you can get information about them from the `clGetProgramInfo` function.
4. Because each type of compute device has an instruction set specific to that device, you must provide a separate binary for each device that you want to use. When you call the `clCreateProgramWithBinary` function, OpenCL validates each binary that you supply one-by-one with its corresponding compute device to ensure that the binary is suitable for that device. This parameter returns an array that tells you the result of this validation for each binary.

Note Program binaries can consist of either the device-specific executable, or the implementation-specific intermediate representation, which will later convert to the device-specific executable. Mac OS X v10.6 supports only device-specific binaries.

Building a Program Executable

After you have created a program object using either `clCreateProgramWithSource` or `clCreateProgramWithBinary`, you must build a program executable from the contents of that program object. Building the program compiles any source code that is in the program object and links the resultant machine code into an executable program. Use the `clBuildProgram` function shown in Listing 3-9 to build your executable.

Listing 3-9 Calling the `clBuildProgram` function

```
iError = clBuildProgram (
    program,                     // a valid program object
    0,                           // number of devices in next parameter
    NULL,                        // device list; NULL for all devices
    (const char*) &buildOptions, // a pointer to a string of build options
    NULL,                        // a pointer to a notification callback function
    NULL                         // data to be passed as a parameter to the
                                 //  callback function
);
```

The `clBuildProgram` function modifies the program object that you pass it to include an executable version of the program. Because of this, some program objects contain executables and some do not.

As with compiling any program source code, it is possible that you'll encounter errors when trying to build your OpenCL program. The OpenCL framework provides the `clGetProgramBuildInfo` function, which you can use to query the OpenCL compiler for details on the latest build. Listing 3-10 shows an example of how you can use `clBuildProgram` and `clGetProgramBuildInfo` together.

Listing 3-10 Checking for build errors

```
// create the opencl program from an inline source code buffer (KernelSource)
cl_program program = clCreateProgramWithSource(context, 1, (const char **) &
    KernelSource, NULL, &err);

// make sure that the program object was created successfully
if (!program)
{
    printf("Error: Failed to create compute program!\n");
    return -1;
}

// build the program executable
cl_int err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// check for errors
if (err != CL_SUCCESS)
{
    size_t len;
    // declare a buffer to hold the build info
    char buffer[2048];

    printf("Error: Failed to build program executable!\n");

    // get the details on the error, and store it in buffer
    clGetProgramBuildInfo(
      program,              // the program object being queried
      device_id,            // the device for which the OpenCL code was built
      CL_PROGRAM_BUILD_LOG, // specifies that we want the build log
      sizeof(buffer),       // the size of the buffer
      buffer,               // on return, holds the build log
      &len);                // on return, the actual size in bytes of the
                            //   data returned

    // print out the build errors
    printf("%s\n", buffer);
    exit(1);
}
```

In Listing 3-10 (page 27) the application uses the `CL_PROGRAM_BUILD_LOG` constant in order to request the details of the build errors. You can use the `clGetProgramBuildInfo` function to request other information as well, such as the build options used when calling `clBuildProgram`, or the status of the compilation. See the *OpenCL Specification* for all the different types of information you can request.

Note To test an OpenCL kernel quickly, use the OpenCL kernel patch in Quartz Composer. See the examples in `/Developer/Examples/Quartz Composer/Compositions/OpenCL`.

Creating Kernel Objects

A kernel is a function you declare in an OpenCL program. You must identify your kernels with the `__kernel` qualifier to let OpenCL know that the function is a kernel function and not a helper function. A kernel object encapsulates the specific kernel function declared in a program and can also encapsulate the parameter values to use when executing this kernel. In other words, the kernel itself is just a function, but a kernel object is an opaque data structure that includes the kernel function and (once you've called the `clSetKernelArg` function) the data on which the kernel operates. When you are ready to execute the kernel, it's the kernel object that you specify in a command queue.

You can call the the `clCreateKernel` function to create a single kernel object or call the `clCreateKernelsInProgram` function to create kernel objects for all the kernels in the OpenCL program.

The following sections provide an overview of how to go about preparing memory objects to hold data, associating data with the kernel object, and executing the kernel.

Creating Memory Objects

Memory objects are reserved regions of global device memory that can serve as containers for your data. After you've created and registered your kernels with the OpenCL runtime, you can send your application data to the kernels running on the various devices by first packaging the data in a memory object, and then associating that memory object with the specific kernel. There are two types of memory objects: buffer objects and image objects. Whereas buffer objects are simple blocks of memory, image objects are opaque structures, specifically for representing 2D or 3D images.

To create buffer objects, use the `clCreateBuffer` function. To create image objects, use the `clCreateImage2D` or `clCreateImage3D` function (whichever is appropriate for your data). The `clCreateBuffer`, `clCreateImage2D`, and `clCreateImage3D` functions all return an object that has the data type `cl_mem`. Upon creation, the contents of the memory objects are undefined. From your host application you must explicitly tell OpenCL to fill these memory objects with your data. To find out more about how to do this, refer to "OpenCL Memory Objects" (page 37).

Executing Kernels

OpenCL always executes kernels in a data-parallel fashion—that is, instances of the same kernel (referred to as *work-items*) execute on different portions of the total data set. (If you want task-parallel execution you must enqueue multiple kernels on different devices.) Each work-item is responsible for executing the kernel once and operating on its assigned portion of the data set. It is your responsibility to tell OpenCL the total number of work-items that you need to process all of your data. Because data sets are commonly organized in one, two, or three dimensions (representing such things as audio data streams, two- or three-dimensional images, or three-dimensional objects), you also need to indicate to OpenCL in how many dimensions your data extends (that is, how many coordinates to use for each data point).

Determining the Data Dimensions

The first step in preparing a kernel for execution is to identify the number of dimensions that you want to use to represent your data. For example, if your data represents a flat image that is *m* pixels wide by *n* pixels high, then you have a two-dimensional data set with each data pointed represented by its coordinates on the m and n axes. On the other hand, if you're dealing with spatial data that involves the *(x, y, z)* position of nodes in three-dimensional space, you have a three-dimensional data set. Another way to look at the dimensionality of your data is in terms of nested loops in a traditional, non-parallel processing model. If you can loop through your entire data set with a single loop, then your data is one-dimensional. If you would use one loop nested in another, your data is two-dimensional, and if you would have loops nested three deep in order to cycle through all your data, your data is three-dimensional. Whatever your data is, it's up to you to determine how many dimensions to use. As of OpenCL 1.0, dimensions greater than three are not supported.

Determining the Number of Work-Items

The next step in preparing your kernel for execution is determining how many work-items you'll need in order to process all of your data. This is known as the **global work size**, and it defines the total number of work-items needed for all dimensions combined. For one-dimensional data, the global work size equals the number of data items. For two-dimensional data with m data items in one dimension and n items in the second dimension, the global work size is $n * m$. Similarly, for three-dimensional data with x, y, and z work items in the three dimensions, the global work size is $x * y * z$. There is practically no limit on the number of work-items, and this should be a large number (over 1000) for good performance on GPUs.

Choosing a Work-Group Size

When enqueuing a kernel to execute on a device, you can specify the size of the work-group that you'd like OpenCL to use during execution. A work-group is a collection of work-items that execute on the same multiprocessor on the same OpenCL device (see "Platform Model" (page 10)). By providing OpenCL with a suggested work-group size, you are telling it how you would like it to delegate the work-items to the various computational units on the device. The work-items executing in the same work-group can share memory and execute synchronously. In order to take advantage of these features, however, you have to know the maximum work-group size allowed by the OpenCL device on which your work-items are executing. To get this information, you use the `clGetKernelWorkGroupInfo` function (as shown in Listing 3-11) and request the `CL_KERNEL_WORK_GROUP_SIZE` property. If you don't need to share data among work-items, pass a `NULL` value to the `local_work_size` parameter when you enqueue your kernel for execution to have OpenCL determine the work-group size for you. Doing so will ensure the most efficient use of the available devices.

Note that you also need to use the `clGetDeviceInfo` with the selector `CL_DEVICE_MAX_WORK_ITEM_SIZES`) to get the maximum work-group size in each dimension, and call the `clGetKernelWorkGroupInfo` function with the selector `CL_KERNEL_WORK_GROUP_SIZE` to get the total work-group size.

There are three conditions that need to be met for the local dimensions to be valid:

1. The number of work-items in each dimension (`local_x`, `local_y`, and `local_z`) in a single work-group must be less than the values returned for the device from `clGetDeviceInfo(CL_DEVICE_MAX_WORK_ITEM_SIZES)`.
2. The total number of work-items in a work-group (`local_x*local_y*local_z`) must be less than or equal to the value returned by `clGetKernelWorkGroupInfo(CL_KERNEL_WORK_GROUP_SIZE)`.
3. The number of work-items in each dimension in a single work-group must divide evenly into the total number of work-items in that dimension (`global_`*n* mod `local_`*n* = 0).

Listing 3-11 (page 31) is a routine from the *OpenCL Procedural Grass and Terrain Example* sample code project that illustrates the use of the `clGetKernelWorkGroupInfo` function.

Listing 3-11 Calling the `clGetKernelWorkGroupInfo` function

```
uint ComputeEngine::getEstimatedWorkGroupSize(
    const char *acKernelName,
    uint uiDeviceIndex)
{
    KernelMapIter pkKernelIter = m_akKernels.find(acKernelName);
    if(pkKernelIter == m_akKernels.end())
    {
        printf("Compute Engine: Failed to retrieve kernel for work group
                                                    size estimation!\n");
        return 0;
    }

    if(uiDeviceIndex >= m_uiDeviceCount)
    {
        printf("Compute Engine: Invalid device index specified for
                                         work group size estimation!\n");
        return 0;
    }

    size_t iMaxSize = 0;
    size_t kReturnedSize = 0;
    cl_kernel kKernel = pkKernelIter->second;
    int iError = clGetKernelWorkGroupInfo(
      kKernel,                         // the kernel object being queried
      m_akDeviceIds[uiDeviceIndex], // a device associated with the kernel object
      CL_KERNEL_WORK_GROUP_SIZE,    // requests the work-group size
      sizeof(iMaxSize),             // size in bytes of return buffer
      &iMaxSize,                    // on return, points to requested information
      &kReturnedSize);              // points to actual size of info returned

    if (iError)
    {
        printf("Compute Engine: Failed to retrieve kernel work group info!\n");
        ReportError(iError);
        return 0;
    }

    return iMaxSize;
}
```

Enqueuing Kernel Execution

After you've identified the dimensions necessary to represent your data, the necessary work-items for each dimension, and an appropriate work-group size, you can enqueue the kernel for execution. Listing 3-12 illustrates how this is done. See *OpenCL Parallel Prefix Sum (aka Scan) Example* for the complete program.

Listing 3-12 Enqueuing a kernel for execution

```
__kernel void                                      //[1]
PreScanKernel(
    __global float *output_data,
    __global const float *input_data,
    __local float* shared_data,
    const uint  group_index,
    const uint  base_index,
    const uint  n)
{ ... }
```

```
...                                                 //[2]

int PreScan(
    size_t *global,
    size_t *local,
    size_t shared,
    cl_mem output_data,
    cl_mem input_data,
    unsigned int n,
    int group_index,
    int base_index)
{
    unsigned int a = 0;

    int err = CL_SUCCESS;
    err |= clSetKernelArg(
       ComputeKernel,            // a valid kernel object
       sizeof(cl_mem),           // the size of the parameter value
       &output_data);            // a pointer to the parameter value
    err |= clSetKernelArg(ComputeKernel,  0, sizeof(cl_mem), &input_data);//[3]
    err |= clSetKernelArg(ComputeKernel,  1, shared,         0);
    err |= clSetKernelArg(ComputeKernel,  2, sizeof(cl_int), &group_index);
    err |= clSetKernelArg(ComputeKernel,  3, sizeof(cl_int), &base_index);
    err |= clSetKernelArg(ComputeKernel,  4, sizeof(cl_int), &n);
    if (err != CL_SUCCESS)
    { ... }

    err = CL_SUCCESS;
    err |= clEnqueueNDRangeKernel(
      ComputeCommands,         // a valid command queue
      ComputeKernel,           // a valid kernel object
      1,                       // the data dimensions                   [4]
      NULL,                    // reserved; must be NULL
      global,                  // work sizes for each dimension         [5]
      local,                   // work-group sizes for each dimension   [6]
```

```
        0,                          // num entires in event wait list        [7]
        NULL,                       // event wait list                       [8]
        NULL);                      // on return, points to new event object [9]
      if (err != CL_SUCCESS)
      { ... }
      return CL_SUCCESS;
}
```

Notes:

1. You call the `clSetKernelArg` function to provide values for the kernel function's parameters, indexed according the order of the parameters in the kernel declaration.
2. You must go through all the intermediate steps listed in "Essential Development Tasks" (page 16) and described in earlier sections in this chapter before proceeding to enqueue the kernel for execution.
3. Parameter index numbers start with `0`.
4. Data can be one, two, or three-dimensional. See "Determining the Data Dimensions" (page 30).
5. This parameter is an array indicating the size of each dimension of the data array for all the data being processed by this kernel. For example, if your data represented a two-dimensional image that is 64 pixels wide by 128 pixels high, then you would supply an array of dimensions `[64,128]`.
6. If you want to specify a work-group size, you must specify it as an array with the same number of dimensions as you used for the data. The values of this array must divide evenly into the array values for the global work size. For example, if your global work size is `[64,144]`, then your work-group-size array could be `[8,12]`, `[4,4]`, or `[32,24]`, but it could not be `[24,32]`. You do not have to provide a work-group size when enqueuing a kernel. Pass a `NULL` value to the `local_work_size` parameter to have OpenCL determine the work-group for you.
7. This and the next two parameters are used to control the sequencing of events if you specified out-of-order execution when you called the `clCreateCommandQueue` function. This parameter specifies the number of entries in the following parameter.
8. If you are using event objects to control the sequence of execution, you can specify in this parameter any events that must complete before this command is executed.
9. If you want to specify that some other command wait for completion of this command before executing, or you want to be able to query this kernel execution instance later on, then you need to obtain an event object for this execution instance.

Retrieving the Results

After your kernels have finished execution, you must read the results back from device to host memory.

Waiting for the Kernels to Complete Execution

Set the `blocking_read` parameter to `CL_TRUE` to ensure that the `clEnqueueReadBuffer` or `clEnqueueReadImage` command does not return until the data has been read and copied into memory. (Although you can use the `clFinish` function to tell your host application to block until all commands on a particular command queue have finished execution, this function is not good for performance.)

Note that within a given command queue, all commands execute in-order. You must synchronize or wait for commands executing on different queues (that is, on different devices). Note also that although you can read to and write from the same buffer object in a kernel, for image objects you must have separate objects for reading and writing.

If you want to wait for one kernel to finish executing and then enqueue another kernel on the same command queue, you can obtain an event object for the enqueued kernel execution instance and specify that the next command wait for that event object before executing. See, for example, the discussion in notes 7, 8, and 9 in "Executing Kernels" (page 29).

Alternatively, you can enqueue the `clEnqueueBarrier` command or memory fence commands (`mem_fence`, `read_mem_fence`, `write_mem_fence`) to synchronize commands within a work-group. To synchronize commands in different work groups, you can use event objects. See, for example, the descriptions of the `clWaitForEvents` and `clEnqueueWaitForEvents` commands in the *The OpenCL Specification*. Use the `clGetEventInfo` command to get information about a command, including its execution status.

Reading the Results

After your kernel has completed execution, you can read data from the device back to the host where it is accessible to the host application. To read the data, call the `clEnqueueReadBuffer` function or the `clEnqueueReadImage` function, depending on what type of memory object you created to store the output results (see "OpenCL Memory Objects" (page 37)).

Note Unless you plan to use the same memory object to hold the input and output of your kernel function, you should create a memory object that's reserved for storing output data, and then set this as an argument to the kernel. In the context of the kernel function, you can read from the input buffer and write to the output buffer. When the kernel finishes executing, from the context of your host application you can enqueue a command to read from that output memory object to bring the results to host memory.

Cleaning Up

After your host application no longer requires the various objects associated with the OpenCL runtime and context, it should free these resources. You can use the following functions to release your OpenCL objects. Each of these functions decrements the reference count for the associated object. Assuming that you've retained and released your memory appropriately, the reference counts for all OpenCL objects should reach zero when your application no longer needs them, returning their associated resources back to the system.

- `clReleaseMemObject`
- `clReleaseKernel`
- `clReleaseProgram`
- `clReleaseCommandQueue`
- `clReleaseContext`

Debugging

Here are a few hints to help you debug your OpenCL application:

- Run your kernel on the CPU first.
- You can use the `printf` function from within your kernel.
- You can use the gdb debugger to look at the assembly code once you've built your program.
- You can use the Shark utility to tune the performance of your application.
- On the GPU, use explicit address range checks to look for out-of-range address accesses. (Remember: there is no memory protection on current GPUs.)
- Make sure your kernels don't take too long to execute on the GPU, as the GPU is a shared resource and long-running kernels negatively impact overall system responsiveness.

OpenCL Memory Objects

Memory objects are reserved regions of global device memory that can serve as containers for your data. There are two types of memory objects: buffer objects and image objects. Whereas buffer objects are for containing any type of generic data, image objects are specifically for representing 2D or 3D images. This chapter discusses both types of memory object.

Representing Data with Buffer Objects

The OpenCL programming interface provides buffer objects for representing generic data in your OpenCL programs. Instead of having to convert your data to the domain of a specific type of hardware, OpenCL enables you to transfer your data as is to an OpenCL device via buffer objects and operate on the data using the same language features that you are accustomed to in C.

There are two principal ways for a kernel to access your host application data: It can follow a host pointer to the data, in which case traveling along a PCI bus might be necessary, or you can copy all of the host data to device memory first and then the kernel can access it locally. Because transmitting data is costly, it is best to minimize reads and writes as much as possible. By packaging all of your host data into a buffer object that can remain on the device, you reduce the amount of data traffic necessary to process your data.

Allocating Buffer Objects

Before you can store your application data in a buffer object, you must first use the `clCreateBuffer` function to create the buffer object, as shown in Listing 4-1.

Listing 4-1 Allocating buffer objects

```
memobjs[0] = clCreateBuffer(context,
                          CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
                          sizeof(cl_float4) * n, srcA, NULL);
if (memobjs[0] == (cl_mem)0)
{
    clReleaseCommandQueue(cmd_queue);
    clReleaseContext(context);
```

```
        return -1;
    }

    memobjs[1] = clCreateBuffer(context,
                              CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                              sizeof(cl_float4) * n, srcB, NULL);
    if (memobjs[1] == (cl_mem)0)
    {
        delete_memobjs(memobjs, 1);
        clReleaseCommandQueue(cmd_queue);
        clReleaseContext(context);
        return -1;

    }

   memobjs[2] = clCreateBuffer(context,
                                CL_MEM_READ_WRITE,
                                sizeof(cl_float) * n, NULL, NULL);
    if (memobjs[2] == (cl_mem)0)
    {
        delete_memobjs(memobjs, 2);
        clReleaseCommandQueue(cmd_queue);
        clReleaseContext(context);
        return -1;
    }
```

Note that this example does not show error handling. You should request an error code and check it. It is not sufficient to check whether the memory object returned is `NULL`.

In this example, the first read buffer is allocated with the `CL_MEM_USE_HOST_PTR` flag set. In contrast, the second read buffer is allocated with `CL_MEM_COPY_HOST_PTR` flag set. In both cases, you must also provide a pointer to your data. When the `CL_MEM_USE_HOST_PTR` flag is set, the OpenCL implementation has the option of caching the data on the OpenCL device, but it keeps the buffers on the two devices synchronized; when that flag is not set, it always allocates the memory on the host device. When the `CL_MEM_COPY_HOST_PTR` flag is set, on the other hand, the OpenCL implementation allocates the buffer on the device. In either case, it is initialized from the data in host memory pointed to by the fourth parameter. If you set the

`CL_MEM_USE_HOST_PTR` flag, you can force OpenCL to allocate the data on the host device by also specifying the `CL_MEM_ALLOC_HOST_PTR` option. You can use these options to initialize the memory buffer, to synchronize memory buffers, and to make data accessible to multiple applications. However, keep in mind that transferring data between devices is costly.

It is perfectly acceptable to create a buffer object without specifying a corresponding pointer to data on the host device. By providing the `clCreateBuffer` function with `NULL` values for the options and for the host pointer, you create a buffer object that is independent of any pointers on the host. If there is specific host data that you'd like to place in that buffer object, you can do so by enqueuing a command to write to the buffer object using the `clEnqueueWriteBuffer` function, as discussed in the following section, "Reading, Writing, and Copying Buffer Objects."

Reading, Writing, and Copying Buffer Objects

After you've created the buffer object, you can enqueue reads, writes, and copies. From your host application, you can use the following functions:

- `clEnqueueReadBuffer`

  This function enqueues a command to read data from a buffer into host memory. This is useful for reading the output results of a kernel back to the host application.

- `clEnqueueWriteBuffer`

  This function enqueues a command to write data from host memory to a buffer. You can use this function to provide data for processing by a kernel executing on the device.

- `clEnqueueCopyBuffer`

  This function copies data from one buffer object to another.

Important The read, write, and copy commands `clEnqueueRead*`, `clEnqueueWrite*`, and `clEnqueueCopy*` functions only enqueue the memory commands; they don't block by default. To know when the command has completed so that's you can be sure the data is available and that it's safe to free the memory, you either need to get an event and check the command's status from that event, or use a blocking form of the command by setting the `blocking_*` parameter in the function call to `CL_TRUE`.

These functions enable you to move data to and from a host. To actually process this data on a device, you have to make this data available to the work-items that execute on the device. The following sections show you how to pass your data to the compute kernels for further processing.

Accessing the Buffer Objects from a Kernel

After your data has been successfully transferred, to access it in a kernel you must then explicitly tell OpenCL to pass the specified buffer object as an argument to the specific kernel function that you defined in your OpenCL program source code. These functions are identified with the `__kernel` qualifier. You can do this by using the `clSetKernelArg` function as shown in Listing 5-1 (page 47).

Once you've associated the appropriate buffer objects with the appropriate kernel arguments, the next time you execute that kernel (using a function such as `clEnqueueNDRangeKernel`), the kernel function receives the buffer objects supplied to the `clSetKernelArg` function as input.

For example, imagine that you have written a kernel function in OpenCL called "`square`" that takes an input value, multiplies it by itself, and then stores the resultant value as output. The source code for such a kernel could read as follows:

Listing 4-2 A squaring kernel function

```
__kernel square(
    __global float* input,
    __global float* output,
    const unsigned int count)
{
    int i = get_global_id(0);
    if(i < count)
        output[i] = input[i] * input[i];
}
```

In your host application source code, it's your responsibility to :

- Prepare the input data.
- Create the buffer object. Use the `clCreateBuffer` function to create a buffer object of the appropriate size.
- Move the input data from host memory. You can do this in the `clCreateBuffer` function by pointing to the data on the host, or you can use the `clEnqueueWriteBuffer` function to enqueue a write from host memory.
- Associate the input data with the kernel's arguments. Use the `clSetKernelArg` function to do this.

Listing 5-1 (page 47) shows an example of how to do this in the host application.

Processing Data in OpenCL

By associating your buffer object with specific kernel arguments, you make it possible to process your data from the context of a kernel function. For example, in Listing 4-2 (page 40), notice how the code sample treats the input data pointer much as you would treat a pointer in C. In this example the input data is an array of `float` values, and you can process each element of the `float` array by indexing into the pointer. Listing 4-2 (page 40) does little more than multiply a value by itself using the `*` operator, but OpenCL-C provides a wide array of data types and operators that enable you to perform more complex arithmetic.

Because OpenCL-C is based on C99, you are free to process your data in OpenCL functions as you would in C with few limitations. Aside from support for recursion and function pointers, there are not many language features that C has that OpenCL doesn't have. In fact, OpenCL provides several beneficial features that the C programming language does not offer natively, such as optimized image access functions.

OpenCL has built-in support for vector intrinsics and offers vector data types. The operators in OpenCL are overloaded, and performing arithmetic between vector data types is syntactically equivalent to performing arithmetic between scalar values. Refer to the *The OpenCL Specification* for more details on the built-in functions and facilities of the OpenCL-C language.

When you are done processing your data and writing these results to an output buffer, your host application can read this data back into host memory using the `clEnqueueReadBuffer` function or the `clEnqueueReadImage` function, depending on what type of memory object you created to store the output results.

Retaining and Releasing Buffer Objects

Buffer objects should be freed when no longer needed to avoid memory leaks. OpenCL uses a reference counting system to keep track of the memory objects currently being used. The reference count represents how many other objects hold references to the particular memory object. Any time you create a buffer object, it immediately receives a reference count of 1. Any time another object would also like to maintain a reference to it, it should increment the buffer object's reference count by calling the `clRetainMemObject` function. When an object wishes to relinquish its reference to a buffer object, it should call `clReleaseMemObject`. When the reference count for a buffer object reaches zero, OpenCL frees it, returning the memory to the system and making any persisting references to the buffer object invalid.

Image Objects

OpenCL has built-in support for processing image data. Using image objects, you can take image data that resides in host memory and make it available for processing in a kernel executing on an OpenCL device. Image objects simplify the process of representing and accessing image data since they offer native support for a multitude of image formats. If you are writing kernel functions that need to efficiently perform calculations on image data, you will find OpenCL native support for images useful.

The following sections show you how to take your image data that resides in host memory and place it in image objects that you can later access within a kernel. It also provides an overview of how to go about processing this image data.

Representing Two-Dimensional Images

In order to be able to process an image in a kernel, you need to create an image object. Creating an image object allocates memory specifically tailored to holding image data. For example, your host application can use the `clCreateImage2D` function (shown in Listing 4-3, taken from the *OpenCL Procedural Grass and Terrain Example* sample code project) to create a two-dimensional image object.

Listing 4-3 Creating a 2D image object

```
ComputeEngine::createImage2D(
    const char* acMemObjName,
    MemFlags eMemFlags,
    ChannelOrder eOrder,
    ChannelType eType,
    uint uiWidth,
    uint uiHeight,
    uint uiRowPitch,
    void* pvData)
{
    uint uiChannelCount = getChannelCount(eOrder);
    if(uiChannelCount == 0)
        return false;

 // set the image format properties and option flags
    cl_image_format kFormat                        ;
    kFormat.image_channel_order = (cl_channel_order) eOrder;
    kFormat.image_channel_data_type = (cl_channel_type) eType;
    cl_mem_flags kFlags = (cl_mem_flags) eMemFlags;

    int iError = CL_SUCCESS;
    cl_mem kImage = clCreateImage2D(
          m_kContext,         // a valid OpenCL context
          kFlags,             // option flags                  [1]
          &kFormat,           // image format properties       [2]
          (size_t)uiWidth,    // width of the image in pixels
          (size_t)uiHeight,   // height of the image in pixels
          (size_t)uiRowPitch, // scan-line pitch in bytes      [3]
          pvData,             // pointer to the image data
          &iError             // on return, the result code);

    if(kImage == 0 || iError != CL_SUCCESS)
    { ... }

    m_akMemObjects[acMemObjName] = kImage;
    return true;
}
```

Notes:

1. The same options as you can use for buffer objects, such as read-only, write-only, or allocate memory on the host. See *The OpenCL Specification* for details.
2. These properties include number of channels, channel order, and channel data type. Examples of values that set the number of channels and channel order are `CL_RG`, `CL_RGBA`, and `CL_BGRA`, where `R` is red, `G` is green, `B` is blue, and `A` is alpha. See *The OpenCL Specification* for the complete list of possible property values.
3. The scan-line pitch, also referred to as row pitch, represents how many bytes are necessary to represent one row (or scan line) of the image. You need to specify the row pitch only if the data from which you are copying the image has a particular row pitch. Internally, OpenCL uses a storage format optimized for the device. Specify `0` if you want OpenCL to calculate the row-pitch value for you.

Calculating Row Pitch

To calculate row pitch, take the width of the image in pixels and multiply it by the number of bytes in each pixel. For example, an image that is in `CL_RGBA` format has four separate image channels per pixel: red, green, blue, and the alpha channel. The pseudocode in Listing 4-4 shows how you can calculate the row pitch of an image. If the pointer to a preexisting image buffer is not `NULL` and you specify `0` for the row pitch parameter, then OpenCL calculates the row pitch as the image width * the size of a pixel element in bytes.

Listing 4-4 Calculating the row pitch for an image in 8-bit RGBA format

```
// assume that each channel is represented with in CL_RGBA / CL_UNORM_INT8 format
num_channels_per_pixel = 4;
image_width = ... ; // the width of the image
channel_size = sizeof(uint8);
pixel_size = channel_size * num_channels_per_pixel;
image_row_pitch = image_width * pixel_size;
```

Representing Three-Dimensional Images

To create an image object that represents three-dimensional data, you must specify the image depth along with the height and width as you do for two-dimensional images. However instead of providing only the row pitch as when creating a two-dimensional image object, you must also provide OpenCL with the *slice pitch* of the three-dimensional image. The slice pitch represents the size, in bytes, of each two-dimensional slice of the image. You can compute this by taking the row pitch of the image and multiplying this by the height of the image. With the image depth and the slice pitch you provide OpenCL with information on the general geometry of the three-dimensional space your image occupies. You can think of it as similar to defining a bounding cube around your three-dimensional image.

Aside from requiring the image depth and slice pitch, calling the `clCreateImage3D` function is essentially identical to creating a two-dimensional image object .

Reading, Writing, and Copying Image Objects

After you've created the image object, you can enqueue reads, writes, and copies to and from host memory. From your host application, you can use the following functions:

- `clEnqueueReadImage`

  This function enqueues a command to read data from an image object into host memory. This is useful for reading the output results of a kernel back to the host application.

- `clEnqueueWriteImage`

This function enqueues a command to write data from host memory. You can use this function to provide image data for processing by a kernel executing on the device.

- `clEnqueueCopyImage`

  This function copies data from one image object to another.

Important The read, write, and copy commands `clEnqueueRead*`, `clEnqueueWrite*`, and `clEnqueueCopy*` functions only enqueue the memory commands; they don't block by default. To know when the command has completed so that's you can be sure the data is available and that it's safe to free the memory, you either need to get an event and check the command's status from that event, or use a blocking form of the command by setting the `blocking_*` parameter in the function call to `CL_TRUE`.

These functions enable you to move images to and from host memory. To actually process this image data on a device, you have to make this data available to the work-items that execute on the device. The following sections show you how to pass your data to the kernels for further processing.

Accessing the Image Objects from a Kernel

After your data has been successfully transferred as an image object, to access it in a kernel you must then explicitly tell OpenCL to pass the specified image object as an argument to the specific kernel function that you defined in your OpenCL program source code. These functions are identified with the `__kernel` qualifier. You can do this by using the `clSetKernelArg` function as shown in Listing 5-1 (page 47).

Once you've associated the appropriate image objects with the appropriate kernel arguments, the next time you execute that kernel (using a function such as `clEnqueueNDRangeKernel`), the kernel function receives the image objects supplied to the `clSetKernelArg` function as input.

For example, imagine that you have written a kernel function in OpenCL called "`foo`" that takes two arguments: an input two-dimensional image, and an output two-dimensional image. The signature for such a kernel could read as follows in Listing 4-5.

Listing 4-5 A image-processing kernel function

```
__kernel void foo (
    read_only image2d_t imageA,
    write_only image2d_t imageB)
{
    ...
}
```

Notice the `read_only` and `write_only` qualifiers that precede the `imageA` and `imageB` arguments, respectively. These are examples of image access qualifiers that you can declare in your kernels to enforce read-only or write-only access to a certain image. The default qualifier is `read_only`. You can find more details on this and other argument qualifiers in *The OpenCL Specification* .

Note Whereas a buffer object can be used for both reads and writes in a given kernel, an image object can be used only for read or writes, not both.

Processing Images in OpenCL

By associating your image object with specific kernel arguments, you make it possible to process your image data from the context of a kernel function. When you use image objects as the arguments to a kernel, they assume the `image2d_t` or `image3d_t` data type. Instead of accessing the buffer of image data directly, you must use the built-in OpenCL read and write image functions such as `read_imagef` and `write_imagef`. The `read_image*` functions return a four component floating-point, integer, or unsigned integer color value. OpenCL identifies the color channels as `x`, `y`, `z`, and `w`, where the `x` component refers to the red channel, the `y` component refers to the green channel, the `z` component refers to the blue channel, and the `w` component refers to the alpha channel. You can find more details on the use of these image functions in the *The OpenCL Specification* .

When you are done processing your image data and writing these results to the output image, the host application can read the image back into host memory using functions such as `clEnqueueReadImage`.

Retaining and Releasing Image Objects

Image objects should be freed when no longer needed to avoid memory leaks. Image objects are retained and released in the same manner as other memory objects; see "Retaining and Releasing Buffer Objects" (page 41) for details.

Basic Programming Sample

This chapter guides you through the creation of a simple OpenCL application that performs calculations on a data set.

Basic Code Sample

Error handling and input data have been left out of this example to save space. For a fully-functioning version of this program, see *OpenCL Hello World Example* .

Listing 5-1 Basic OpenCL code sample

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <OpenCL/opencl.h>

////////////////////////////////////////////////////////////////////////////////

// Use a static data size for simplicity
//
#define DATA_SIZE (1024)

////////////////////////////////////////////////////////////////////////////////

// Simple compute kernel that computes the square of an input array.       [1]
//
```

```
const char *KernelSource = "\n" \
"__kernel square(                                                      \n" \
"   __global float* input,                                             \n" \
"   __global float* output,                                            \n" \
"   const unsigned int count)                                          \n" \
"{                                                                     \n" \
"   int i = get_global_id(0);                                          \n" \
"   if(i < count)                                                      \n" \
"       output[i] = input[i] * input[i];                               \n" \
"}                                                                     \n" \
"\n";

////////////////////////////////////////////////////////////////////////////////

int main(int argc, char** argv)
{
    int err;                            // error code returned from api calls

    float data[DATA_SIZE];              // original data set given to device
    float results[DATA_SIZE];           // results returned from device
    unsigned int correct;               // number of correct results returned

    size_t global;                      // global domain size for our calculation
    size_t local;                       // local domain size for our calculation

    cl_device_id device_id;             // device ID
    cl_context context;                 // context
    cl_command_queue queue;             // command queue
    cl_program program;                 // program
    cl_kernel kernel;                   // kernel

    cl_mem input;                       // device memory used for the input array
    cl_mem output;                      // device memory used for the output array
    // Get data on which to operate
    //

   int i = 0;
    unsigned int count = DATA_SIZE;
    for(i = 0; i < count; i++)
        data[i] = ...;
    ...

    // Get an ID for the device                                   [2]
    int gpu = 1;
    err = clGetDeviceIDs(NULL, gpu ? CL_DEVICE_TYPE_GPU : CL_DEVICE_TYPE_CPU, 1,
                                                        &device_id, NULL);
    if (err != CL_SUCCESS)
    { ... }                                                       //      [3]

    // Create a context                                           [4]
    //
    context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
    if (!context)
    { ... }

    // Create a command queue                                             [5]
    //
    queue = clCreateCommandQueue(context, device_id, 0, &err);
    if (!queue)
    { ... }

    // Create the compute program from the source buffer                  [6]
    //
    program = clCreateProgramWithSource(context, 1,
                            (const char **) & KernelSource, NULL, &err);
    if ( program)
    { ... }
```

```
// Build the program executable                                  [7]
//
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
if (err != CL_SUCCESS)
{
    size_t len;
    char buffer[2048];

    printf("Error: Failed to build program executable\n");       [8]
    clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG,
                                   sizeof(buffer), buffer, &len);
    printf("%s\n", buffer);
    exit(1);
}

// Create the compute kernel in the program we wish to run      [9]
//
kernel = clCreateKernel(program, "square", &err);
if (!kernel || err != CL_SUCCESS)
{ ... }

// Create the input and output arrays in device memory for our calculation
//                                                              [10]
input = clCreateBuffer(context,  CL_MEM_READ_ONLY,  sizeof(float) *count,
                                                               NULL, NULL);
output = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(float) *count,
                                                               NULL, NULL);
if (!input || !output)
{ ... }

// Write our data set into the input array in device memory     [11]
//
err = clEnqueueWriteBuffer(queue, input, CL_TRUE, 0,
                              sizeof(float) *count, data, 0, NULL, NULL);
```

```
if (err != CL_SUCCESS)
{ ... }

// Set the arguments to our compute kernel                          [12]
//
err = 0;
err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &output);
err |= clSetKernelArg(kernel, 2, sizeof(unsigned int), &count);
if (err != CL_SUCCESS)
{ ... }

// Get the maximum work-group size for executing the kernel on the device
//                                                                  [13]
err = clGetKernelWorkGroupInfo(kernel, device_id, CL_KERNEL_WORK_GROUP_SIZE,
                                                  sizeof(int), &local, NULL);
if (err != CL_SUCCESS)
{ ... }

// Execute the kernel over the entire range of the data set          [14]
//
global = count;
err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global, &local,
                                                          0, NULL, NULL);
if (err)
{ ... }

// Wait for the command queue to get serviced before reading back results
//                                                                  [15]
clFinish(queue);

// Read the results from the device                                 [16]
//
err = clEnqueueReadBuffer(queue, output, CL_TRUE, 0,
                            sizeof(float) *count, results, 0, NULL, NULL );
```

```
    if (err != CL_SUCCESS)
    { ... }

    // Shut down and clean up
    //
    clReleaseMemObject(input);
    clReleaseMemObject(output);
    clReleaseProgram(program);
    clReleaseKernel(kernel);
    clReleaseCommandQueue(queue);
    clReleaseContext(context);

    return 0;
}
```

Notes:

1. To avoid having to specify the length of each string, null-terminate the strings.
2. In this example, we're only looking for one device, which may be a GPU or a CPU. Depending on the kernel you're trying to run, you might want to execute it preferentially on a GPU if one is available, for example. You can use the information returned by this function to do so.
3. Error handling code omitted for sake of brevity.
4. You can specify a callback function to handle errors, but in this code sample we're not providing one.
5. Our command queue is specified for the device and context we've already set up. We are also specifying that the commands are to be executed strictly in order. This code sample uses only one command queue. Note that this function only creates the command queue, it does not put any commands into the queue. Functions that put commands into the command queue begin with the string `clEnqueue`... There are several examples of such commands later in this code sample.
6. This code sample has only a single kernel function (identified by the `__kernel` keyword) in the kernel program. If you had included more than one kernel in your source, this function would concatenate them into a single program. You can specify any number of strings at any number of locations for your source code. In this code sample, all the source code is together and all strings are null-terminated to avoid having to specify a pointer and length for each string.

7. This function links and compiles your program for the device or devices you specify. You can also specify a callback function to handle errors; in this code sample we do not specify a callback function. Once you have generated the program binary, you can use the `clGetProgramInfo` function to obtain the binary. If you cache the binary, the next time your program is run you can use the `clCreateProgramWithBinary` function rather than the `clCreateProgramWithSource` function to create the program object. Doing so will significantly reduce the initialization time for the application after the first time the application is run on a particular device.
8. If the build failed, you can read the build log to get information about what went wrong.
9. You can create each kernel object separately by using this function and specifying the kernel function name of each kernel. Alternately, you can call the `clCreateKernelsInProgram` function to create kernel objects for all the kernels in the OpenCL program. In this code sample, there's only a single kernel in the program, so the two functions would be equivalent.
10. In this sample, the buffers are allocated with no pointer to a host buffer and without any option that would require the buffer to be allocated on the host. For more examples of the use of buffer allocation functions, see "Representing Data with Buffer Objects" (page 37).
11. This function enqueues the command that writes the kernel from the host's memory. This function call specifies `CL_TRUE` for the `blocking_write` parameter. For a write, that means that the function does not return until the data has been copied and the write has been enqueued. The actual write might or might not have been completed when this function returns.
12. The parameters for the kernel function are provided in order as declared in the kernel source, counting from `0` for the first parameter.
13. OpenCL determines the maximum work-group size based on the capabilities of the device specified and the requirements of the kernel. In this function call, the maximum work-group size is returned in the `local` variable.
14. This function enqueues a command that runs the kernel on the device. In this sample, the data set is defined as one-dimensional and we're using the maximum number of work-group items for the specified device (stored in the `local` variable).
15. The `clFinish` function blocks until all queued commands in the command queue have been processed and completed.
16. Now we have to read the results of the kernel calculations back onto the host device. In this function call, `output` is the buffer on the device into which results were written, and `result` is the buffer on the host device into which this function transfers the results. Note that we are reading from a buffer object that was allocated with the `CL_MEM_WRITE_ONLY` option. The read-only and write-only options refer to reads and writes by kernel functions, not by the host.

    In order for an application to gain an advantage from using OpenCL, the amount of time saved by doing computations on the compute devices must be greater than the time required to transfer data to and from the device.

Document Revision History

This table describes the changes to *OpenCL Programming Guide for Mac OS X* .

Date	Notes
2009-06-10	New document for using OpenCL in programs that use the parallel-processing power of GPUs and multi-core CPUs for general-purpose computing.