OpenCL Programming Guide for Mac

Developer

Contents

Figures, Tables, and Listings

About OpenCL for OS X

OpenCL™ (Open Computing Language) is an open standard for cross-platform, programming of modern highly-parallel processor architectures. Introduced with OS X v10.6, OpenCL consists of a C99-based programming language designed for parallelism, a powerful scheduling API, and a flexible runtime that executes kernels on the CPU or GPU. OpenCL lets your application harness the computing power of these processors to improve performance and deliver new features based on compute-intensive algorithms.

In addition to support for the OpenCL 1.1 standard, OS X v10.7 adds integration between OpenCL, Grand Central Dispatch (**GCD**), and Xcode to make it even easier to use OpenCL in your application.

At a Glance

Using OpenCL is easier than ever as of OS X v10.7:

- OpenCL is fully supported by Xcode. The Xcode offline compiler removes a configuration step that used to have to be performed before the kernel could be run and facilitates debugging earlier in the development process. See "Hello World!" (page 10).
- You can write OpenCL functions in separate files and include them in your Xcode project. You can compile the kernels when your application is built, before it runs. This improves runtime performance.
- OpenCL now integrates with GCD, making it easier for you to focus on making your OpenCL kernels more efficient. See "Using Grand Central Dispatch With OpenCL" (page 39).
- The autovectorizer compiles and accelerates performance of kernels that run on the CPU up to four times without additional effort. The autovectorizer allows you to write one kernel that runs efficiently on both a CPU and a GPU. You can invoke the autovectorizer regardless of whether you are compiling from Xcode or building the kernels at runtime. Or you can disable the autovectorizer if necessary. See "Autovectorizer" (page 101).

You can, of course, continue to use code you've already written to the OpenCL 1.1 standard. But see "Binary Compatibility Of OpenCL Kernels" (page 145) for a note about how to handle existing binaries.

Because OpenCL C is based on C99, you are free to process your data in OpenCL C 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 C doesn't have. In fact, OpenCL C provides several beneficial features that the C programming language does not offer natively, such as optimized image access functions. OpenCL C has

built-in support for vector intrinsics and offers vector data types. The operators in OpenCL C 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.

Prerequisites

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

See Also

The OpenCL Specification, available from the Khronos Group at http://www.khronos.org/registry/cl/ provides information on the OpenCL standard.

The *OpenCL Programming Guide* by Aaftab Munshi, Benedict Gaster, Timothy G. Mattson, James Fung, and Dan Ginsburg, available from Pearson Education, Inc., is a helpful introduction to the OpenCL language and standard; these topics are not discussed in this book.

For more information about Grand Central Dispatch queues, see Concurrency Programming Guide: Dispatch Queues.

Hello World!

Support built into Xcode in OS X v10.7 and later makes developing OpenCL applications much easier than it used to be. This chapter describes how to create an OpenCL project in Xcode. (You don't have to regenerate OpenCL projects that are already working.)

Creating An Application That Uses OpenCL In Xcode

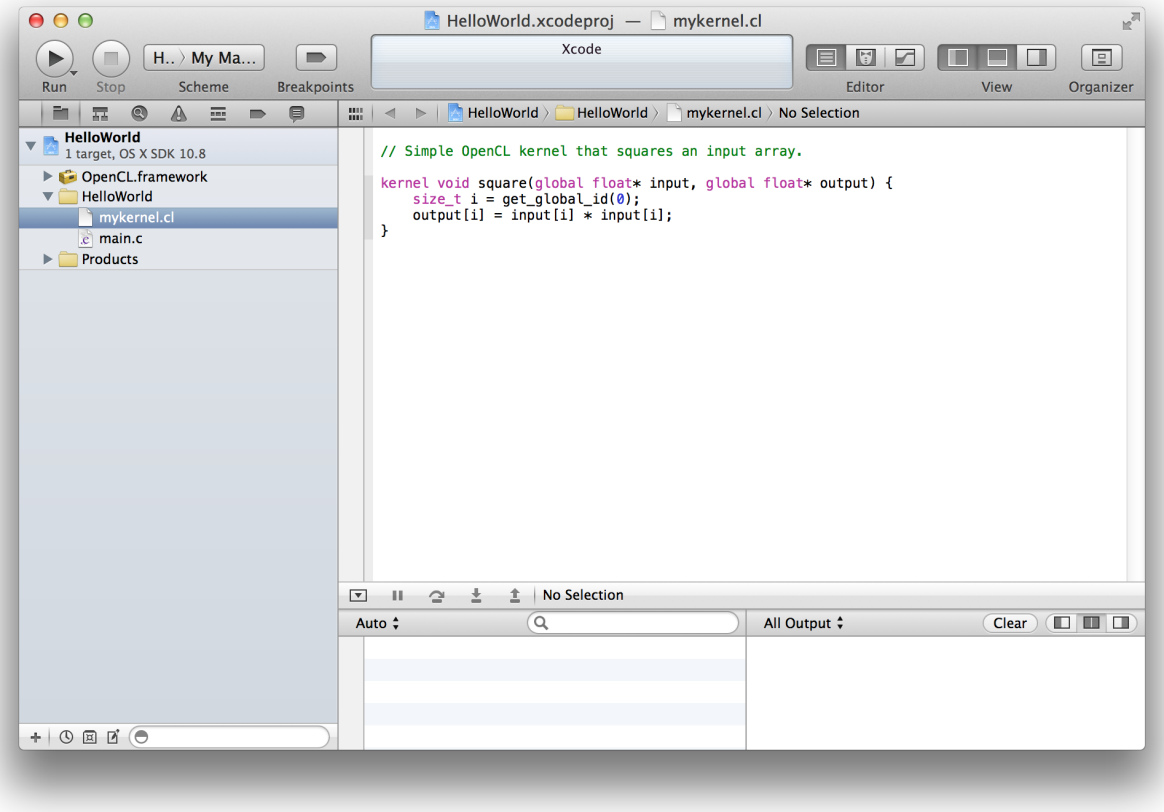
To create a project that uses OpenCL in OS X v10.7 or later:

1. Create your OpenCL project in Xcode as a new OS X project (empty is fine).
2. Place your kernel code in one or more .cl files in your Xcode project. You can place all your kernels into a single .cl file, or you can separate them as you choose. You can also include non-kernel code that will run on the same OpenCL device as the kernel in each .cl file.

   Each .cl file is compiled by default into three files containing bitcode for i386, x86_64, and gpu_32 architectures. (You can change which bitcodes are generated using the **OpenCL Architectures** build setting.)

At runtime your host application discovers the available devices and determines which of the compiled kernels to enqueue and execute. Figure 1-1 (page 11) shows a very simple OpenCL project in Xcode.

Figure 1-1 A simple OpenCL kernel in Xcode



3. You can set the following build settings for your OpenCL apps:
    - **OpenCL—Build**
        - **OpenCL Architectures**. The default is that the product is built for all three architectures. The dropdown allows you to choose up to three of `-triple i386-applecl-darwin`, `-triple x86_64-applecl-darwin`, and `-triple gpu_32-applecl-darwin`.
        - **OpenCL Compiler Version**. The default is OpenCL C 1.1.
    - **OpenCL—Code Generation**
        - **Auto-vectorizer**. Choose `Yes` to turn the autovectorizer on or `No` to turn the autovectorizer off. This setting takes effect only for the CPU.
        - **Double as single**. If you set this parameter to `Yes`, the compiler treats double-precision floating-point expressions as single-precision floating-point expressions. This option is available for GPUs only. The default is `No`.

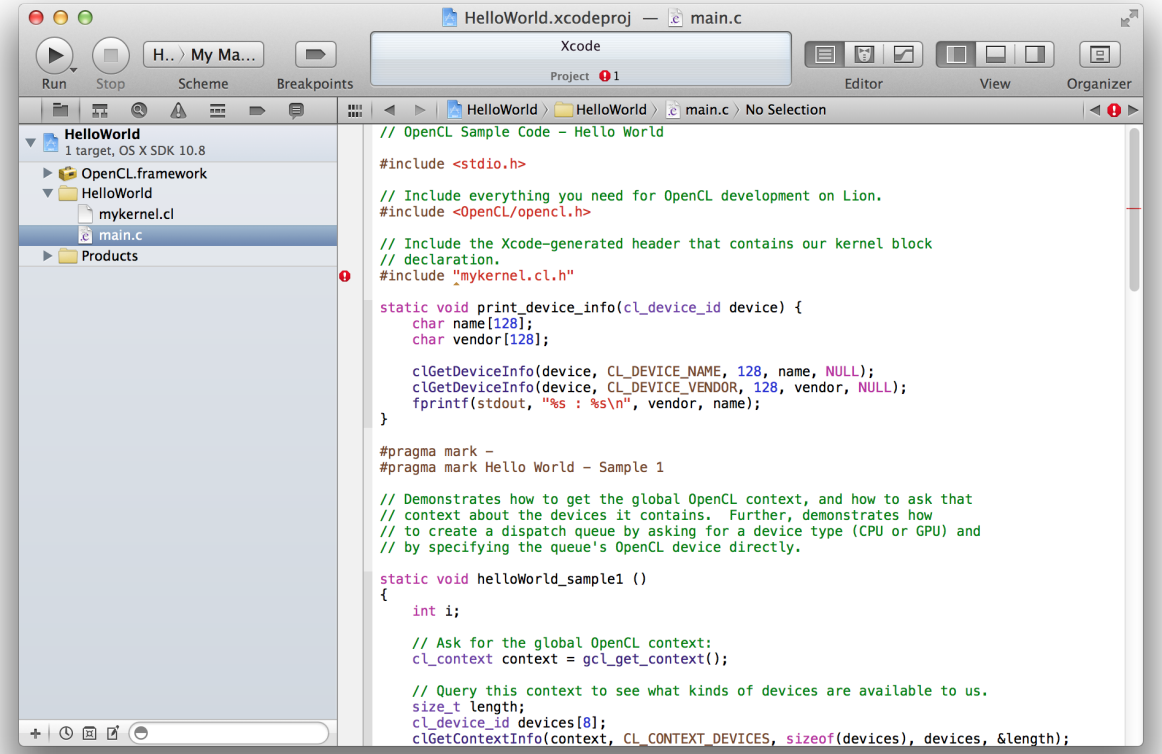
- **Flush denorms to zero**. This Boolean controls how single- and double-precision denormalized numbers are handled. If you set this parameter to `Yes`, the compiler may flush single-precision denormalized numbers to zero; it may also flush double-precision denormalized numbers to zero if the optional extension for double-precision is supported. This is intended to be a performance hint and the OpenCL compiler can choose not to flush denorms to zero if the device supports single-precision (or double-precision) denormalized numbers (that is, if the `CL_FP_DENORM` bit is not set in `CL_DEVICE_SINGLE_FP_CONFIG`). This flag only applies for scalar and vector single-precision floating-point variables and computations on these floating-point variables inside a program. It does not apply to reading from or writing to image objects. The default is `No`.
- **Optimization Level**. You can choose between several types of optimization from fastest performance to smallest code size. The default is to optimize for fastest performance.
- **Relax IEEE Compliance**. If you set this parameter to `Yes`, the compiler allows optimizations for floating-point arithmetic that may violate the IEEE 754 standard and the OpenCL numerical compliance requirements defined in section 7.4 for single-precision floating-point, section 9.3.9 for double-precision floating-point, and edge case behavior as defined in section 7.5 of the OpenCL 1.1 specification. This is intended to be a performance optimization. This option causes the preprocessor macro `__FAST_RELAXED_MATH__` to be defined in the OpenCL program. The default is `No`.
- **Use MAD**. If you set this parameter to `Yes`, you allow expressions of the type `a * b + c` to be replaced by a Multiply-Add (MAD) instruction. If MAD is enabled, multistep instructions in the form `a * b + c` are performed in a single step, but the accuracy of the results may be compromised. For example, to optimize performance, some OpenCL devices implement MAD by truncating the result of the `a * b` operation before adding it to `c`. The default for this parameter is `No`.

- **OpenCL—Preprocessing**

  You can enter a space-separated list of preprocessor macros of the form `"foo"` or `"foo=bar"` here if you wish.

4. Place your host code in one or more .c files in your Xcode project. "Compiling From the Command Line" (page 15) shows host code in an Xcode project.

Figure 1-2 OpenCL host code in Xcode



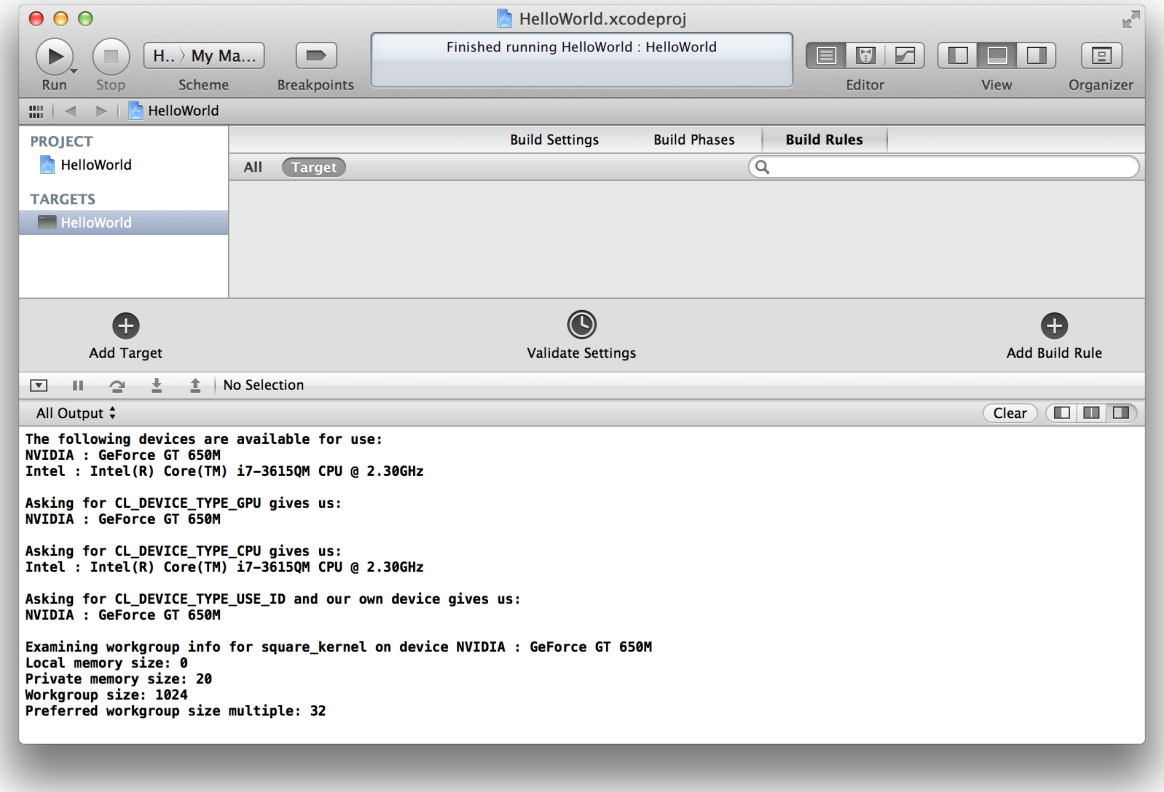
> **Note:** When you first include your Xcode-generated header that contains the kernel declaration, your kernel will not have been compiled yet. The `mykernel.cl.h` file will be flagged as missing. The `mykernel.cl.h` file is generated by Xcode when you build the application.

5. Link to the OpenCL framework in your project. See Adding an Existing Framework to a Project.
6. Build.

   Because you are compiling your host and your kernel code before you run them, you can see compiler errors and warnings before you run your application.
7. Run.

Results are shown in the Xcode output pane as shown in Figure 1-3 (page 14).

Figure 1-3 Results



See "Basic Programming Sample" (page 18) for a line-by-line description of the host and kernel code in the Hello World sample project.

Debugging

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

- Run your kernel on the CPU first. There is no memory protection on some GPUs. If an index goes out of bounds on the GPU, it is likely to take the whole system down. If an index goes out of bounds on the CPU, it may crash the program that's running, but it will not take the whole system down.
- 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. See GDB website.

- On the GPU, use explicit address range checks to look for out-of-range address accesses. (Remember: there is no memory protection on some GPUs.)

Compiling From the Command Line

You can also compile and run your OpenCL application outside of Xcode.

To compile from the command line, call `openclc`.

You can set the following compile line parameters:

- OpenCL Compiler Version

  The OpenCL C compiler version supported by the platform. To set this parameter from the command line, use:

  `-cl-std=CL1.1`

  The default is OpenCL C 1.1.

- OpenCL — Architectures

  A `StringList` specifying the list of the architectures for which the product will be built. This is usually set to a predefined build setting provided by the platform. To set this parameter from the command line, list up to three of the following, separated by whitespace:

  - `-triple i386-applecl-darwin`
  - `-triple x86_64-applecl-darwin`
  - `-triple gpu_32-applecl-darwin`

  For example, to compile for i386 and x86_64, your list would look like this:

  `-triple i386-applecl-darwin-triple x86_64-applecl-darwin`

  The default is that the product is built for all three architectures.

- Auto-vectorizer

  This switch enables or disables autovectorization of OpenCL kernels compiled for the CPU. To set this parameter from the command line, use:

  - `-cl-auto-vectorize-enable` to enable the autovectorizer.
  - `-cl-autovectorize-disable` to disable the autovectorizer.

  The default is `-cl-auto-vectorize-enable`.

- Double as single

  If enabled, double-precision floating-point expressions are treated as single-precision floating-point expressions. This option is available for GPUs only. To enable this parameter from the command line, use:

`-cl-double-as-single`

By default, this parameter is disabled.

- Flush denorms to zero

  This Boolean controls how single- and double-precision denormalized numbers are handled. If specified as a build option, the single-precision denormalized numbers may be flushed to zero; double-precision denormalized numbers may also be flushed to zero if the optional extension for double-precision is supported. This is intended to be a performance hint and the OpenCL compiler can choose not to flush denorms to zero if the device supports single-precision (or double-precision) denormalized numbers (that is, if the `CL_FP_DENORM` bit is not set in `CL_DEVICE_SINGLE_FP_CONFIG`). This flag only applies for scalar and vector single-precision floating-point variables and computations on these floating-point variables inside a program. It does not apply to reading from or writing to image objects.

  To enable this parameter from the command line, use:

  `-cl-denorms-are-zero`

  By default, this parameter is disabled.

- Optimization Level

  Specifies whether to optimize for fastest performance or smallest code size.

  To set this parameter from the command line, use:

  - `-Os` - optimize for smallest code size
  - `-O1` - optimize for fast performance
  - `-O2` - optimize for faster performance
  - `-O3` - optimize for fastest performance
  - `-O0` - do not optimize

  The default is fast `-O1` optimization.

- Relax IEEE Compliance

  If enabled, allows optimizations for floating-point arithmetic that may violate the IEEE 754 standard and the OpenCL numerical compliance requirements defined in section 7.4 for single-precision floating-point, section 9.3.9 for double-precision floating-point, and edge case behavior in section 7.5 of the OpenCL 1.1 specification. This is intended to be a performance optimization. This option causes the preprocessor macro `__FAST_RELAXED_MATH__` to be defined in the OpenCL program.

  To enable this parameter from the command line, use:

  `-cl-fast-relaxed-math`

  By default, this parameter is disabled.

- Use MAD

If enabled, allows expressions of the type `a * b + c` to be replaced by a Multiply-Add (MAD) instruction. If MAD is enabled, multistep instructions in the form `a * b + c` are performed in a single step, but the accuracy of the results may be compromised. For example, to optimize performance, some OpenCL devices implement MAD by truncating the result of the `a * b` operation before adding it to `c`.

To enable this parameter from the command line, use:

```
-cl-mad-enable
```

By default, this parameter is disabled.

- OpenCL—Preprocessing

  Space-separated list of preprocessor macros of the form `"foo"` or `"foo=bar"`. To specify preprocessor macros from the command line, prefix the string of macros with:

  ```
  -D
  ```

Basic Programming Sample

This chapter provides a tour through the code of a simple OpenCL application that performs calculations on a test data set. The code in Listing 2-2 (page 19) calls the kernel defined in Listing 2-1 (page 18). The kernel squares each value. Once the kernel completes its work, the host validates that every value was processed by the kernel.

Basic Kernel Code Sample

Listing 2-1 (page 18) is example kernel code. See to download the project. See "How the Kernel Interacts With Data in OS X OpenCL" (page 34) for more information about passing parameters to the kernel and retrieving information from the kernel.

Listing 2-1 Kernel code sample

```
// Simple OpenCL kernel that squares an input array.
// This code is stored in a file called mykernel.cl.
// You can name your kernel file as you would name any other
// file.  Use .cl as the file extension for all kernel
// source files.

// Kernel block.                                    //   1
kernel void square(                                 //   2
                   global float* input,             //   3
                   global float* output)
{
    size_t i = get_global_id(0);
    output[i] = input[i] * input[i];
}
```

Notes:

1. Wrap your kernel code into a **kernel block**:

```
kernel void kernelName(
                          global float* inputParameterName,
                          global float* [anotherInputParameter],
                             …,
                          global float* outputParameterName)
{
              ...
}
```

2. Kernels always return `void`.
3. Pass parameters to the kernel just as you would pass them to any other function.

Basic Host Code Sample

Listing 2-2 (page 19) is example code that would run on a host. It calls a kernel to square a set of values, then tests to ensure that the kernel processed all the data. Each numbered line in the listing is described in more detail following the listing. See to download the project.

Listing 2-2 Host code sample

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

// This include pulls in everything you need to develop with OpenCL in OS X.
#include <OpenCL/opencl.h>

// Include the header file generated by Xcode.  This header file contains the
//  kernel block declaration.                                              // 1
#include "mykernel.cl.h"

// Hard-coded number of values to test, for convenience.
#define NUM_VALUES 1024

// A utility function that checks that our kernel execution performs the
// requested work over the entire range of data.
```

```
static int validate(cl_float* input, cl_float* output) {
    int i;
    for (i = 0; i < NUM_VALUES; i++) {

        // The kernel was supposed to square each value.
        if ( output[i] != (input[i] * input[i]) ) {
            fprintf(stdout,
                    "Error: Element %d did not match expected output.\n", i);
            fprintf(stdout,
                    "       Saw %1.4f, expected %1.4f\n", output[i],
                            input[i] * input[i]);
            fflush(stdout);
            return 0;
        }
    }
    return 1;
}

int main (int argc, const char * argv[]) {
    int i;
    char name[128];

    // First, try to obtain a dispatch queue that can send work to the
    // GPU in our system.                                                // 2
    dispatch_queue_t queue =
               gcl_create_dispatch_queue(CL_DEVICE_TYPE_GPU, NULL);

    // In the event that our system does NOT have an OpenCL-compatible GPU,
    // we can use the OpenCL CPU compute device instead.
    if (queue == NULL) {
        queue = gcl_create_dispatch_queue(CL_DEVICE_TYPE_CPU, NULL);
    }

    // This is not required, but let's print out the name of the device
```

```
// we are using to do work.  We could use the same function,
// clGetDeviceInfo, to obtain all manner of information about the device.
cl_device_id gpu = gcl_get_device_id_with_dispatch_queue(queue);
clGetDeviceInfo(gpu, CL_DEVICE_NAME, 128, name, NULL);
fprintf(stdout, "Created a dispatch queue using the %s\n", name);

// Here we hardcode some test data.
// Normally, when this application is running for real, data would come from
// some REAL source, such as a camera, a sensor, or some compiled collection
// of statistics—it just depends on the problem you want to solve.
float* test_in = (float*)malloc(sizeof(cl_float) * NUM_VALUES);
for (i = 0; i < NUM_VALUES; i++) {
    test_in[i] = (cl_float)i;
}

// Once the computation using CL is done, will have to read the results
// back into our application's memory space.  Allocate some space for that.
float* test_out = (float*)malloc(sizeof(cl_float) * NUM_VALUES);

// The test kernel takes two parameters: an input float array and an
// output float array.  We can't send the application's buffers above, since
// our CL device operates on its own memory space.  Therefore, we allocate
// OpenCL memory for doing the work.  Notice that for the input array,
// we specify CL_MEM_COPY_HOST_PTR and provide the fake input data we
// created above.  This tells OpenCL to copy the data into its memory
// space before it executes the kernel.                                // 3
void* mem_in  = gcl_malloc(sizeof(cl_float) * NUM_VALUES, test_in,
                           CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR);

// The output array is not initalized; we're going to fill it up when
// we execute our kernel.                                              // 4
void* mem_out =
       gcl_malloc(sizeof(cl_float) * NUM_VALUES, NULL, CL_MEM_WRITE_ONLY);

// Dispatch the kernel block using one of the dispatch_ commands and the
```

```
// queue created earlier.                                          // 5

dispatch_sync(queue, ^{
    // Although we could pass NULL as the workgroup size, which would tell
    // OpenCL to pick the one it thinks is best, we can also ask
    // OpenCL for the suggested size, and pass it ourselves.
    size_t wgs;
    gcl_get_kernel_block_workgroup_info(square_kernel,
                                        CL_KERNEL_WORK_GROUP_SIZE,
                                        sizeof(wgs), &wgs, NULL);

    // The N-Dimensional Range over which we'd like to execute our
    // kernel.  In this case, we're operating on a 1D buffer, so
    // it makes sense that the range is 1D.
    cl_ndrange range = {                                            // 6
        1,                      // The number of dimensions to use.

        {0, 0, 0},              // The offset in each dimension.  To specify
                                // that all the data is processed, this is 0
                                // in the test case.                  // 7

        {NUM_VALUES, 0, 0},     // The global range—this is how many items
                                // IN TOTAL in each dimension you want to
                                // process.

        {wgs, 0, 0}             // The local size of each workgroup.  This
                                // determines the number of work items per
                                // workgroup.  It indirectly affects the
                                // number of workgroups, since the global
                                // size / local size yields the number of
                                // workgroups.  In this test case, there are
                                // NUM_VALUE / wgs workgroups.
    };
    // Calling the kernel is easy; simply call it like a function,
```

```
        // passing the ndrange as the first parameter, followed by the expected
        // kernel parameters.  Note that we case the 'void*' here to the
        // expected OpenCL types.  Remember, a 'float' in the
        // kernel, is a 'cl_float' from the application's perspective.   // 8

        square_kernel(&range,(cl_float*)mem_in, (cl_float*)mem_out);

        // Getting data out of the device's memory space is also easy;
        // use gcl_memcpy.  In this case, gcl_memcpy takes the output
        // computed by the kernel and copies it over to the
        // application's memory space.                                   // 9

        gcl_memcpy(test_out, mem_out, sizeof(cl_float) * NUM_VALUES);

    });


    // Check to see if the kernel did what it was supposed to:
    if ( validate(test_in, test_out)) {
        fprintf(stdout, "All values were properly squared.\n");
    }

    // Don't forget to free up the CL device's memory when you're done. // 10
    gcl_free(mem_in);
    gcl_free(mem_out);

    // And the same goes for system memory, as usual.
    free(test_in);
    free(test_out);

    // Finally, release your queue just as you would any GCD queue.    // 11
    dispatch_release(queue);
}
```

Notes:

1. Include the header file that contains the kernel block declaration. The name of the header file for a .cl file will be the name of the .cl file with .h appended to it. For example, if the .cl file is named `mykernel.cl`, the header file you must include will be `mykernel.cl.h`.
2. Call `gcl_create_dispatch_queue` to create the dispatch queue.
3. Create memory objects to hold input and output data and write input data to the input objects. Allocate an array on the OpenCL device from which to read kernel results back into host memory. Use `gcl_malloc` and make sure to use the OpenCL size of the datatype being returned. For example, write `gcl_malloc(sizeof(cl_float) * NUM_VALUES`. Because the CL device operates on its own memory space, allocate OpenCL memory for the input data upon which the kernel will work. Specify `CL_MEM_COPY_HOST_PTR` to tell OpenCL to copy over the input data from host memory into its memory space before it executes the kernel.
4. Allocate OpenCL memory in which the kernel will store its results.
5. Dispatch your kernel block using one of the dispatch commands and the queue you created above. In your dispatch call, you can specify workgroup parameters.
6. Describe the **data parallel range** (the ndrange) over which to execute the kernel in the `cl_ndrange` structure.

   OpenCL always executes kernels in a **data parallel** fashion—that is, instances of the same kernel (**work items**) execute on different portions of the total data set. Each work item is responsible for executing the kernel once and operating on its assigned portion of the data set.

   You use the `cl_ndrange` field to specify how the workgroups are to be organized. For more information, see "Specifying How To Divide Up A Dataset" (page 35).
7. Always pass an offset for each of three dimensions even though the workgroup may have fewer than three dimensions. See "Specifying How To Divide Up A Dataset" (page 35) for more information.
8. Call the kernel as you would call a function. Pass the `ndrange` as the first parameter, followed by the expected kernel parameters. Case the `void*` types to the expected OpenCL types. Remember, if you use `float` in your kernel, that's a `cl_float` from the application's perspective. The call to the kernel will look something like this:

```
kernelName(
       &ndrange,
       (cl_datatype*)inputArray,
       (cl_datatype*)outputArray);
```

9. Retrieve the data from the OpenCL device's memory space with `gcl_memcpy`. The output computed by the kernel is copied over to the host application's memory space.
10. Free OpenCL memory objects.
11. Call `dispatch_release`(...) on the dispatch queue you created with `gcl_create_dispatch_queue`(...) once you are done with it.

OpenCL On OS X Basics

Tools provided on OS X let you include OpenCL kernels as resources in Xcode projects, compile them along with the rest of your application, invoke kernels by passing them parameters just as if they were typical functions, and use Grand Central Dispatch (GCD) as the queuing API for executing OpenCL commands and kernels on the CPU and GPU.

If you need to create OpenCL programs at runtime, with source loaded as a string or from a file, or if you want API-level control over queueing, see *The OpenCL Specification* , available from the Khronos Group at http://www.khronos.org/registry/cl/.

Concepts

In the OpenCL specification, computational processors are called **devices**. An OpenCL device has one or more **compute units**. A **workgroup** executes on a single compute unit. A compute unit is composed of one or more processing elements and local memory.

A Mac computer always has a single CPU. It may not have any GPUs or it may have several. The CPU on a Mac has multiple compute units, which is why it is called a **multicore CPU**. The number of compute units in a CPU limits the number of workgroups that can execute concurrently.

CPUs usually contain between two and eight compute units, sometimes more. A graphics processing unit (GPU) typically contains many compute units-GPUs in current Mac systems feature tens of compute units, and future GPUs may contain hundreds. To OpenCL the number of compute units is irrelevant. OpenCL considers a CPU with eight compute units and a GPU with 100 compute units each to be a single **device**.

The OS X v10.7 implementation of the OpenCL API facilitates designing and coding **data parallel** programs to run on both CPU and GPU devices. In a data parallel program, the same program (or **kernel**) runs concurrently on different pieces of data and each invocation is called a **work item** and given a work item ID. The work item IDs are organized in up to three dimensions (called an **N-D range**).

A kernel is essentially a function written in the OpenCL language that enables it to be compiled for execution on any device that supports OpenCL. However, a kernel differs from a function called by another programming language because when you invoke “a” kernel, what actually happens is that many instances of the kernel execute, each of which processes a different chunk of data.

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 host application is run by OS X on the CPU. The device on which the host application executes is known as the **host device**. Before it runs the kernels, the host application typically:

1. Determines what compute devices are available, if necessary.
2. Selects compute devices appropriate for the application.
3. Creates dispatch queues for selected compute devices.
4. Allocates the memory objects needed by the kernels for execution. (This step may occur earlier in the process, as convenient.)

> **Note:** The host device (the CPU) can itself be an OpenCL device. Both the host application and kernels may run on the same CPU.

The host application can enqueue commands to read from and write to memory objects that are also accessible by kernels. See "Memory Objects in OS X OpenCL" (page 51). **Memory objects** are used to manipulate device memory. There are two types of memory objects used in OpenCL: **buffer objects** and **image objects**. Buffer objects can contain any type of data; image objects contain data organized into pixels in a given format.

Although kernels are enqueued for execution by host applications written in C, C++, or Objective-C, a kernel must be compiled separately 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.

OpenCL kernels can be:

- Compiled at compile time, then run when queued by the host application.

  or

- Compiled and then run at runtime when queued by the host application.

  or

- Run from a previously-built binary.

A **work item** is a parallel execution of a kernel on some data. It is analogous to a thread. Each kernel is executed upon hundreds of thousands of work items.

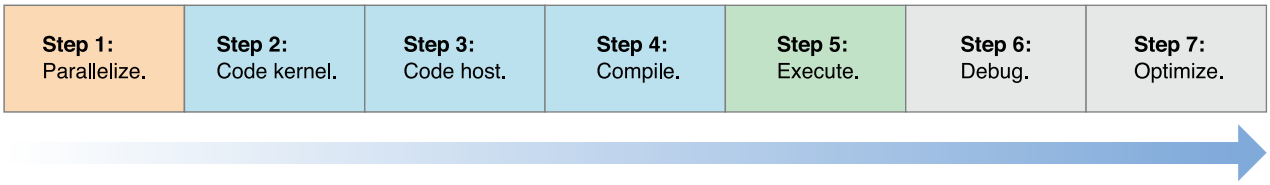
A **workgroup** is a set of work items that execute concurrently and share data. Each workgroup is executed on a compute unit.

Workgroup **dimensions** determine how kernels operate upon input in parallel. The application usually specifies the dimensions based on the size of the input. There are constraints; for example, there may be a maximum number of work items that can be launched for a certain kernel on a certain device.

Essential Development Tasks

As of OS X v10.7, the OpenCL development process includes these major steps:

Figure 3-1 OpenCL Development Process



1. Identify the tasks to be parallelized.

   Determining how to parallelize your program effectively is often the hardest part of developing an OpenCL program. See "Identifying Parallelizable Routines" (page 30).

2. Write your kernel functions.
   - See "How the Kernel Interacts With Data in OS X OpenCL" (page 34).
   - The "Basic Kernel Code Sample" (page 18) shows how you can store your kernel code in a file that can be compiled using Xcode.

3. Write the host code that will call the kernel(s).
   - See "Using Grand Central Dispatch With OpenCL" (page 39) for information about how the host can use GCD to enqueue the kernel.
   - See "Memory Objects in OS X OpenCL" (page 51) for information about how the host passes parameters to and retrieves results from the kernel.
   - See "Sharing Data Between OpenCL and OpenGL" (page 82) for information about how the OpenCL host can share data with OpenGL applications.
   - See "Controlling OpenCL / OpenGL Interoperation With GCD" (page 91) for information about how the OpenCL host can synchronize processing with OpenGL applications using GCD.
   - See "Using IOSurfaces With OpenCL" (page 98) for information about how the OpenCL host can use IOSurfaces to exchange data with a kernel.
   - The "Basic Host Code Sample" (page 19) shows how you can store your host code in a file that can be compiled with Xcode.

4. Compile using Xcode.

   See "Hello World!" (page 10).

5. Execute.

6. Debug (if necessary).

   See "Debugging" (page 14).

7. Improve performance (if necessary):

   - If your kernel(s) will be running on a CPU, see "Autovectorizer" (page 101) and, for suggestions about additional optimizations, see "Improving Performance On the CPU" (page 130).
   - If your kernel(s) will be running on a GPU, see "Tuning Performance On the GPU" (page 105).

Identifying Parallelizable Routines

This chapter provides a very basic introduction to how to design a program so that it can be implemented as an OpenCL kernel and called from an OpenCL host. For specifics about design patterns that may positively or negatively affect application performance, see "Tuning Performance On the GPU" (page 105) and "Improving Performance On the CPU" (page 130).

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 re-entrant 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 workgroup can share local data, they execute synchronously and so no 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. This application consists of two main tasks:

1. Compute the final grade for each student, assuming the final grade for each student is the average of all the grades that student received.
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: to calculate the class average, you must first calculate the final grade for each student.

Although you cannot perform task 1 and task 2 simultaneously, there is still an opportunity for parallelization here.

The pseudocode in Listing 4-1 (page 31) 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 final grade for each student independently and 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.

Listing 4-1 Pseudocode that computes the final grade for each student

```
// 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;
}
```

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. No students' grades affect another student's grades. This means that you can calculate each student's average in parallel.

To express this programmatically: first separate your task (calculating the final grade of a student) from your data (the list of students in the class). Listing 4-2 (page 31) shows how you can isolate the final-grading task.

Listing 4-2 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 the individual students in the class in parallel. Because OpenCL has native support for parallel computing, you can rewrite the task shown in Listing 4-2 (page 31) 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 to identify 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 might 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 to process data in parallel. 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.

How the Kernel Interacts With Data in OS X OpenCL

There are two parts of every OpenCL program. The part that runs on the device is called the **kernel**; the part that creates memory objects, then configures and calls the kernel is called the **host** and usually runs on the CPU. A kernel is essentially a function written in the OpenCL language that enables it to be compiled for execution on any device that supports OpenCL. The kernel is the only way the host can call a function that will run on a device. When the host invokes a kernel, many work items start running on the device. Each work item runs the code of the kernel, but works on a different part of the dataset. The kernel manages work items by accessing them using their IDs using functions such as `get_global_id(…)` and `get_local_id(…)`. Although kernels are enqueued for execution by host applications written in C, C++, or Objective-C, a kernel must be compiled separately to be customized for the device on which it is going to run.

Interacting with kernels is easier using tools provided by OS X than it is using standard OpenCL. As of OS X v10.7, you can include OpenCL kernels as resources in Xcode projects and compile them along with the rest of your application. Also as of OS X v10.7, the host can invoke kernels by passing them parameters just as if they were typical functions (see "Passing Data To a Kernel" (page 37)); it is no longer necessary to explicitly set kernel arguments using special OpenCL APIs.

Accessing Objects From a Kernel

In order for a device to actually process data, you have to make the data available to the work items that execute on the device.

To pass data from the host to a compute kernel:

- Prepare the input data.
- Specify how data is to be assigned to work items. See "Specifying How To Divide Up A Dataset" (page 35).
- Create buffer and image object(s) of the appropriate size. Move the input data from host memory using `gcl_malloc` or the various `gcl_ copy` functions (such as `gcl_memcpy`) to the device. See "Memory Objects in OS X OpenCL" (page 51) for more information.
- Invoke the kernel. Unlike in standard OpenCL, you don't have to explicitly set kernel arguments or enqueue the kernel to an OpenCL command queue; instead just queue the kernel as a block to a dispatch queue. See "Passing Data To a Kernel" (page 37).
- Retrieve results. See "Retrieving Results From a Kernel" (page 38).

Specifying How To Divide Up A Dataset

When you write a kernel in OpenCL, you are writing the code that each work item will execute-instructions on how to process one portion of your overall dataset. By launching many work items, each of which operates on just a small portion of the data, you end up processing the whole data set. The **ndrange structure** is used to specify how data is assigned to work items.

The n-dimensional range (`cl_ndrange`) structure you pass to the kernel consists of the following fields:

- `size_t work_dim`:

  The number of dimensions to use for the kernel launch: 1, 2, or 3.

  Some problems are easiest to break up into kernel-sized chunks if you treat them as one-dimensional. An example of this type of problem is computing the md5 hash for a list of 50 million words. You could write a kernel that computes the md5 hash for one word and launch 50 million instances of the kernel. In this case, the ndrange is a 1-D range: a single range (0 - 50 million) that has only one coordinate. You can think of it as as index. In your kernel, you can call `get_global_id(0)`, and it will give you that coordinate—a value from 0 to 49,999,999 that represents the index into your data that this instance of the kernel should process.

  If your data represents a flat image that is *x* pixels wide by *y* pixels high, then you have a two-dimensional data set with each data point represented by its coordinates on the x and y axes. Many image processing algorithms are best represented using a two-dimensional ndrange. Let's say you want to do something different to each pixel of a 2048 x 1024 image. You could write an OpenCL kernel that does something to a single pixel, and then launch it using an two-dimensional ndrange with a global work size of 2048 x 1024. In this case, your kernel is (obviously) not one-dimensional. You can call `get_global_id(0)` and `get_global_id(1)` to get the x,y coordinates of *this* instance in the entire ndrange. Because there is a 1-to-1 mapping between the ndrange and pixels, selecting the pixel to process is really easy; just call these two functions.

  If you are dealing with spatial data that involves the *(x, y, z)* position of nodes in three-dimensional space, you can use a three-dimensional ndrange.

  Another way to look at the dimensionality of your data is in terms of nested loops in traditional, non-parallel applications. 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 to cycle through all your data, your data is three-dimensional.

  > **Note:** Currently OpenCL, supports a maximum of three dimensions.

- `global_work_size`:

The `global_work_size` field specifies the size of each dimension. Effectively, this determines the number of total work items that will be launched. If you have a one-dimensional range and you want to process a million things, then the `global_work_size` field will be `{1000000, 0, 0}`. If you are processing a 2048 pixel by 1024 pixel image, you would set `work_dim = 2` and `global_work_size = { 2048, 1024, 0 }`.

The only constraint on the `global_work_size` is that the work size of each dimension must be a multiple of the `local_work_size` of that dimension.

- `global_work_offset`:

  The `global_work_offset` field specifies a per-dimension offset to add to the values returned by `get_global_id(…)`. Say, for example, you have a list of one million words and you want to compute the md5 hash of words 50,000 - 60,000. Because the data is one-dimensional, the ndrange would have a `work_dim` of `1`. Because there are 10,000 items to be processed, set `global_work_size = {10000, 0, 0}`. To "skip" to word 50,000 from the get-go, set the `global_work_offset = {50000, 0, 0}`. That way, the very first call to `get_global_id(0)` returns the 50,000th pixel, the second returns the 50,001st pixel, and so on.

- `local_work_size`:

  A **workgroup** is a collection of work items that execute on the same compute unit on the same OpenCL device. The way data is broken up into workgroups can affect the performance of an algorithm on certain hardware. When enqueuing a kernel to execute on a device, you can specify the size of the workgroup that you'd like OpenCL to use during execution. By providing OpenCL with a suggested workgroup size, you are telling it how you would like it to delegate the work items to the various computational units on the device. Work items within a workgroup have the unique ability to share local memory with one another, and synchronize with one another at programmer-specified barriers.

  The `local_work_size` gives you control over the workgroup size directly.

  > **Note:** The underlying hardware restricts the maximum workgroup size (`CL_DEVICE_MAX_WORK_GROUP_SIZE`) and the maximum value allowed for each dimension of `local_work_size` (`CL_DEVICE_MAX_WORK_ITEM_SIZES`). To obtain these values, call the OpenCL API's `clGetDeviceInfo` function. This API must be called inside a block submitted to a GCD queue created using `gcl_create_dispatch_queue`.
  >
  > In addition, the number of work items in each dimension in a single workgroup must divide evenly into the total number of work items in that dimension (`global_work_size_n mod local_work_size_n = 0`).

Passing Data To a Kernel

Xcode uses your kernel code to automatically generate the kernel function prototype in the kernel header file. To pass data to a kernel, pass the memory objects as parameters (just as you would pass parameters to any other function) when you call the kernel from your host code. OpenCL kernel arguments can be scoped with a local or global qualifier, designating the memory storage for these arguments. This means that, as of OS X v10.7, kernel parameters declared with the `local` or `__local` address qualifier are declared as `size_t` in the block declaration of the kernel.

For example, if a kernel has an argument declared with the `local` address qualifier:

```
kernel void foo(
                global float *a,
                local float *shared);  // This kernel parameter is of type
                                       // local float; will be size_t in the
                                       // kernel block
```

The compiler generates the following extern declaration of this kernel block:

```
extern void (^foo_kernel)(
                  const cl_ndrange *ndrange,
                  float *a,
                  size_t shared       // In the generated declaration,
                                      // local float is declared as size_t
             );
```

By associating your buffer objects with specific kernel arguments, you make it possible to process your data using a kernel function. For example, in "Example: Allocating, Using, and Releasing Buffer Objects" (page 76), 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.

Note: "Example: Allocating, Using, and Releasing Buffer Objects" (page 76) 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. For complete descriptions of these data types and operators, see *The OpenCL Specification* .

Retrieving Results From a Kernel

If the kernel will be returning results in a buffer, call a function such as `gcl_memcpy(...)` while inside a block on a given queue.

To make sure that the results are all accessible to the host before you continue, use `dispatch_sync` or wait using another synchronization method.

If the kernel will be returning results in a buffer, call the `dispatch_sync` function like this:

```
dispatch_sync(queue,
              ^{
                 gcl_memcpy(ptr_c,
                            device_c,
                            num_floats * sizeof(float));
               });
```

If the kernel will be returning results in an image, call the `dispatch_sync` function like this:

```
dispatch_sync(queue,^{
                        size_t origin = {0,0,0};
                        size_t region = {512, 512, 1};
                        gcl_copy_image_to_ptr(
                                              results_ptr,
                                              image,
                                              origin,
                                              region);
                     });
```

This will copy the bytes for 512 x 512 pixels from the image to the buffer specified by the `results_ptr` parameter.

Using Grand Central Dispatch With OpenCL

As of OS X v10.7, OpenCL developers can enqueue work coded as OpenCL kernels to Grand Central Dispatch (GCD) queues backed by OpenCL compute devices. You can use GCD with OpenCL to:

- Investigate the computational environment in which your OpenCL application is running. Specifically, you can learn which devices in the system would be best for performing particular OpenCL computations and operations:
  - You can find out about the computational power and technical characteristics of each OpenCL-capable device in the system. See "Discovering Available Compute Devices" (page 39).
  - GCD can suggest which OpenCL device(s) would be best for running a particular kernel.
  - You can obtain recommendations about how to configure kernels. For example, you can get the suggested optimal size of the workgroup for each kernel on any particular device. See "Notes" (page 40).
- Enqueue the kernel.
- Synchronize work between the host and OpenCL devices and synchronize work between devices.

  Your host can wait on completion of work in all queues (see "Using GCD To Synchronize A Host With OpenCL" (page 91)) or one queue can wait on completion of another queue (see "Synchronizing Multiple Queues" (page 97)).

Discovering Available Compute Devices

OpenCL kernels assume a Single Instruction, Multiple Data (**SIMD**) parallel model of computation. In SIMD, you have a large amount of data divided into chunks, and you want the kernel to perform the same computation on each chunk.

Some SIMD algorithms perform better on CPUs; others perform better on GPUs; some work better on certain kinds of GPUs rather than on others. Tools in OS X v10.7 and later facilitate discovery of the types of devices that are available to process data.

In order to learn about the environment in which your OpenCL kernels will be running, you have to retrieve the default global **context**. The context gives you information about the set of devices, the memory accessible to those devices, and one or more queues used to schedule execution of one or more kernels.

From the context, your application can discover the types of devices in the system and can obtain recommendations as to the optimal configuration for running a kernel. Once it knows the context, your application can call on GCD to create a queue for a particular type of device or to create a queue for a specific device.

To find out about available compute devices, an application:

1. Calls the `gcl_get_context` function to get the "global" OpenCL context that OS X creates for you.

   Note: Since this context is created by OpenCL, you should *not* retain/release it. (You *should* retain/release any contexts that you explicitly create.)

2. Calls the `clGetDeviceIds( ... )` function (an API in the OpenCL standard API), specifying the context you just obtained as the context parameter. This call returns a list of the IDs of the attached OpenCL devices. See *The OpenCL Specification* for details about this function.
3. You can choose to send different types of work to a device depending upon its characteristics and capabilities. Once you have the IDs of the devices in the context, call the `clGetDeviceInfo()` function for each of the devices to obtain information about the device. The sample code in Listing 6-1 (page 42) requests the vendor (the manufacturer) and the device name. You could also use the `clGetDeviceInfo()` function to request more technical information such as the number of compute cores, the cache line size and so on. See *The OpenCL Specification* for information about the types of information you can obtain.

   Notes:

   - A context is needed to share memory objects between devices. If you use OS X `gcl_` APIs, you can just retrieve and use the default global context; no context creation is needed.
   - If you are using APIs defined in the OpenCL specification, you do need to create your own contexts.
   - An OpenCL context is similar to an OpenGL **sharegroup**. A sharegroup is a set of tools that allow blocks of memory to be accessed by both a GPU and a CPU. See "Sharing Data Between OpenCL and OpenGL" (page 82).

Enqueueing A Kernel To A Dispatch Queue

Your application must use an OpenCL-compatible dispatch queue for its OpenCL work. You can create a queue for a particular device in the system or you can create a queue for a particular type of device. You can enqueue as many kernels on each queue as you choose. You can create as many different queues as you would like:

- To create a dispatch queue to run on any device so long as it's of a particular type, call the `gcl_create_dispatch_queue` function passing `CL_DEVICE_TYPE_CPU`, `CL_DEVICE_TYPE_GPU`, or `CL_DEVICE_TYPE_ACCELERATOR` as the first parameter.

  > **Note:** The dispatch queue you create must be attached to a particular device type. You cannot create an OpenCL-compatible dispatch queue for the default device type (`CL_DEVICE_TYPE_DEFAULT`).

  OS X OpenCL will create a dispatch queue that uses a GPU or CPU, depending upon the device type you specified. If more than one GPU is available, OS X OpenCL enqueues the kernel on the device of the type you specify that has the largest number of compute cores.

  > **Note:** If you've created your dispatch queue specifying `CL_DEVICE_TYPE_GPU`, you won't know which GPU is being used. To find out which device is actually attached to a given dispatch queue, call the `gcl_get_device_id_with_dispatch_queue` function.

- If you know exactly which OpenCL device ID you want to use because you've obtained it with the `clGetDeviceIds` function and found out about it using the `clGetDeviceInfo` function, call the `cl_create_dispatch_queue` function with `CL_DEVICE_TYPE_USE_ID` and pass the ID of the device you want to use.

Both of these methods of enqueing a kernel on a dispatch queue are illustrated in Listing 6-1 (page 42).

> **Note:** Always call the `dispatch_release`(...) function on the dispatch queue you created with the `gcl_create_dispatch_queue`(...) function once you are done with it. All of the example code contains this call.

Once you have created a queue, you can enqueue as many kernels onto that queue as necessary. Or, you can create additional queues with different characteristics.

For more information about GCD queues, see Concurrency Programming Guide: Dispatch Queues.

Determining the Characteristics Of A Kernel On A Device

To obtain information specific to a kernel/device pair, such as how much private and local memory the kernel will consume on a device, or the optimal workgroup size for execution, call the `gcl_get_kernel_block_workgroup_info` function. This information is useful when you are tuning performance for a kernel running on a particular device or debugging performance issues.

You can use the suggested workgroup size returned by the `gcl_get_kernel_block_workgroup_info` function for a particular kernel on a particular device as the `cl_ndrange.local_work_size`.

Note: You must call the `gcl_get_kernel_block_workgroup_info` function inside a block submitted to a GCD dispatch queue created using the `gcl_create_dispatch_queue` function.

Sample Code: Creating a Dispatch Queue

Listing 6-1 (page 42) demonstrates how to get the global OpenCL context, and how to ask that context about the devices it contains. It also shows how to create a dispatch queue by asking for a device type (CPU or GPU), and by specifying the queue's OpenCL device directly.

Listing 6-2 (page 46) shows how to obtain workgroup information -- useful for obtaining peak performance -- from the kernel block.

Listing 6-1 Creating a dispatch queue

```
#include <stdio.h>

// Include OpenCL/opencl.h to include everything you need for OpenCL
// development on OS X v10.7 or later.
#include <OpenCL/opencl.h>

// In this example, mykernel.cl.h is the header file that contains
// the kernel block declaration.  The name of this header file would
// be different if the name of the file containing the kernel source
// were different.
// This header file is generated by Xcode.
#include "mykernel.cl.h"
```

```
static void print_device_info(cl_device_id device) {
    char name[128];
    char vendor[128];

    clGetDeviceInfo(device, CL_DEVICE_NAME, 128, name, NULL);
    clGetDeviceInfo(device, CL_DEVICE_VENDOR, 128, vendor, NULL);
    fprintf(stdout, "%s : %s\n", vendor, name);
}

// Demonstrates how to get the global OpenCL context, and how to ask that
// context about the devices it contains.  It also shows how
// to create a dispatch queue by asking for a device type (CPU or GPU) and
// by specifying the queue's OpenCL device directly.

static void hello_world_sample1 ()
{
    int i;

    // Ask for the global OpenCL context:
    // Note: If you will not be enqueing to a specific device, you do not need
    // to retrieve the context.

    cl_context context = gcl_get_context();

    // Query this context to see what kinds of devices are available.

    size_t length;
    cl_device_id devices[8];
    clGetContextInfo(
         context, CL_CONTEXT_DEVICES, sizeof(devices), devices, &length);

    // Walk over these devices, printing out some basic information.  You could
    // query any of the information available about the device here.
```

```
fprintf(stdout, "The following devices are available for use:\n");
int num_devices = (int)(length / sizeof(cl_device_id));
for (i = 0; i < num_devices; i++) {
    print_device_info(devices[i]);
}

// To do any work, you need to create a dispatch queue associated
// with some OpenCL device.  You can either let the system give you
// a GPU—perhaps the only GPU—or the CPU device.  Or, you can
// create a dispatch queue with a cl_device_id you specify.  This
// device id comes from the OpenCL context, as above.  Below are three
// examples.

// 1. Ask for a GPU-based dispatch queue; notice that here we do not provide
// a device id.  Instead, we let the system tell us the most capable GPU.

dispatch_queue_t gpu_queue =
   gcl_create_dispatch_queue(CL_DEVICE_TYPE_GPU, NULL);

// Get the device from the queue, so we can ask OpenCL questions about it.
// Note that we check to make sure there WAS an OpenCL-capable GPU in the
// system by checking against a NULL return value.

if (gpu_queue != NULL) {

    cl_device_id gpu_device =
      gcl_get_device_id_with_dispatch_queue(gpu_queue);
    fprintf(stdout, "\nAsking for CL_DEVICE_TYPE_GPU gives us:\n");
    print_device_info(gpu_device);

} else {
    fprintf(stdout, "\nYour system does not contain an OpenCL-compatible "
            "GPU\n.");
}
```

```
    // 2. Try the same thing for CL_DEVICE_TYPE_CPU.  All Mac
    // systems have a CPU OpenCL device, so you don't have to
    // check for NULL, as you have to do in the case of a GPU.

    dispatch_queue_t cpu_queue =
        gcl_create_dispatch_queue(CL_DEVICE_TYPE_CPU, NULL);
    cl_device_id cpu_device = gcl_get_device_id_with_dispatch_queue(cpu_queue);
    fprintf(stdout, "\nAsking for CL_DEVICE_TYPE_CPU gives us:\n");
    print_device_info(cpu_device);

    // 3. Or perhaps you are in a situation where you want a specific device
    // from the list of devices you found on the context.
    // Notice the difference here:
    // Pass CL_DEVICE_TYPE_USE_ID and a device_id. This example just uses the
    // first device on the context from above, whatever that might be.

    dispatch_queue_t custom_queue =
        gcl_create_dispatch_queue(CL_DEVICE_TYPE_USE_ID, devices[0]);
    cl_device_id custom_device =
        gcl_get_device_id_with_dispatch_queue(custom_queue);
    fprintf(stdout,
       "\nAsking for CL_DEVICE_TYPE_USE_ID and our own device gives us:\n");
    print_device_info(custom_device);

    // Now you can use any of these 3 dispatch queues to run some kernels.
    …                                                 // Run your kernels here.

    // Use the GCD API to free your queues.

    dispatch_release(custom_queue);
    dispatch_release(cpu_queue);

    if (gpu_queue != NULL) dispatch_release(gpu_queue);
}
```

Listing 6-2 Obtaining workgroup information

```
// This listing shows how to obtain workgroup info —
// useful for obtaining peak performance—from the kernel block.

static void hello_world_sample2() {

    // Get a queue backed by a GPU for running our squaring kernel.
    dispatch_queue_t queue =
       gcl_create_dispatch_queue(CL_DEVICE_TYPE_GPU, NULL);

    // Did we get a GPU?  If not, fall back to the CPU device.
    if (queue == NULL) {
        gcl_create_dispatch_queue(CL_DEVICE_TYPE_GPU, NULL);
    }

    // In any case, print out the device you're using:

    fprintf(stdout, "\nExamining workgroup info for square_kernel on device ");
    print_device_info(gcl_get_device_id_with_dispatch_queue(queue));

    // Now find out what OpenCL thinks is the best workgroup size for
    // executing this kernel on this particular device.  Notice that this
    // method is executed in a block, on a dispatch queue you've created
    // with OpenCL.

    dispatch_sync(queue,
                  ^{
                      size_t wgs, preferred_wgs_multiple;
                      cl_ulong local_memsize, private_memsize;

                      // The next two calls give you information about how much
                      // memory, local and private, is used by the kernel on this
```

```
        // particular device.
        gcl_get_kernel_block_workgroup_info(square_kernel,
                  CL_KERNEL_LOCAL_MEM_SIZE,
                  sizeof(local_memsize),
                  &local_memsize, NULL);
       fprintf(stdout, "Local memory size: %lld\n", local_memsize);

        gcl_get_kernel_block_workgroup_info(square_kernel,
                  CL_KERNEL_PRIVATE_MEM_SIZE,
                  sizeof(private_memsize),
                  &private_memsize, NULL);
        fprintf(stdout,
            "Private memory size: %lld\n", private_memsize);

         // Ask OpenCL to suggest the optimal workgroup
         // size for this kernel on this device.
         gcl_get_kernel_block_workgroup_info(square_kernel,
                 CL_KERNEL_WORK_GROUP_SIZE,
                 sizeof(wgs), &wgs, NULL);
          fprintf(stdout, "Workgroup size: %ld\n", wgs);

       // Finally, you can ask OpenCL for a workgroup size multiple.
         // This is a performance hint.
         gcl_get_kernel_block_workgroup_info(square_kernel,
                 CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE,
                 sizeof(preferred_wgs_multiple),
                 &preferred_wgs_multiple, NULL);
        fprintf(stdout, "Preferred workgroup size multiple: %ld\n",
                 preferred_wgs_multiple);

        // You can now use these workgroup values to craft an
        // appropriate cl_ndrange structure for use in launching
        // your kernel.

});
```

```
    dispatch_release(queue);
}


int main(int argc, const char* argv[]) {
    hello_world_sample1();
    hello_world_sample2();
}
```

Sample Code: Obtaining the Kernel's Workgroup Size

In Listing 6-1 (page 42), the host calls the `gcl_get_kernel_block_workgroup_info` method in a block on a dispatch queue created with OpenCL to request the local memory size:

```
gcl_get_kernel_block_workgroup_info(
                              square_kernel,
                              CL_KERNEL_LOCAL_MEM_SIZE,
                              sizeof(local_memsize),
                              &local_memsize, NULL);
```

Then, in Listing 6-2 (page 46), the `gcl_get_kernel_block_workgroup_info` function returns what it considers to be the optimal workgroup size for this kernel on this device:

```
gcl_get_kernel_block_workgroup_info(
                              square_kernel,
                              CL_KERNEL_WORK_GROUP_SIZE,
                              sizeof(workgroup_size), &workgroup_size, NULL);
                              fprintf(stdout, "Workgroup size: %ld\n",
                                      workgroup_size);
```

Finally, the host calls the `gcl_get_kernel_block_workgroup_info` function to suggest a workgroup size multiple based on the capabilities of the underlying device:

```
gcl_get_kernel_block_workgroup_info(
                              square_kernel,
                              CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE,
                              sizeof(preferred_workgroup_size_multiple),
                              &preferred_workgroup_size_multiple, NULL);
```

You can use the returned workgroup values to craft an appropriate `cl_ndrange structure` to use in launching your kernel.

```
cl_ndrange range = {
        1,                  // The number of dimensions to use.

        {0, 0, 0},          // The offset in each dimension.  Want to process
                                // ALL of the data, so all three offsets are 0.
                                // Always pass an offset for each of the
                                // three dimensions even though the workgroup
                                // may have fewer than three dimensions.

        {NUM_VALUES, 0, 0},  // The global range—this is how many items
                                // total in each dimension you want to
                                // process.
                                // Always pass the global range for each of the
                                // three dimensions even though the workgroup
                                // may have fewer than three dimensions.

        {workgroup_size, 0, 0 } // The local size of each workgroup.  This
                                // determines the number of work items per
                                // workgroup.  It indirectly affects the
                                // number of workgroups, since the global
                                // size / local size yields the number of
                                // workgroups.  So in this test case,
                                // have NUM_VALUE/workgroup_size workgroups.
                                // Always pass the workgroup size for each of the
                                // three dimensions even though the workgroup
                                // may have fewer than three dimensions.
```

```
};
```

Memory Objects in OS X OpenCL

OpenCL uses memory in a different way than conventional programs do. This chapter introduces memory buffers and images as they are used by OpenCL applications. See "Creating and Managing Image Objects In OpenCL" (page 60) and "Creating and Managing Buffer Objects In OpenCL" (page 70) for specifics about managing image and buffer memory respectively.

See "Using IOSurfaces With OpenCL" (page 98) for specifics about using IOSurfaces with OpenCL.

Overview

Like all computational processes, processes that run on OpenCL devices consist of:

- Data

  The data accessed by OpenCL instructions exists as `cl_image` memory objects and memory buffers. Use **image objects** for representing two- or three-dimensional images (see "Creating and Managing Image Objects In OpenCL" (page 60)). Use **buffer objects** to contain other types of generic data (see "Creating and Managing Buffer Objects In OpenCL" (page 70)).

- Instructions in an OpenCL program that manipulate data.

For many devices (such as GPUs), the OpenCL memory is housed in a physically distinct piece of silicon. For other devices , the device memory is physically on the same chip. Regardless of its physical location, device memory can only be read and written by the OpenCL kernel code and host memory can only be read and written by the host.

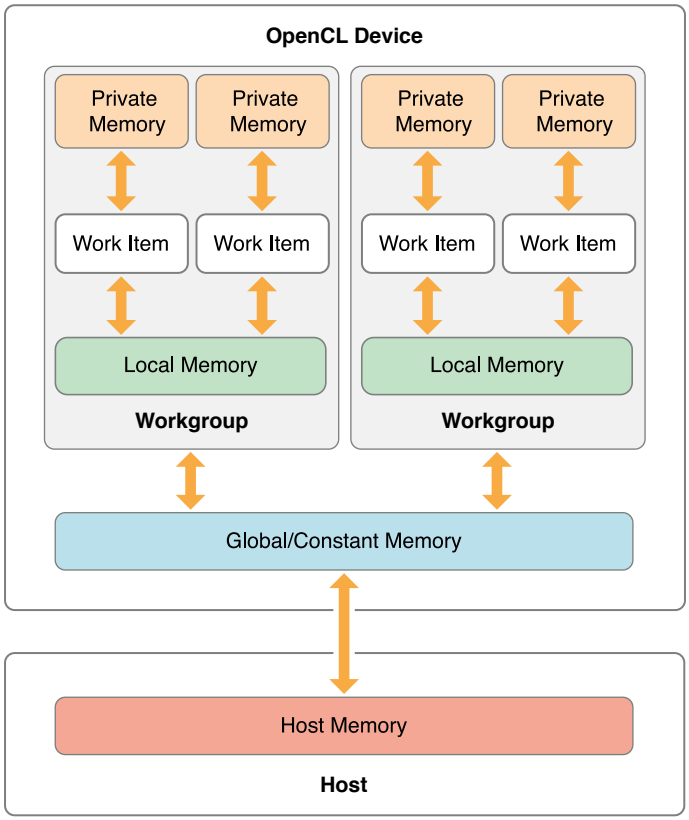
In other words, even if host and OpenCL memory are physically contiguous, host memory is distinct from OpenCL memory. Kernels can only access data in the memory of OpenCL devices. The host computer can read and write to device memory, but only to set it up and retrieve results. During computation, a device looks only in device memory, and the host stays out of its way.

When data has been passed to the kernel by the host, the data resides on the OpenCL device at the time of execution. A kernel cannot read or write to the host memory; it can only access data in the device's own separate memory area.

Memory Visibility

In a typical multi-device environment, memory is distributed between devices. No device can access all memory. Figure 7-1 (page 52) illustrates the memory spaces in an OpenCL system. Each memory space has different visibility to the host and the kernel.

Figure 7-1 Physical memory of an OpenCL system



- **Private memory** is memory that can only be accessed by one work item.

  The host has no ability to read or write private memory, nor can other work items.

- **Local memory** is memory that work items *within* a workgroup can share. Local memory is useful if more than one work item in a group needs to access a particular chunk of global memory. You can write your OpenCL program so that one work item loads data from global memory to local memory; the rest of the work items that need that piece of data can then access the local copy.

  It takes GPU devices much less time to access local memory than to access global memory. The host has no ability to read or write local memory.

- **Global memory** is the (relatively) massive chunk of device memory that all work items can "see". Any work item can read/write to a buffer declared to be in global memory.

  The host can also read and write to global memory.

- **Constant memory** is a specialized section of global memory used for data that you know will not change throughout the execution of your kernel. It is usually more limited in size than other types of global memory, but is faster to access on many devices than other global memory.

  The host can read and write to constant memory.

For example, an OpenCL kernel executes on a separate memory space from the host that calls it. In order for the kernel to access the data it is to process, the data must be moved into the device's memory. However, transferring data between memory areas to allow different devices to work can result in considerable overhead. To optimize performance, minimize the amount of data transferred.

The host specifies the memory space for a given buffer when it declares each kernel argument.

Memory Consistency

Changes a work item has made to global or local memory may not immediately become visible to other work items within a workgroup. A system's **consistency** tells you when changes a work item has made to global or local memory become visible to other work items within a workgroup.

OpenCL uses what is called a **relaxed memory consistency model**, which means that:

- Work items can access data within their own private, local, constant, and global memory spaces.
- Work items can share local memory during the execution of a workgroup. However, memory is only guaranteed to be consistent after defined synchronization points.

  If a work item needs to read something that another work item has written, call the **barrier** built-in function in your OpenCL kernel code at the point where you want the memory to be consistent:

```
barrier(CLK_LOCAL_MEM_FENCE);
barrier(CLK_GLOBAL_MEM_FENCE);
barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
```

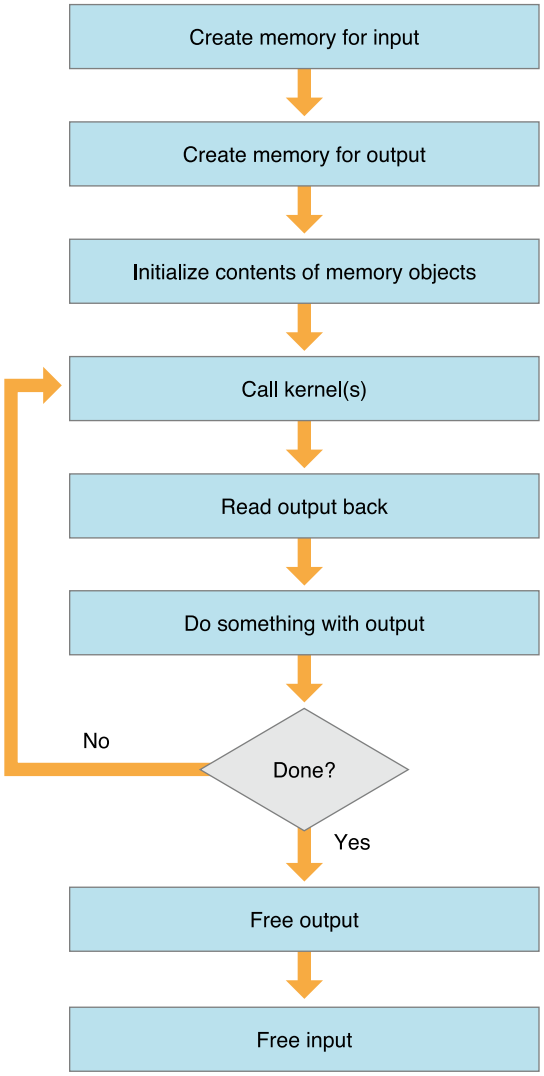
  The barrier will stop any work item at the point where it calls the barrier until all other work items have "caught up". That's why it works: every work item in the workgroup has written its memory by the time it reaches the barrier, so you can safely read anything any work item in the group has written.

  > **Note:** You can create a barrier that applies to local memory or to global memory, but consistency only applies to work items within a workgroup. *There is no such thing as a global memory barrier that can make all threads in an execution wait.* OpenCL only guarantees memory consistency at a barrier within a workgroup.

Memory Management Workflow

Figure 7-2 (page 54) illustrates the basic memory management workflow in OpenCL.

Figure 7-2 Memory workflow



The basic memory workflow with OpenCL is:

1. The host creates memory objects for use by OpenCL.

   The host requests that memory be set aside for it on the device. The host can ask for as many memory objects as it wants and gets them as long as there is memory available. You can create separate memory objects for input and output, but that is not a requirement. A single buffer object could be used for both input and output, provided this makes sense for your algorithm. Images, however, are more restricted. An image can be used for either input or output in a kernel, but not both at once.

2. The host initializes the contents of the memory objects.

   The host can pass data to the device to be stored in its memory objects. This is data that can be processed by the kernel. The host can also instruct the device to leave some memory objects uninitialized so that when the kernel runs on the device, it can fill these memory objects with output.

   The host instructs the device to execute the kernel, passing it the memory objects it has created on the OpenCL device as arguments. The host does not wait while the kernel works, but continues doing other work.

3. The kernel runs on the device, processing data, producing output.

4. The host reads results from the memory objects.

   When the host detects that the kernel has completed its tasks, it copies the results from kernel memory into memory the host can access.

5. The host destroys the memory objects.

   Once all kernels that need the memory objects have been run, the host instructs the device to free up the memory it had set aside for the kernel to use. For example, in a scenario where one buffer is used for input to many kernels, each of which runs many times, the host code frees the memory when every kernel execution that needed that memory object has been enqueued. You can specify a finalizer for objects. The finalizer will be called by the garbage collector when it destroys the object. See "Setting the finalizer" (page 55).

> **Note:** Since OpenCL memory objects are constructed in C, OS X ARC will not automatically release them. Your host code must explicitly destroy the memory objects the kernel uses.

Setting the finalizer

You can specify a finalizer, a function member of a reference class that is called automatically by the garbage collector when destroying an object. To specify the finalizer which OpenCL will invoke when the memory specified by the `object` parameter is to be deallocated, call:

```
void gcl_set_finalizer(void *object,
                       void (*gcl_pfn_finalizer)(void *object, void *user_data),
                       void *user_data);
```

Parameter	Description
`object`	The buffer or image memory object to which the finalizer is to be attached.

Parameter	Description
`(*gcl_pfn_finalizer)(void *object, void *user_data)`	Callback function that will be executed by the garbage collector when it is destroying the specified `object`. The parameters passed to the callback are: • The `object` parameter is a pointer to the buffer or image memory object to which you wish to attach the finalizer. • The `user_data` parameter is a pointer to the user data that will be passed to the finalizer callback.
`user_data`	The `user_data` parameter is a pointer to the user data that will be passed to the finalizer callback.

Listing 7-1 (page 56) is an example of how you would use the `gcl_set_finalizer` function.

Listing 7-1 Using gcl_set_finalizer

```
// This is the finalizer function you want OpenCL to call when the object is freed.
void my_finalizer(void* memObj,         // This is a pointer to the object whose
                                        // destruction triggered calling of this
                                        // finalizer.
                  void* importantData   // Pointer to data the finalizer will use.
                   )
{
// do something with memObj
// do something with importantData
}
...
void main() {
// Do stuff ...


void* my_mem_object = gcl_malloc(128, NULL, 0);      // GCL memory object
struct foo *special_data =
           (struct foo*)malloc(sizeof(struct foo)); // Some program-specific data

// Specify your finalizer callback.
```

```
gcl_set_finalizer(my_mem_object,  // Call the finalizer function when
                                  // my_mem_object is about to be freed.
                my_finalizer,     // In particular, call the my_finalizer function
                                  // when my_my_object is about to be freed.
                  special_data    // Pass this special_data to the my_finalizer
                                  // function before my_mem_object is freed.
                  );

// Do more stuff ...
// ...
// and then, at some point, free the my_mem_object.
gcl_free(my_mem_object);     // Before the my_mem_object object is actually freed,
                             // OpenCL will call my_finalizer, and pass it
                             // my_mem_object and special_data.
}
```

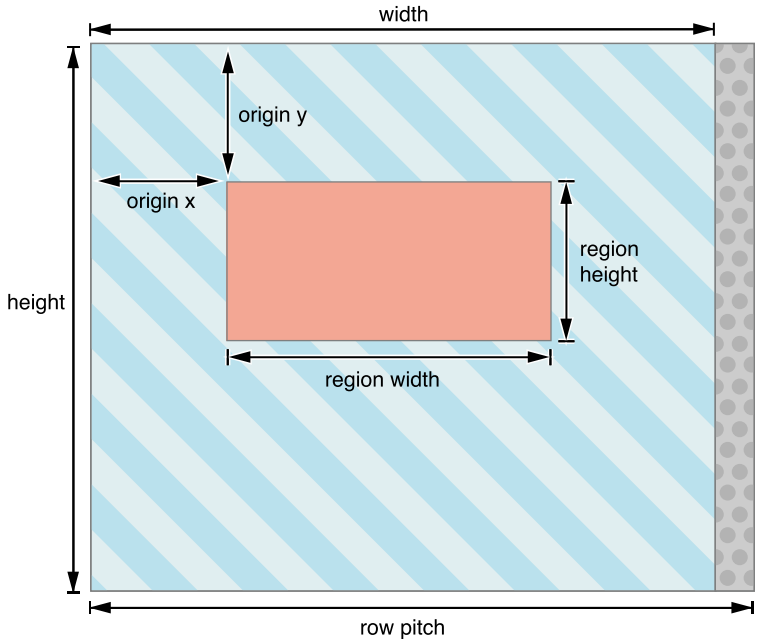
Note: This function provides functionality similar to that of the OpenCL standard `clSetMemObjectDestructorCallback` function.

Parameters That Describe Images and Buffers in OS X OpenCL

Several functions that manipulate images and buffers in the OpenCL API accept origin, region, and row pitch parameters.

In Figure 7-3 (page 58), the memory that contains the image is represented by the striped rectangle bounded by the width and the height, and includes the solid-colored region in its center. The **region** (represented by the solid pink rectangle bounded by the **region width** and **region height**), specifies the shape and size of the part of the image you want to manipulate. In other words, the region is just a certain amount of memory which lives at a given **offset** (or **origin**) into the image. The **origin** is the (x,y,z) position of the region within the image.

Figure 7-3 Pitch versus region



The **row pitch** is the width of the receiving area. The **row pitch** is the length in bytes in one row of the image. Usually, the row pitch is equal to the width in pixels of the image times the size of each pixel, in bytes. But sometimes, for performance reasons, it helps to use a bigger memory region to store the image than is strictly necessary. This is illustrated in Figure 7-3 (page 58), where the row pitch is a bit larger than the image width. For example, if the image were 1000 pixels wide, and each pixel was 4 bytes, your row_pitch would be 4000 bytes. But it might be more efficient for the hardware to process 4096 bytes per row, in which case you could specify the `row_pitch` parameter as `4096`.

> **Note:** The row pitch must be greater than or equal to the element size in bytes times the width. If you specify `row_pitch` as `0`, OpenCL calculates the appropriate row pitch to be the size of each element in bytes multiplied by width.

The **slice_pitch** parameter used in some OpenCL multidimensional buffer manipulation functions such as `gcl_memcpy_rect` (see "Creating and Managing Buffer Objects In OpenCL" (page 70)) is the size in bytes of a 2D slice that forms one layer of a 3D image. In other words, the slice pitch is the number of bytes in the row pitch multiplied by the number of bytes in the height of the image. Set the `slice_pitch` parameter to `0` if image is two-dimensional.

The slice pitch value must be greater than or equal to the row pitch multiplied by the height. If you specify the `slice_pitch` parameter as `0`, OpenCL calculates the appropriate slice pitch to be the row pitch multiplied by the height.

Creating and Managing Image Objects In OpenCL

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 for OS X's native support for images useful.

This chapter illustrates how to take image data residing in host memory and place it into image objects that a kernel can access. It also provides an overview of how to go about processing this image data. See "Parameters That Describe Images and Buffers in OS X OpenCL" (page 57) for conceptual descriptions of the kinds of parameters typically passed to these functions.

Creating and Using Images in OpenCL

To create image objects, use `gcl_create_image`. This function can be used to create two-dimensional image and three-dimensional image objects. To specify a two-dimensional image, set the `image_depth` parameter to `0`. To create a three-dimensional image object, specify the `image_depth` in pixels. If you pass an `IOSurfaceRef` as the `io_surface` parameter, the image will be created using the IOSurface you pass. Otherwise, set the `io_surface` parameter to `NULL`.

```
cl_image gcl_create_image(
                            const cl_image_format *image_format,
                            size_t image_width,
                            size_t image_height,
                            size_t image_depth,
                            IOSurfaceRef io_surface
);
```

Parameter	Description
`image_format`	An OpenCL image format descriptor.
`image_width`	The image width in pixels.

Parameter	Description
`image_height`	The image height in pixels.
`image_depth`	The image depth in pixels.
`io_surface`	If you pass an `IOSurfaceRef` as the `io_surface` parameter, the image will be created using the IOSurface you pass. Otherwise, set this parameter to `NULL`.

Reading, Writing, and Copying Image Objects

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

- To copy data between two images or to copy data from one portion of an image to another portion (in the same image), call:

```
void gcl_copy_image(
                cl_image dst_image,
                cl_image src_image,
                const size_t dst_origin[3],
                const size_t src_origin[3],
                const size_t region[3]);
```

  The copy starts from `src_origin` (an x, y, z value) and starts writing at `dst_origin` (also x, y, z). It copies the two- or three-dimensional rectangular amount specified by `region`. Because we are copying between images, all parameters are specified in pixels.

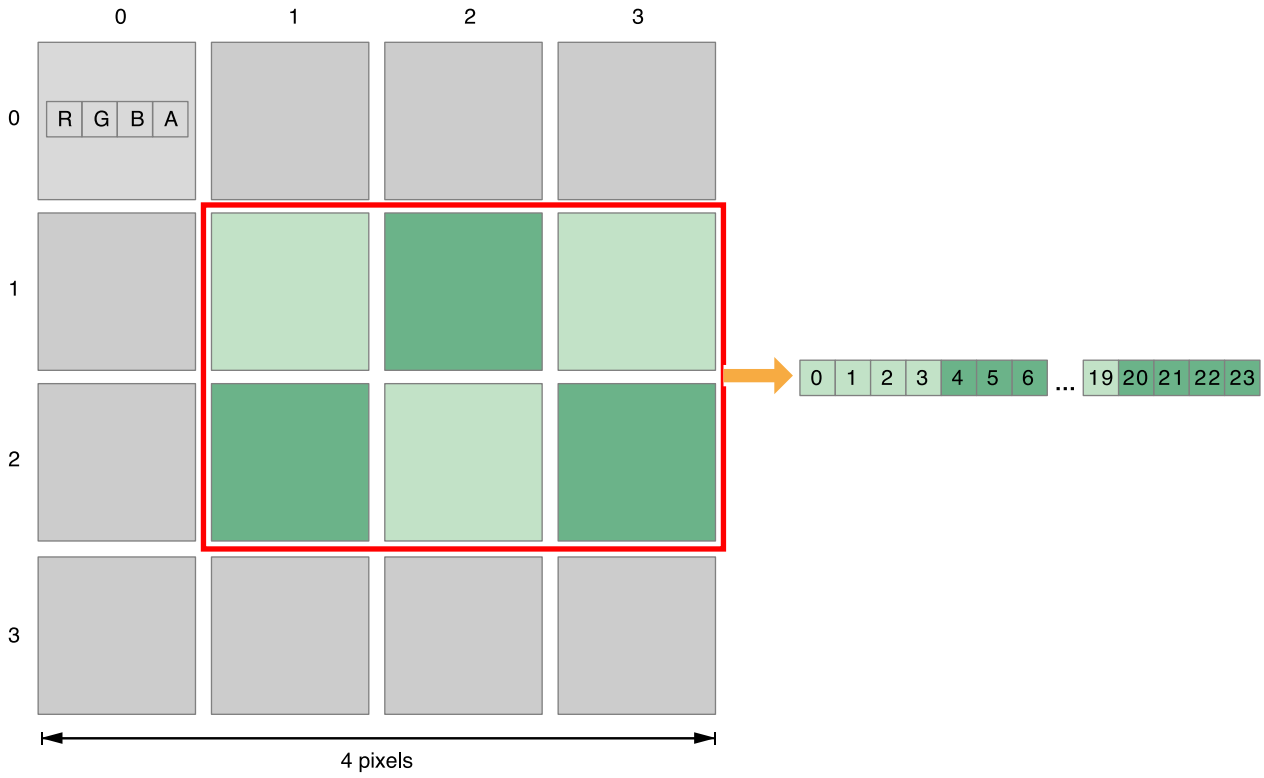
- To copy data from an image to a buffer, call:

```
void gcl_copy_image_to_ptr(
                 void *buffer_ptr,
                 cl_image src_image,
                 const size_t src_origin[3],
                 const size_t region[3]);
```

The `buffer_ptr` parameter points to the destination buffer to which pixels will be copied. The `src_image` is the image from which pixels are to be copied. The `src_origin` is the first pixel to be copied. The `region` is the region to which pixels are to be copied. Because we are copying from an image, the origin and region are specified in pixels.

For example, say we have a 4 pixel x 4 pixel image. Each pixel requires 4 bytes, one byte for each red, green, blue, and alpha channel. In this example, we want to take a portion of this image data—say a region 3 pixels wide and 2 pixels high—starting at pixel 1,1 of the image and copy this portion of the image to a buffer. In Figure 8-1 (page 62), we would want to copy all the light and dark green pixels, but not the gray pixels.

Figure 8-1 Copying a portion of an image to a buffer



Because our image requires four bytes per pixel and we want to copy out 3 x 2 = 6 pixels, we require a buffer of (at least) 6 x 4 = 24 bytes to accommodate the copy.

The call would look like this:

```
const size_t origin[3] = { 1, 1, 0 };
const size_t region[3] = { 3, 2, 1};
gcl_copy_image_to_ptr(our_buffer, the_image, origin, region);
```

Notes: The parameters to this function are the same as those of the standard OpenCL `clEnqueueCopyImageToBuffer` function. Specifically:

- The last element of origin, `origin[2]` is set to `0`. That's required since we're dealing with a two-dimensional image. (If we were dealing with a three-dimensional image, this number would be the index into the third dimension.)
- When copying a two-dimensional image, `region[2]` is set to 1. This results in a buffer that contains a slice of pixels 3 wide, 2 high, and 1 deep. For a two-dimensional image, `region[2]` is set to 1 because you are copying one "slice" of pixels width x height in size. If you set it to 2, then the number of pixels you get doubles, and you get one more slice "down" in the "stack" of pixels.

- To copy data from a buffer to an image, call:

```
void gcl_copy_ptr_to_image(
                cl_image dst_image,
                void *src_buffer_ptr,
                const size_t dst_origin[3],
                const size_t buffer_region[3]);
```

The parameters you pass to the `gcl_copy_ptr_to_image` function are similar to those passed to the `gcl_copy_image_to_ptr` function, except that the destination is the image and the pixels are being copied from a buffer.

Accessing Image Objects From a Kernel

The `gcl_copy*` 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.

See "How the Kernel Interacts With Data in OS X OpenCL" (page 34) for more information.

Mapping Image Objects

To map a region in an image into the host address space, call:

```
void *gcl_map_image(cl_image image,
                    cl_map_flags map_flags,
                    const size_t origin[3],
                    const size_t region[3]);
```

Returns a pointer to the region it has mapped.

Parameter	Description
`image`	The image to be mapped.
`map_flags`	Bitfield specifying `CL_MAP_READ` and/or `CL_MAP_WRITE`, depending on how you intend to use the returned pointer.
`origin`	The (x, y, z) position in pixels in the image at which to start the mapping.
`region`	The region (in pixels) to read.

> **Note:** Since the values are specified in pixels rather than bytes, the actual number of bytes mapped will depend upon the image format.

This function provides functionality similar to that of the OpenCL standard `clEnqueueMapImage` function.

Unmapping Image Objects

To unmap memory mapped by the `gcl_map_ptr` or `gcl_map_image` functions, call:

```
void gcl_unmap(void *ptr);
```

Parameter	Description
`ptr`	Pointer to the device memory, or image, to unmap.

Note: Call this function on the device memory pointer or the image memory object, NOT on the value returned by `gcl_map_ptr` or `gcl_map_image` functions.

Retaining and Releasing Image Objects

To avoid memory leaks, image objects should be freed when they are no longer needed.

- `void gcl_retain_image(cl_image image)`
- `void gcl_release_image(cl_image image)`

Example

In the following example, the host creates one image for input and one image for output, calls the kernel to swap the red and green pixels, then checks the results.

Listing 8-1 Sample host function creates images then calls kernel function

```
#include <stdio.h>
#include <stdlib.h>
#include <OpenCL/opencl.h>

// Include the automatically-generated header which provides the kernel block
// declaration.
#include "kernels.cl.h"

#define COUNT 2048

static void display_device(cl_device_id device)
{
    char name_buf[128];
    char vendor_buf[128];

    clGetDeviceInfo(device, CL_DEVICE_NAME, sizeof(char)*128, name_buf, NULL);
   clGetDeviceInfo(device, CL_DEVICE_VENDOR, sizeof(char)*128, vendor_buf, NULL);
```

```
    fprintf(stdout, "Using OpenCL device: %s %s\n", vendor_buf, name_buf);
}

static void image_test(const dispatch_queue_t dq)
{
    // This example uses a dispatch semaphore to achieve synchronization
    // between the host application and the work done for us by the OpenCL device.
    dispatch_semaphore_t dsema = dispatch_semaphore_create(0);

    // This example creates a "fake" RGBA, 8-bit-per channel image, solid red.
    // In a real program, you would use some real raster data.
    // Most OpenCL devices support a wide variety of image formats.

    unsigned int i;
    size_t height = 2048, width = 2048;

    unsigned int *pixels =
                (unsigned int*)malloc( sizeof(unsigned int) * width * height );

    for (i = 0; i < width*height; i++)
        pixels[i] = 0xFF0000FF; // 0xAABBGGRR: 8bits per channel, all red.

    // This image data is on the host side.
    // You need to create two OpenCL images in order to perform some
    // manipulations: one for the input and one for the ouput.

    // This describes the format of the image data.
    cl_image_format format;
    format.image_channel_order = CL_RGBA;
    format.image_channel_data_type = CL_UNSIGNED_INT8;

    cl_mem input_image = gcl_create_image(&format, width, height, 1, NULL);
    cl_mem output_image = gcl_create_image(&format, width, height, 1, NULL);
```

```
dispatch_async(dq, ^{

  // This kernel is written such that each work item processes one pixel.
  // Thus, it executes over a two-dimensional range, with the width and
  // height of the image determining the dimensions
  // of execution.

    cl_ndrange range = {
        2,                  // Using a two-dimensional execution.
        {0},                // Start at the beginning of the range.
        {width, height},    // Execute width * height work items.
        {0}                 // And let OpenCL decide how to divide
                            // the work items into work-groups.
    };

    // Copy the host-side, initial pixel data to the image memory object on
    // the OpenCL device.  Here, we copy the whole image, but you could use
    // the origin and region parameters to specify an offset and sub-region
    // of the image, if you'd like.
    const size_t origin[3] = { 0, 0, 0 };
    const size_t region[3] = { width, height, 1 };
    gcl_copy_ptr_to_image(input_image, pixels, origin, region);

    // Do it!
    red_to_green_kernel(&range, input_image, output_image);

    // Read back the results; then reuse the host-side buffer we
    // started with.
    gcl_copy_image_to_ptr(pixels, output_image, origin, region);

    // Let the host know we're done.
    dispatch_semaphore_signal(dsema);
});

// Do other work, if you'd like...
```

```
    // ... but eventually, you will want to wait for OpenCL to finish up.
    dispatch_semaphore_wait(dsema, DISPATCH_TIME_FOREVER);

    // We expect '0xFF00FF00' for each pixel.
    // Solid green, all the way.
    int results_ok = 1;
    for (i = 0; i < width*height; i++) {
        if (pixels[i] != 0xFF00FF00) {
            fprintf(stdout,
                "Oh dear. Pixel %d was not correct.
                 Expected 0xFF00FF00, saw %x\n",
                i, pixels[i]);
            results_ok = 0;
            break;
        }
    }

    if (results_ok)
        fprintf(stdout, "Image results OK!\n");

    // Clean up device-size allocations.
    // Note that we use the "standard" OpenCL API here.
    clReleaseMemObject(input_image);
    clReleaseMemObject(output_image);

    // Clean up host-side allocations.
    free(pixels);
}

int main (int argc, const char * argv[])
{
    // Grab a CPU-based dispatch queue.
    dispatch_queue_t dq = gcl_create_dispatch_queue(CL_DEVICE_TYPE_CPU, NULL);
```

```
    if (!dq)
    {
        fprintf(stdout, "Unable to create a CPU-based dispatch queue.\n");
        exit(1);
    }

    // Display the OpenCL device associated with this dispatch queue.
    display_device(gcl_get_device_id_with_dispatch_queue(dq));

    image_test(dq);

    fprintf(stdout, "\nDone.\n\n");

    dispatch_release(dq);
}
```

Listing 8-2 Sample kernel swaps the red and green channels

```
// A simple kernel that swaps the red and green channels.

const sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE | CLK_FILTER_NEAREST;

kernel void red_to_green(read_only image2d_t input, write_only image2d_t output)
{
    size_t x = get_global_id(0);
    size_t y = get_global_id(1);

    uint4 tap = read_imageui(input, sampler, (int2)(x,y));
    write_imageui(output, (int2)(x,y), tap.yxzw);
}
```

Creating and Managing Buffer Objects In OpenCL

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 then operate on that data using the same language features that you are accustomed to in C.

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 Memory For A Buffer Object In Device Memory

To create a buffer object in device memory call:

```
void * gcl_malloc(size_t bytes, void *host_ptr, cl_malloc_flags flags)
```

The `gcl_malloc` function is very similar to the C language `malloc` function. The `gcl_malloc` function returns an opaque pointer to a device memory buffer.

> **Note:** This return value cannot be used to access the memory directly. You can pass this handle to a kernel as a parameter that expects a memory object, or you can pass it to other `gcl_` functions that expect a device memory pointer (such as `gcl_memcpy`).

If insufficient memory exists on the device to satisfy the request, this function returns `NULL`.

Parameter	Description
`bytes`	The size in bytes of the allocation request.
`host_ptr`	Pointer to a host-side buffer which will be used to initialize the memory allocation if `CL_MEM_COPY_HOST_PTR` is present in the `flags` parameter.
`flags`	Bitfield which consists of 0 or more memory flags discussed in Section 5.2.1 of the OpenCL 1.1 Specification. If you specify some combination of `flags` that requires a `host_ptr`, pass a non-`NULL``host_ptr` parameter. Otherwise, just pass `NULL`.

Converting a Handle To a cl_mem Object For Use With a Standard OpenCL API

If you are going to be using a standard OpenCL API call, you'll need a `cl_mem` object. To create a `cl_mem` object, call the `gcl_malloc` function to allocate the memory, then call the `gcl_create_buffer_from_ptr` function to convert the handle `gcl_malloc` returns for use with the standard OpenCL API. Call:

```
cl_mem gcl_create_buffer_from_ptr(void *ptr)
```

This function is required *only* in cases where you will be using the standard OpenCL API alongside the `gcl_` entry points. It returns a `cl_mem` object suitable for use with the standard OpenCL API.

It accepts a `ptr` parameter—a pointer created by the `gcl_malloc` function and returns a corresponding `cl_mem` object suitable for use with the standard OpenCL API.

> **Note:** Be sure to call the `clReleaseMemObject` function on `cl_mem` objects created using `gcl_create_buffer_from_ptr` *before* you free the pointer using the `gcl_free` function.

The code will look something like this:

```
void* device_ptr = gcl_malloc(…);
cl_mem device_mem = gcl_create_buffer_from_ptr(device_ptr);

// Do stuff with device_ptr and device_mem.

clReleaseMemObject(device_mem);
gcl_free(device_ptr);
```

Parameter	Description
`ptr`	A pointer returned by the `gcl_malloc` function.

Accessing Device Global Memory

To access the device global memory represented by a given pointer that was created by calling the `gcl_malloc` function, call:

```
void *gcl_map_ptr(void *ptr, cl_map_flags map_flags, size_t cb);
```

The `gcl_map_ptr` function provides functionality similar to that of the OpenCL standard `clEnqueueMapBuffer` function. It returns a host-accessible pointer to the memory represented by a device memory pointer that is suitable for reading and writing. You can use this as an alternative to the various `gcl_` copy functions to access the device global memory represented by a given pointer that was created by a call to the `gcl_malloc` function.

Parameter	Description
`ptr`	Pointer into the device memory which is to be mapped. This pointer is created by the `gcl_malloc` function.
`map_flags`	Bitfield specifying `CL_MAP_READ` and/or `CL_MAP_WRITE`, depending on how you intend to use the returned pointer.
`cb`	Number of bytes of the buffer to map. (**cb** stands for 'count in bytes').

Copying Buffer Objects

When you allocate device memory using the `gcl_malloc` function, you need not create it on a device-specific dispatch queue. But when the time comes to actually *use* the memory, either for a kernel execution or a copy of some sort, OpenCL needs to know which device you intend to use.

Copying Data From Device or Host Memory To Host or Device Memory

To copy data from either device or host memory to either host or device memory, call:

```
void gcl_memcpy(void *dst, const void *src, size_t size);
```

Parameter	Description
`dst`	A pointer that points to the memory into which the bytes will be copied. It can either be a regular host pointer, or it can be a device memory pointer created by the `gcl_malloc` function.
`src`	A pointer to the memory that is to be copied. As with the `dst` parameter, this can be a pointer to either host or device memory.
`size`	The amount of memory in bytes to copy from `src` to `dst`.

Important: Make sure that both buffers are of sufficient size to accommodate the copy.

The call to this function must occur within a block on a particular dispatch queue.

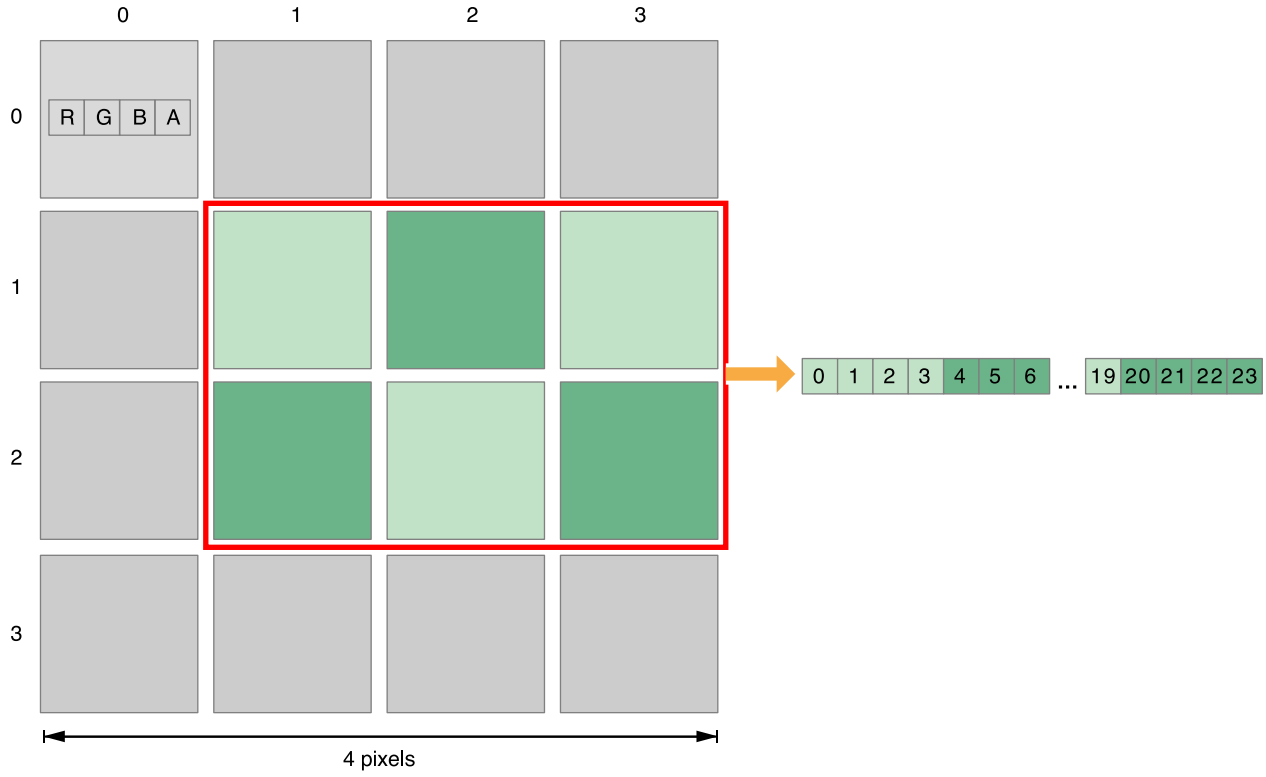
Performing a Generalized Buffer-To-Buffer Copy

To perform a generalized buffer-to-buffer copy which accommodates the case where the buffer data is conceptually multidimensional, call:

```
void gcl_memcpy_rect(
                        void *dst,
                        const void *src,
                         const size_t dst_origin[3],
                         const size_t src_origin[3],
                         const size_t region[3],
                         size_t dst_row_pitch,
                         size_t dst_slice_pitch,
                         size_t src_row_pitch,
                         size_t src_slice_pitch);
```

This function provides functionality similar to that of the OpenCL standard `clEnqueueCopyBufferRect` function; it copies a one-, two-, or three-dimensional rectangular region from the `src` pointer to the `dst` pointer, using the respective origin parameters to determine the points at which to read and write. As shown in Figure 9-1 (page 74), the `region` parameter specifies both the size and shape of the area to be copied.

Figure 9-1 The region specifies both the size and shape of an area



Since this is a buffer to buffer copy, all parameters are in bytes.

As with the OpenCL standard `clEnqueueCopyBufferRect` function, copying begins at the source offset (`src_origin`) and destination offset (`dst_origin`). Each byte of the region's width is copied from the source offset to the destination offset. After each width has been copied, the source and destination offsets are incremented by their respective source and destination row pitches.

After each two-dimensional rectangle is copied, the source and destination offsets are incremented by the source (`src_slice_pitch`) and destination (`dst_slice_pitch`) slice pitches respectively.

Parameter	Description
`dst`	Pointer to the memory to which the bytes will be copied. It can be either a regular host pointer or a device memory pointer created by the `gcl_malloc` function.

Parameter	Description
`src`	Pointer to the memory that is to be copied. As with the `dst` parameter, this can be a pointer to either host or device memory.

`dst_origin[3]`	Offset, in bytes, which specifies where in the destination buffer writing should start. It is calculated as:
	`dst_origin[0] * dst_row_pitch +`
	`dst_origin[1] * dst_slice_pitch +`
	`dst_origin[2]`

`src_origin[3]`	Offset, in bytes, which specifies where to begin reading in the source buffer. It is calculated as:
	`src_origin[0] * src_row_pitch +`
	`src_origin[1] * src_slice_pitch +`
	`src_origin[2]`

`region[3]`	The two- or three-dimensional region to copy.
`dst_row_pitch`	The length of each row in bytes to be used for the memory region associated with `dst_buffer`. If you specify `dst_row_pitch` to be `0`, OpenCL assigns `dst_row_pitch` to be `region[0]`.
`dst_slice_pitch`	After each two-dimensional rectangle is copied, the source and destination offsets are incremented by the source (`src_slice_pitch`) and destination (`dst_slice_pitch`) slice pitches respectively.
`src_row_pitch`	The length of each row in bytes to be used for the memory region associated with `src_buffer`. If you specify `src_row_pitch` to be `0`, OpenCL assigns `src_row_pitch` to be `region[0]`.
`src_slice_pitch`	After each two-dimensional rectangle is copied, the source and destination offsets are incremented by the source (`src_slice_pitch`) and destination (`dst_slice_pitch`) slice pitches respectively.

Important: Make sure that both buffers are of sufficient size to accommodate the copy.

The call to this function must occur within a block on a particular dispatch queue.

Releasing Buffer Objects

To avoid memory leaks, free buffer objects when they are no longer needed. Call the `gcl_free` function to free buffer objects created using the `gcl_malloc` function.

```
void   gcl_free(void *ptr);
```

Parameter	Description
`ptr`	Handle of the buffer object to be released.

Example: Allocating, Using, and Releasing Buffer Objects

In the Listing 9-1 (page 76), the host creates one input buffer and one output buffer, initializes the input buffer, calls the kernel (see Listing 9-2 (page 80)) to square each value in the input buffer, then checks the results.

Listing 9-1 Sample host function creates buffers then calls kernel function

```
#include <stdio.h>
#include <stdlib.h>
#include <OpenCL/opencl.h>

// Include the automatically-generated header which provides the
// kernel block declaration.
#include "kernels.cl.h"

#define COUNT 2048

static void display_device(cl_device_id device)
{
    char name_buf[128];
```

```
    char vendor_buf[128];

    clGetDeviceInfo(
            device, CL_DEVICE_NAME, sizeof(char)*128, name_buf, NULL);
    clGetDeviceInfo(
            device, CL_DEVICE_VENDOR, sizeof(char)*128, vendor_buf, NULL);

    fprintf(stdout, "Using OpenCL device: %s %s\n", vendor_buf, name_buf);
}

static void buffer_test(const dispatch_queue_t dq)
{
    unsigned int i;

    // We'll use a semaphore to synchronize the host and OpenCL device.
    dispatch_semaphore_t dsema = dispatch_semaphore_create(0);

    // Create some input data on the _host_ ...
    cl_float* host_input = (float*)malloc(sizeof(cl_float) * COUNT);

    // ... and fill it with some initial data.
    for (i=0; i<COUNT; i++)
        host_input[i] = (cl_float)i;

    // Let's use OpenCL to square this array of floats.
    // First, allocate some memory on our OpenCL device to hold the input.
    // We *could* write the output to the same buffer in this case,
    //  but let's use a separate buffer.

    // Memory allocation 1: Create a buffer big enough to hold the input.
    // Notice that we use the flag 'CL_MEM_COPY_HOST_PTR' and pass the
    // host-side input data.  This instructs OpenCL to initialize the
    // device-side memory region with the supplied host data.
    void* device_input =
                gcl_malloc(sizeof(cl_float)*COUNT, host_input,
```

```
                        CL_MEM_COPY_HOST_PTR);

// Memory allocation 2: Create a buffer to store the results
// of our kernel computation.
void* device_results = gcl_malloc(sizeof(cl_float)*COUNT, NULL, 0);

// That's it -- we're ready to send the work to OpenCL.
// Note that this will execute asynchronously with respect
// to the host application.
dispatch_async(dq, ^{

    cl_ndrange range = {
        1,          // We're using a 1-dimensional execution.
        {0},        // Start at the beginning of the range.
        {COUNT},    // Execute 'COUNT' work items.
        {0}         // Let OpenCL decide how to divide work items
                    // into workgroups.
    };

    square_kernel(
             &range, (cl_float*) device_input,
             (cl_float*) device_results );

    // The computation is done at this point,
    // but the results are still "on" the device.
    // If we want to examine the results on the host,
    // we need to copy them back to the host's memory space.
    // Let's reuse the host-side input buffer.
    gcl_memcpy(host_input, device_results, COUNT * sizeof(cl_float));

    // Okay -- signal the dispatch semaphore so the host knows
    // it can continue.
    dispatch_semaphore_signal(dsema);
});
```

```
    // Here the host could do other, unrelated work while the OpenCL
    // device works on the kernel-based computation...
    // But now we wait for OpenCL to finish up.
    dispatch_semaphore_wait(dsema, DISPATCH_TIME_FOREVER);

    // Test our results:
    int results_ok = 1;
    for (i=0; i<COUNT; i++)
    {
        cl_float truth = (cl_float)i * (cl_float)i;
        if (host_input[i] != truth) {
            fprintf(stdout,
                    "Incorrect result @ index %d: Saw %1.4f, expected %1.4f\n\n",
                     i, host_input[i], truth);
            results_ok = 0;
            break;
        }
    }

    if (results_ok)
        fprintf(stdout, "Buffer results OK!\n");


    // Clean up device-side memory allocations:
    gcl_free(device_input);

    // Clean up host-side memory allocations:
    free(host_input);
}


int main (int argc, const char * argv[])
{
    // Grab a CPU-based dispatch queue.
```

```
    dispatch_queue_t dq = gcl_create_dispatch_queue(CL_DEVICE_TYPE_CPU, NULL);
    if (!dq)
    {
        fprintf(stdout, "Unable to create a CPU-based dispatch queue.\n");
        exit(1);
    }

    // Display the OpenCL device associated with this dispatch queue.
    display_device(gcl_get_device_id_with_dispatch_queue(dq));

    buffer_test(dq);

    fprintf(stdout, "\nDone.\n\n");

    dispatch_release(dq);
}
```

Listing 9-2 Sample kernel squares an input array

```
// A very simple kernel which squares an input array.  The results are
// stored in another buffer, but could just as well be stored in the
// 'input' array -- that's a developer choice.

// Note that input and results are declared as 'global', indicating
// that they point to allocations in the device's global memory.

kernel void square( global float* input, global float* results )
{
    // We've launched our kernel (in the host-side code) such that each
    // work item squares one incoming float.  The item each work item
    // should process corresponds to its global work item id.
    size_t index = get_global_id(0);

    float val = input[index];
```

```
    results[index] = val * val;
}
```

Sharing Data Between OpenCL and OpenGL

OpenGL (Open Graphics Library) is an API for writing applications that produce two- and three-dimensional computer graphics. OpenCL and OpenGL are designed to interoperate. In particular, OpenCL and OpenGL can share data, which reduces overhead. For example, OpenGL objects and OpenCL memory objects created from OpenGL objects can access the same memory. In addition, GLSL (OpenGL Shading Language) shaders and OpenCL kernels can access the shared data.

To make sure OpenCL and OpenGL work together smoothly:

- Set your program up to do all its computation and rendering on the GPU.

  This will improve performance because you'll avoid having to transfer data between the host and the GPU.
- Allocate memory to ensure that data is shared efficiently.

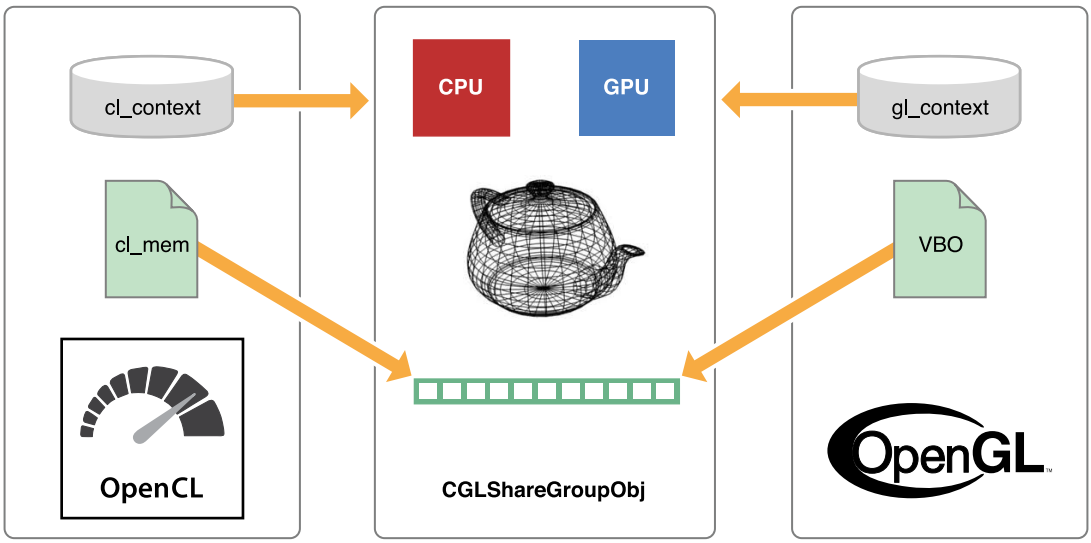
This chapter shows how the OpenCL API can be used to create OpenCL memory objects from OpenGL vertex buffer objects (VBOs), texture objects, and renderbuffer objects. It also shows how to create OpenCL buffer objects from OpenGL buffer objects and how to create an OpenCL image object from an OpenGL texture or renderbuffer object.

Sharegroups

To allow an OpenCL application to access an OpenGL object, use an OpenCL context that is created from an OpenGL **sharegroup** (`CGLShareGroupObj`) object. Open*CL* objects you create in this context will reference these Open*GL* objects. When an OpenCL context is connected to an OpenGL sharegroup object in this way, both the OpenCL and OpenGL contexts can reference the same objects.

Figure 10-1 (page 83) illustrates a typical scenario in which Open*CL* generates geometry on the GPU and Open*GL* renders the shared geometry, also on the GPU. They may see what they are looking at as different objects - in this case, OpenGL sees the data as a VBO and OpenCL sees it as a `cl_mem`. The result is that both the OpenCL and OpenGL contexts end up referencing the same sharegroup (CGLShareGroupObj) object, see the same devices, and access this shared geometry.

Figure 10-1 OpenGL and OpenCL share data using sharegroups



To interoperate between OpenCL and OpenGL:

1. Set the sharegroup:

```
CGLContextObj cgl_context = CGLGetCurrentContext();
CGLShareGroupObj sharegroup = CGLGetShareGroup(cgl_context);
gcl_gl_set_sharegroup(sharegroup);
...
```

2. After the sharegroup has been set, you can create OpenCL memory objects from the existing OpenGL objects:
   - To create an OpenCL buffer object from an OpenGL buffer object, call:

```
void * gcl_gl_create_ptr_from_buffer(GLuint bufobj);
```

   - To create an OpenCL image object from an OpenGL texture object, call:

```
cl_image gcl_gl_create_image_from_texture(
            GLenum texture_target,
            GLint mip_level,
            GLuint texture);
```

- To create an OpenCL two-dimensional image object from an OpenGL render buffer object, call:

```
cl_image gcl_gl_create_image_from_renderbuffer(GLuint render_buffer);
```

Synchronizing Access To Shared OpenCL / OpenGL Objects

To ensure data integrity, the application must synchronize access to objects shared by OpenCL and OpenGL. Failure to provide such synchronization may result in race conditions or other undefined behavior including non-portability between implementations. For information about synchronizing OpenCL and OpenGL events and fences, see "Controlling OpenCL / OpenGL Interoperation With GCD" (page 91).

Example: OpenCL and OpenGL Sharing Buffers

This code example is excerpted from the OpenCL Procedural Geometric Displacement Example with GCD Integration. It shows how OpenCL can bind to existing OpenGL buffers in order to avoid copying data back from a compute device when it needs to use the results of the computation for rendering. In this example, an OpenCL compute kernel displaces the vertices of an OpenGL-managed vertex buffer object (**VBO**). The compute kernel calculates several octaves of procedural noise to push the resulting vertex positions outwards and calculates new normal directions using finite differences. The vertices are then rendered by OpenGL.

To build this application:

1. Include the header file generated by Xcode. This header file contains the kernel block declaration.

   ```
   #include "displacement_kernel.cl.h"
   ```

2. Declare the dispatch queue and the dispatch semaphore used for synchronization between OpenCL and OpenGL. For more information about the dispatch semaphores, see "Synchronizing A Host With OpenCL Using A Dispatch Semaphore" (page 92).

   ```
   static dispatch_queue_t queue;
   static dispatch_semaphore_t cl_gl_semaphore;
   ```

3. Declare the memory objects that will hold input and output data.

```
static void *InputVertexBuffer;
static void *OutputVertexBuffer;
static void *OutputNormalBuffer;
```

4. Set the sharegroup, create the dispatch queue and the dispatch semaphore, and get the compute device.

```
static int setup_compute_devices(int gpu)
{
    int err;
    size_t returned_size;
    ComputeDeviceType =
      gpu ? CL_DEVICE_TYPE_GPU : CL_DEVICE_TYPE_CPU;

    printf(SEPARATOR);
    printf("Using active OpenGL context...\n");

    // Set the sharegroup.
    CGLContextObj kCGLContext =
                  CGLGetCurrentContext();
    CGLShareGroupObj kCGLShareGroup =
                  CGLGetShareGroup(kCGLContext);

    gcl_gl_set_sharegroup(kCGLShareGroup);

    // Create a dispatch queue.
    queue = gcl_create_dispatch_queue(
                  ComputeDeviceType, NULL);
    if (!queue)
    {
        printf("Error: Failed to create a dispatch queue!\n");
        return EXIT_FAILURE;
    }

    // Create a dispatch semaphore.
```

```
cl_gl_semaphore = dispatch_semaphore_create(0);
if (!cl_gl_semaphore)
{
    printf("Error:
       Failed to create a dispatch semaphore!\n");
    return EXIT_FAILURE;
}

// Get the device ID.
ComputeDeviceId =
     gcl_get_device_id_with_dispatch_queue(queue);

// Report the device vendor and device name.
cl_char vendor_name[1024] = {0};
cl_char device_name[1024] = {0};
err = clGetDeviceInfo(
        ComputeDeviceId, CL_DEVICE_VENDOR,
        sizeof(vendor_name), vendor_name,
        &returned_size);
err|= clGetDeviceInfo(
        ComputeDeviceId, CL_DEVICE_NAME,
        sizeof(device_name), device_name,
        &returned_size);
if (err != CL_SUCCESS)
{
    printf(
        "Error: Failed to retrieve device info!\n");
    return EXIT_FAILURE;
}

printf(SEPARATOR);
printf(
        "Connecting to %s %s...\n",
         vendor_name, device_name);
return CL_SUCCESS;
```

```
}
```

5. Create the memory objects that will hold input and output data and initialize the input memory object with the input data.

   This example creates a new memory object, `InputVertexBuffer`, using the `gcl_malloc` function. It then uses the `gcl_gl_create_ptr_from_buffer` function to create OpenCL objects (`OutputVertexBuffer` and `OutputNormalBuffer`) from existing OpenGL VBOs (`VertexBufferId` and `NormalBufferId`). The `InputVertexBuffer` memory object is initialized with input data stored in `VertexBuffer`, using the `gcl_memcpy` function.

```
static int setup_compute_memory()
{
    size_t bytes =
        sizeof(float) * VertexComponents * VertexElements;

    printf(SEPARATOR);
    printf("Allocating buffers on compute device...\n");

    InputVertexBuffer = gcl_malloc(bytes, NULL, 0);
    if (!InputVertexBuffer)
    {
        printf("Failed to create InputVertexBuffer!\n");
        return EXIT_FAILURE;
    }
    OutputVertexBuffer =
        gcl_gl_create_ptr_from_buffer(VertexBufferId);
    if (!OutputVertexBuffer)
    {
        printf("Failed to create OutputVertexBuffer!\n");
        return EXIT_FAILURE;
    }

    OutputNormalBuffer =
        gcl_gl_create_ptr_from_buffer(NormalBufferId);
    if (!OutputNormalBuffer)
    {
```

```
        printf("Failed to create OutputNormalBuffer!\n");
        return EXIT_FAILURE;
    }

    dispatch_async(queue,
        ^{gcl_memcpy(
           InputVertexBuffer,
           VertexBuffer, VertexBytes);});

    return CL_SUCCESS;
}
```

6. Create the compute kernel from the kernel block generated by Xcode.

```
static int setup_compute_kernels(void)
{
    int err = 0;

    ComputeKernel =
       gcl_create_kernel_from_block(displace_kernel);

    // Get the maximum work group size for executing
    // the kernel on the device.
    size_t max = 1;
    err = clGetKernelWorkGroupInfo(
        ComputeKernel, ComputeDeviceId,
        CL_KERNEL_WORK_GROUP_SIZE, sizeof(size_t),
        &max, NULL);
    if (err != CL_SUCCESS)
    {
        printf("Error:
          Failed to retrieve kernel work group
          info! %d\n", err);
        return EXIT_FAILURE;
    }
```

```
    MaxWorkGroupSize = max;
    printf("Maximum Workgroup Size '%d'\n",
         MaxWorkGroupSize);

    return CL_SUCCESS;
}
```

7. Use the `cl_ndrange` structure to specify the data parallel range over which to execute the kernel. Dispatch the kernel block asynchronously using the `dispatch_async` function. After you dispatch the kernel, signal the dispatch semaphore to indicate that OpenGL can now access the shared resources.

```
static int recompute(void)
{
    size_t global[2];
    size_t local[2];
    cl_ndrange ndrange;

    uint uiSplitCount = ceilf(sqrtf(VertexElements));
    uint uiActive = (MaxWorkGroupSize / GroupSize);
    uiActive = uiActive < 1 ? 1 : uiActive;

    uint uiQueued = MaxWorkGroupSize / uiActive;

    local[0] = uiActive;
    local[1] = uiQueued;

    global[0] = divide_up(uiSplitCount, uiActive) * uiActive;
    global[1] = divide_up(uiSplitCount, uiQueued) * uiQueued;

    ndrange.work_dim = 2;
    ndrange.global_work_offset[0] = 0;
    ndrange.global_work_offset[1] = 0;
    ndrange.global_work_size[0] = global[0];
    ndrange.global_work_size[1] = global[1];
    ndrange.local_work_size[0] = local[0];
```

```
    ndrange.local_work_size[1] = local[1];

    dispatch_async(queue,
                   ^{
                     displace_kernel(&ndrange,
                     InputVertexBuffer,
                     OutputNormalBuffer,
                     OutputVertexBuffer,
                     ActualDimX, ActualDimY,
                     Frequency, Amplitude, Phase,
                     Lacunarity, Increment, Octaves,
                     Roughness, VertexElements);
                     dispatch_semaphore_signal(
                       cl_gl_semaphore);
                    });
    return 0;
}
```

8. Free the OpenCL memory objects, release the compute kernel, the dispatch queue, and the semaphore.

```
static void shutdown_opencl(void)
{
    gcl_free(InputVertexBuffer);
    gcl_free(OutputVertexBuffer);
    gcl_free(OutputNormalBuffer);

    clReleaseKernel(ComputeKernel);

    if(VertexBuffer)
        free(VertexBuffer);
    if(NormalBuffer)
        free(NormalBuffer);
    dispatch_release(cl_gl_semaphore);
    dispatch_release(queue);
}
```

Controlling OpenCL / OpenGL Interoperation With GCD

An application running on a host (a CPU) can route work or data (possibly in disparate chunks) to a device using the standard OpenCL and OpenGL APIs and OS X v10.7 extensions. While the device does the work it has been assigned, the host can continue working asynchronously. Eventually, the host needs results that have not yet been generated by the device performing the work. At that point, the host waits for the device to notify it that the assigned work has been completed.

OpenCL and OpenGL can also share work and data. Typically, use OpenCL to generate or modify buffer data which is then rendered by OpenGL. Or, you might use OpenGL to create an image and then post-process it using OpenCL. In either case, make sure you synchronize so that processing occurs in proper order.

You can always make sequential hard-coded calls to standard OpenCL and OpenGL functions in order to obtain fine-grained synchronization when working on shared data. You can always just intermix calls to OpenGL and OpenCL functions. (See the OpenGL and OpenGL specifications for more information.)

This chapter shows how to synchronize processing programmatically using GCD queues. You can use GCD to synchronize:

- A host with OpenCL. (See "Using GCD To Synchronize A Host With OpenCL" (page 91).)
- A host with OpenCL using a dispatch semaphore. (See "Synchronizing A Host With OpenCL Using A Dispatch Semaphore" (page 92).)
- Multiple OpenCL Queues. (See "Synchronizing Multiple Queues" (page 97).)

Using GCD To Synchronize A Host With OpenCL

In Listing 11-1 (page 91), the host enqueues data in two queues to GCD. In this example, the queued data is processed while the host continues to do its own work. When the host needs the results, it waits for both queues to complete their work.

Listing 11-1 Synchronizing the host with OpenCL processing

```
// Create a workgroup so host can wait for results from more than one kernel.
dispatch_group_t group = dispatch_group_create();
```

```
// Enqueue some of the data to the add_arrays_kernel on q0.
dispatch_group_async(group, q0,
                     ^{  // Because the call is asynchronous,
                         // the host will not wait for the results.
                         cl_ndrange ndrange = { 1, {0}, {N/2}, {0} };
                         add_arrays_kernel(&ndrange, a, b, c);
                      });

// Enqueue some of the data to the add_arrays_kernel on q1.
dispatch_group_async(group, q1,
                     ^{  // Because the call is asynchronous,
                         // the host will not wait for the results.
                         cl_ndrange ndrange = { 1, {N/2}, {N/2}, {0} };
                         add_arrays_kernel(&ndrange, a, b, c);
                      });

// Perform more work independent of the work being done by the kernels.
//  ...
// At this point, the host needs the results before it can proceed.
// So it waits for the entire workgroup (on both queues) to complete its work.
dispatch_group_wait(group, DISPATCH_TIME_FOREVER);
```

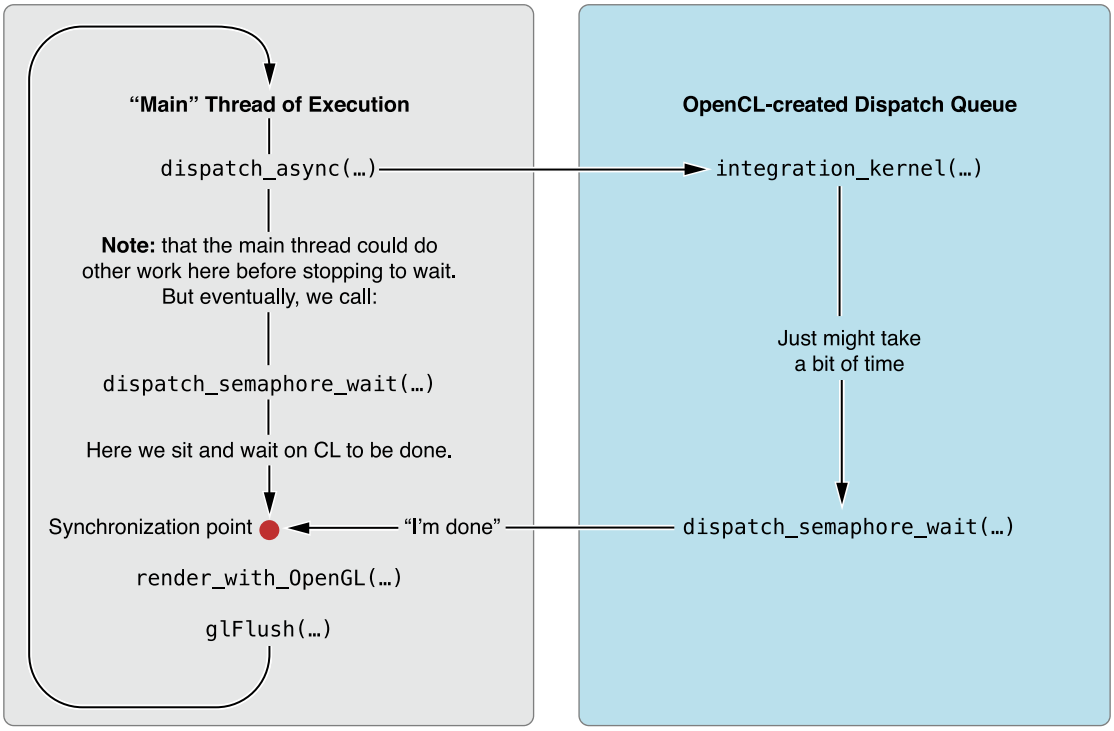
Synchronizing A Host With OpenCL Using A Dispatch Semaphore

Listing 11-2 (page 93) illustrates how you can use OpenCL and OpenGL together in an application. In this example, two vertex buffer objects (VBOs) created in OpenGL (not shown) represent the positions of some objects in an N-body simulation. OpenCL memory objects created from these VBOs (line [2]) allow an OpenCL kernel to operate directly on the device memory containing this data. The kernel updates these positions according to some algorithm, expressed as a per-object operation in the included kernel. The objects are then rendered in the resulting VBO using OpenGL (commented, but not shown, at [4]).

OpenCL is updating the positions on a dispatch queue that runs asynchronously with respect to the thread that does the OpenGL rendering. To ensure that the objects are not rendered before the kernel has finished updating the positions, the application uses a **dispatch semaphore** mechanism.

The `dispatch_semaphore_t` (line [1]) is created before the main loop begins. In the block submitted to the dispatch queue created in OpenCL, just after the kernel call, the semaphore is signaled. Meanwhile, the "main" thread of execution has continued along -- perhaps doing more work -- eventually arriving at the call to `dispatch_semaphore_wait(...)` (line [3]). The main thread stops at this point and waits until the post-kernel signal "flips" the semaphore. Once that occurs, the code can continue to the OpenGL rendering portion of the code, safe in the knowledge that the position update for this round is complete.

Figure 11-1 Rendering loop—each pass on the main thread creates a new frame for display



Listing 11-2 Synchronizing a host with OpenCL using a dispatch semaphore

```
// In this case, the kernel code updates the position of the vertex.
// ...

// The host code:

    // Create the dispatch semaphore.                                    [1]
    dispatch_queue_t queue;
    dispatch_semaphore_t cl_gl_semaphore;

    void *pos_gpu[2], *vel_gpu[2];
```

```
GLuint vbo[2];
float *host_pos_data, *host_vel_data;
int num_bodies;
int curr_read_index, curr_write_index;

// The extern OpenCL kernel declarations.

extern void (^integrateNBodySystem_kernel)(const cl_ndrange *ndrange,
                                    float4 *newPos, float4 *newVel,
                                    float4 *oldPos, float4 *oldVel,
                                    float deltaTime, float damping,
                                    float softening, int numBodies,
                                    size_t sharedPos);


void initialize_cl()
{
    gcl_gl_set_sharegroup(CGLGetShareGroup(CGLGetCurrentContext());

    // Create a CL dispatch queue.
    queue = gcl_create_dispatch_queue(CL_DEVICE_TYPE_GPU, NULL);

    // Create a dispatch semaphore to support CL/GL data sharing.
    cl_gl_semaphore = dispatch_semaphore_create(0);

    // Create CL objects from GL VBOs that have already been created.     [2]
    pos_gpu[0] = gcl_gl_create_ptr_from_buffer(vbo[0]);
    pos_gpu[1] = gcl_gl_create_ptr_from_buffer(vbo[1]);

    vel_gpu[0] = gcl_malloc(sizeof(float4)*num_bodies, NULL, 0);
    vel_gpu[1] = gcl_malloc(sizeof(float4)*num_bodies, NULL, 0);


    // Allocate and generate position and velocity data
```

```
    // in host_pos_data and host_vel_data.

    …

    // Initialize CL buffers with host position and velocity data.
    dispatch_async(queue,
                   ^{gcl_memcpy(pos_gpu[curr_read_index], host_pos_data,
                                                sizeof(float4)*num_bodies);
                     gcl_memcpy(vel_gpu[curr_read_index], host_vel_data,
                                                sizeof(float4)*num_bodies);});
}


void execute_cl_gl_main_loop()
{
    // Queue CL kernel to dispatch queue.
    dispatch_async(queue,
                   ^{
                     ndrange_t ndrange = { 1, {0}, {num_bodies} } ;
                    // Get local workgroup size that kernel can use for
                    // device associated with queue.
                    gcl_get_kernel_block_workgroup_info(
                               integrateNBodySystem_kernel,
                               CL_KERNEL_WORK_GROUP_SIZE,
                               sizeof(size_t), &nrange.local_work_size[0],
                               NULL);
                    // Queue CL kernel to dispatch queue.
                    integrateNBodySystem_kernel(&ndrange,
                                pos_gpu[curr_write_index],
                                vel_gpu[curr_write_index],
                                pos_gpu[curr_read_index],
                                vel_gpu[curr_read_index],
                                damping, softening, num_bodies,
                                sizeof(float4)*ndrange.local_work_size[0]);
```

```
                    // Signal the dispatch semaphore to indicate that
                    // GL can now use resources.
                    dispatch_semaphore_signal(cl_gl_semaphore);});

    // Do work not related to resources being used by CL in dispatch block.

    // Need to use VBOs that are being used by CL so wait for the CL commands
    // in dispatch queue to be issued to the GPU's command-buffer.       [3]
    dispatch_semaphore_wait(cl_gl_semaphore, DISPATCH_TIME_FOREVER);

    // Bind VBO that has been modified by CL kernel.
    glBindBuffer(GL_ARRAY_BUFFER, pos_gpu[curr_write_index]);

    // Now render with GL.                                               [4]

    // Flush GL commands.
    glFlush();
}

void release_cl()
{
    gcl_free(pos_gpu[0]);
    gcl_free(pos_gpu[1]);
    gcl_free(vel_gpu[0]);
    gcl_free(vel_gpu[1]);

    dispatch_release(cl_gl_semaphore);
    dispatch_release(queue);
}
```

Synchronizing Multiple Queues

In Listing 11-3 (page 97), the host enqueues data in two queues to GCD. The second queue waits for the first queue to complete its processing before doing its work. The host application does not wait for completion of either queue.

Listing 11-3 Synchronizing multiple queues

```
// Create the workgroup which will consist of just the work items
// that must be completed first.
dispatch_group_t group = dispatch_group_create();
dispatch_group_enter(group);

// Start work on the workgroup.
dispatch_async(q0,
              ^{
                      cl_ndrange ndrange = { 1, {0}, {N/2}, {0} };
                      add_arrays_kernel(&ndrange, a, b, c);
                      dispatch_group_leave(group);
               });

// Simultaneously enqueue data on q1,
// but immediately wait until the workgroup on q0 completes.
dispatch_async(q1,
             ^{
                     // Wait for the work of the group to complete.
                     dispatch_group_wait(group, DISPATCH_TIME_FOREVER);
                     cl_ndrange ndrange = { 1, {N/2}, {N/2}, {0} };
                     add_arrays_kernel(&ndrange, a, b, c);
             });

// Host application does not wait.
```

Using IOSurfaces With OpenCL

An IOSurface is an abstraction for sharing image data. IOSurfaces are an efficient way to manage image memory because when you use an IOSurface, if no copy is necessary, no time is wasted on making a copy. An IOSurface transcends APIs, architectures, address spaces, and processes.

> **Note:** IOSurfaces are an independent technology on the Mac platform, but OS X provides facilities for using them with OpenCL when possible.

If you create an OpenCL image from an IOSurface, you can use the OpenCL C language to modify the image data, taking full advantage of the parallelism OpenCL gives you. You use the IOSurface's ID to pass the IOSurface from process to process, so that completely separate applications can share the same object. This makes sharing an IOSurface between devices very easy.

If you create an OpenCL image memory object from an existing IOSurface, you can modify the data contained in the IOSurface either in your main program running on the CPU, or in an OpenCL kernel running on either the CPU or a GPU.

If you want to modify or read the IOSurface directly on the host, then you need to first call the `IOSurfaceLock` function. Call the `IOSurfaceUnlock` function when done. Otherwise, you can just use the image in OpenCL as if it were a normal, old non-IOSurface-backed image.

Creating Or Obtaining An IOSurface

You can either create an IOSurface in code or you can request an IOSurface from another running process such as Photo Booth. The underlying texture transfer mechanism for an IOSurface combines `GL_UNPACK_CLIENT_STORAGE_APPLE` and `GL_STORAGE_HINT_CACHED_APPLE` together. The transfer is done as a straight DMA to and from system memory and video memory with no format conversions of any kind (other than some GPU-specific memory layout details). No matter how many different OpenGL contexts (in the same process or not) bind a texture to an IOSurface, they all share the same system memory and GPU memory copies of the data.

Creating An Image Object from An IOSurface

Once you've created or obtained an IOSurface, before you use it in OpenCL, you need to create an OpenCL image memory object using the IOSurface. When you create the memory object, you are not making a copy; the image memory object points at the same memory as the original IOSurface. This makes using the IOSurface very efficient.

If you are using GCD to interact with the IOSurface, create the IOSurface-backed CL image as shown in Listing 12-1 (page 99).

Listing 12-1 Creating an IOSurface-backed CL Image

```
// Create a 2D image (depth = 0 or 1) or a 3D image (depth > 1).
// Can also be used to create an image from an IOSurfaceRef.
cl_image gcl_create_image(
  const cl_image_format *image_format,
  size_t image_width,
  size_t image_height,
  size_t image_depth,
  IOSurfaceRef io_surface);
```

If you are using the standard OpenCL API and not using GCD to create an IOSurface-backed OpenCL object, use `clCreateImageFromIOSurface2D` as shown in Listing 12-2 (page 99).

Listing 12-2 Extracting an Image From an IOSurface

```
cl_image_format image_format;
image_format.image_channel_order = CL_RGBA;
image_format.image_channel_data_type = CL_UNORM_INT8;
cl_mem image = clCreateImageFromIOSurface2D(
  context, flags, image_format, width, height, surface, &err
);
```

Sharing the IOSurface With An OpenCL Device

Sharing an IOSurface in OpenCL is very simple. The key is to lock the IOSurface properly.

If your CPU (host) is going to modify the IOSurface and then share it with an OpenCL device, you should lock the IOSurface before reading or writing to it, then unlock it before passing it to a kernel:

- The host creates or obtains the IOSurface and creates its OpenCL image object.
- If the host will be writing to the IOSurface, the host write-locks the IOSurface by calling the `IOSurfaceLock(..., write type lock)` function. If the host will only be reading from the IOSurface, the host read-locks it.
- The host writes to and reads from the IOSurface as necessary.
- The host unlocks the IOSurface by calling the `IOSurfaceUnlock(...)` function. This tells the system that you changed the data. You can then use the IOSurface-backed image in OpenCL. The IOSurface object handles any necessary read-locking internally for you.
- The host enqueues the OpenCL kernel, passing it the IOSurface.

The locking and unlocking are simply the minimal calls needed to give OS X enough information to ensure that each device always gets the latest, correct data.

If you will be using OpenCL to modify the IOSurface, you don't have to lock it. Just access the image memory object directly.

Autovectorizer

GPUs process scalar data efficiently, but a CPU needs vectorized data to keep it fully busy. Which means that, either you write a different kernel for every device your application might run on, each optimized for that device, or application performance will suffer.

The autovectorizer detects operations in a scalar program that can be run in parallel and converts them into vector operations that can be handled efficiently by today's CPUs. If you know you will be running your code through the autovectorizer, you can write a single, simple scalar program knowing that the autovectorizer will vectorize that code for you so that its performance on the CPU is maximized while the same code runs on the GPU as written.

> **Note:** Some GPUs also give higher performance when your code operates on vector data (such as loads and stores).

What the Autovectorizer Does

- Achieves performance improvements of up to the vector width of the CPU without additional effort on your part.
- Allows you to write one kernel that runs efficiently on the CPU or GPU.
- Packs work items together into vector instructions.
- Generates a loop over the entire workgroup.
- Runs by default whenever compiling kernels to a CPU.
- Workgroup size can be increased if autovectorization is successful.
- Runs by default when compiling to the CPU.

Writing Optimal Code For the CPU

To write code that can best be optimized by the autovectorizer:

Do

- Write a single kernel that can run on the CPU and GPU. Use appropriate data types (scalar or float) as needed by your algorithm.
- Move memory access out of control flow if possible; for example:

```
if (condition)
   a[index] = 1
else
   a[index] = 2
```

  should be coded as:

```
if (condition)
  tmp = 1;
else
  tmp = 2;

a[index] = tmp;
```

- When accessing array elements, it is best to access consecutive array elements in consecutive work-items. See Table 14-2 (page 123) for a real-world example of how to access consecutive array elements in consecutive work-items most efficiently.

Don't

- Write device-specific optimizations.
- Write work item ID-dependent control flow, if possible. (If this occurs in many places in the code, it would likely prevent autovectorization from succeeding.)

Enabling and Disabling the Autovectorizer In Xcode

You can disable or re-enable autovectorization in the Xcode build settings for your OpenCL apps. By default, the autovectorizer is enabled when you generate kernels in Xcode. Choose No to turn the autovectorizer off. This setting takes effect only for the CPU.

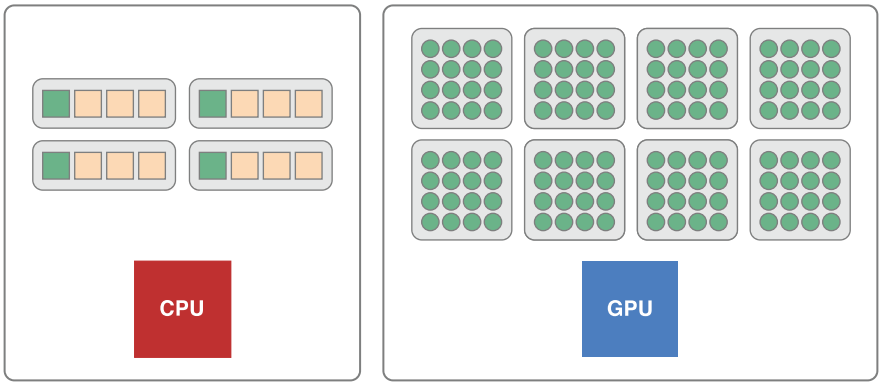
Setting	Type	Default	Command Line Flag
Auto-vectorizer	Boolean	YES	To enable: `-cl-auto-vectorize-enable` To disable: `-cl-autovectorize-disable`

Example: Autovectorization of Scalar Floats

OpenCL sees devices as having a number of compute cores and within them a number of processing elements. When scalar code runs on a CPU, it runs on each core but does not take advantage of the entire vector unit.

For example, on an SSE4 CPU, scalar code runs in one lane of the vector unit when it could be running in four lanes. The monitor would report that the CPU is completely busy because all the cores are running, but the CPU is actually only using a quarter of its vector width.

Figure 13-1 Before autovectorization: A simple float sent to the CPU and GPU



Listing 13-1 (page 103) is an example of a kernel that accepts simple floats:

Listing 13-1 Passing single floats into a kernel

```
kernel void add_arrays(global float* a, global float* b, global float* c)
{
  size_t i = get_global_id(0);
  c[i] = a[i] + b[i];
}
```

If you pass simple floats into a kernel, the kernel does a scalar addition; operating on one data element at a time. As illustrated in Figure 13-1 (page 103), if you pass a scalar float to a GPU, the GPU becomes fully engaged in processing the data. If you pass this same float to a CPU, only one quarter of the vector width of the processing element in each core is used.

You could instead pass `float4*` parameters to the kernel, that makes the addition a vector addition. The computation is now specialized for the CPU. It would extract as much work as possible from the CPU but leave the GPU idle.

Without the autovectorizer, you would have to write multiple device-specific, non-scalar kernels, one for the CPU and one for the GPU.

Tuning Performance On the GPU

GPUs and CPUs have fundamentally different architectures and so require different optimizations for OpenCL. A CPU has a relatively small number of processing elements and a large amount of memory (both a large cache and a much larger amount of RAM available on the circuit board). A GPU has a relatively large number of processing elements and usually has less memory than a CPU. Therefore, the code that runs fastest on a GPU will be designed to take up less memory and take advantage of the GPU's superior processing power. In addition, GPU memory access is fast when the access pattern matches the memory architecture, so the code should be designed with this in mind.

It is possible to write OpenCL code that can run efficiently on both a CPU and a GPU. However, to obtain optimal performance it is usually necessary to write different code for each type of device.

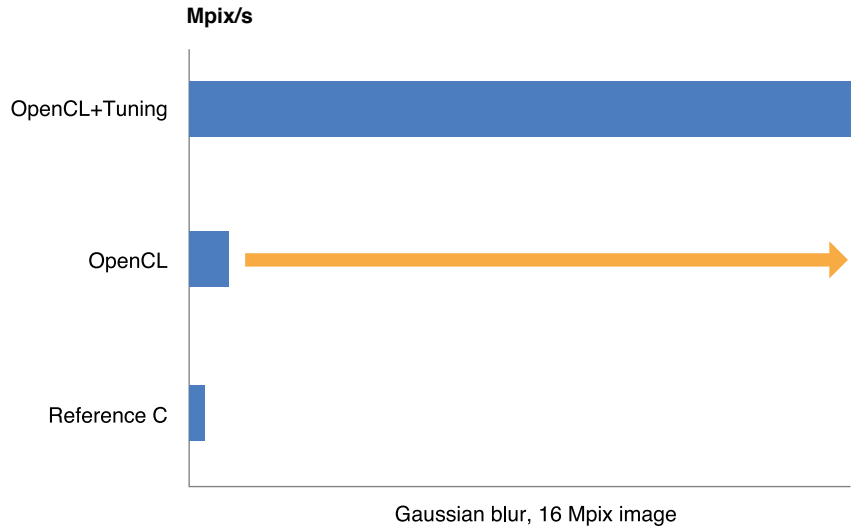
This chapter focuses on how to improve performance on the GPU. It begins by describing the significant performance improvements on the GPU that can be obtained through tuning (see "Why You Should Tune" (page 106)), lists APIs you can use to time code execution (see "Measuring Performance On Devices" (page 107)), describes how you can estimate the optimal performance of your GPU devices (see "Generating the Compute/Memory Access Peak Benchmark" (page 108)), describes a protocol that can be followed to tune GPU performance (see "Tuning Procedure" (page 112)), then steps through an example in which performance improvement is obtained. (See "Improving Performance On the CPU" (page 130) for suggestions for optimizing performance on the CPU.)See Table 14-1 (page 118) at the end of the chapter for generally applicable suggestions for measuring and improving performance on most GPUs.

Note: This chapter is based upon a talk at WWDC 2011 called What's New in OpenCL?.

Why You Should Tune

Tuning your OpenCL code for the GPU can result in a two- to ten-fold improvement in performance. Figure 14-1 (page 106) illustrates typical improvements in processing speed obtained when an application that executes a Gaussian blur on a 16 MP image was optimized. The process followed to optimize this code is described in "Example: Tuning Performance Of a Gaussian Blur" (page 114).

Figure 14-1 Improvement expected



Before Optimizing Code

Before you decide to optimize code:

1. Decide whether the code really needs to be optimized. Optimization can take significant time and effort. Weigh the costs and benefits of optimization before starting any optimization effort.
2. Estimate optimal performance. Run some simple kernels on your GPU device to estimate its capabilities. You can use the techniques described in "Measuring Performance On Devices" (page 107) to measure how long kernel code takes to run. See "Generating the Compute/Memory Access Peak Benchmark" (page 108) for examples of code you can use to test memory access speed and processing speed.

3. Generate or collect sample data to feed through each iteration of optimization. Run the unoptimized original code through the sample code and save the results. Then run each major iteration of the optimized code against the same data and compare the results to the original results to ensure your output has not been corrupted by the changed code.

Measuring Performance On Devices

The point of optimizing an OpenCL application is to get it to run faster. At each step of the optimization process, you need to know how fast the optimized code takes to run. To determine how much time it takes a kernel to execute:

- Start a timer just before you call the kernel:

  ```
  cl_timer gcl_start_timer(void);
  ```

  Call this function to start the timer. It returns a `cl_timer` that you use to stop the timer.

- Stop the timer immediately after the kernel returns:

  ```
  double gcl_stop_timer(cl_timer timer);
  ```

  Call this function to stop the timer created when you called the `gcl_start_timer` function. It returns the elapsed time in seconds since the call to `cl_start_timer` associated with the `timer` parameter.

Measuring the execution time of several consecutive calls to the same kernel(s) usually improves the reliability of results. Because "warming-up" the device also improves consistency of benchmarking results, it's recommended that you call the code that enqueues the kernel at least once before you begin timing. Listing 14-1 (page 107) stores performance information about a kernel that it enqueues. Notice how the loop index starts at `-2` but the timer is started when the index has been incremented to `0`. :

Listing 14-1 Sample benchmarking loop on the kernel

```
const int iter = 10; // number of iterations to benchmark
cl_timer blockTimer;
for (int it = -2; it < iter; it++) { // Negative values not timed: warm-up
  if (it == 0) {                     // start timing
    blockTimer = gcl_start_timer(void);
  }
  <code to benchmark>
}
clFinish(queue);
```

```
gcl_stop_timer(blockTimer);

// t = execution time for one iteration (s)
double t = blockTimer / (double)iter;
```

Generating the Compute/Memory Access Peak Benchmark

Before you optimize your code, you need to estimate how fast your particular GPU device is when accessing memory and when executing floating point operations. You can use two simple kernels to benchmark these capabilities:

- Copy kernel

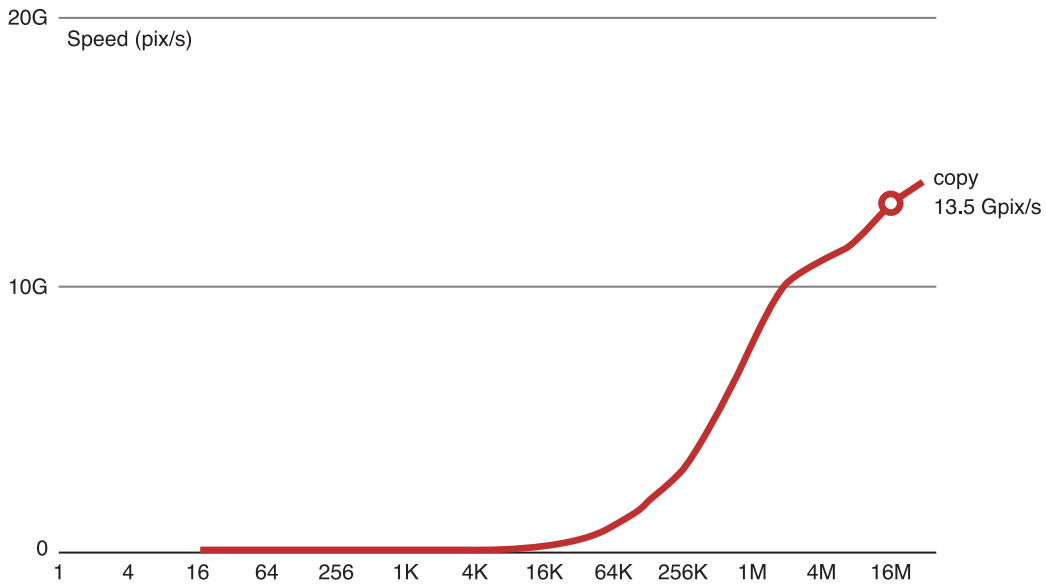
  The kernel in Listing 14-2 (page 108) reads a large image (results shown in this chapter come from processing a 16 megapixel (MP) image) stored in an input buffer and copies the image to an output buffer. Running this code gives a good indication of the best possible memory access speed to expect from a GPU.

Listing 14-2 Copy kernel

```
kernel void copy(global const float * in,
                 global float * out,
                 int w,int h)
{
  int x = get_global_id(0); // (x,y) = pixel to process
  int y = get_global_id(1); // in this work item
  out[x+y*w] = in[x+y*w]; // Load and Store
}
```

Figure 14-2 (page 109) graphs copy kernel speed as a function of image size. Running the copy kernel on our GPU, we were able to copy 13.5 gigapixels per second. This is a good indication of the maximum speed of memory access on this device.

Figure 14-2 Copy kernel performance



- MAD kernel

  In a Multiply plus ADd (**MAD**) kernel such as Listing 14-3 (page 109), a large graphic image is read in, some floating point operations are performed on the image data, and the results are saved to an output buffer. The MAD kernel can be used to estimate optimal processing speed of the GPU, but when using the MAD kernel, you must add in enough floating point operations to ensure that memory constraints do not mask floating point capabilities.

Listing 14-3 MAD kernel with 3 flops

```
kernel void mad3(global const float * in,
                 global float * out,
                 int w,int h) {
  int x = get_global_id(0);
  int y = get_global_id(1);
  float a = in[x+y*w];           // Load
  float b = 3.9f * a * (1.0f-a); // Three floating point ops
  out[x+y*w] = b;                // Store
}
```

The MAD benchmark shows the compute/memory ratio in the case of a single sequence of dependent operations. Compute kernels can reach much higher compute/memory ratios when they execute several independent dependence chains. For example, a matrix * matrix multiply kernel can process nearly 2 Tflop/s on the same GPU.

As shown in Figure 14-3 (page 110), when we added three floating point operations to the copy kernel code (the top (red) line), we were still able to process 11.9 GP/s. This indicates that with only three flops, processing remains memory-bound.

Figure 14-3 MAD/copy kernel performance with 3 flops

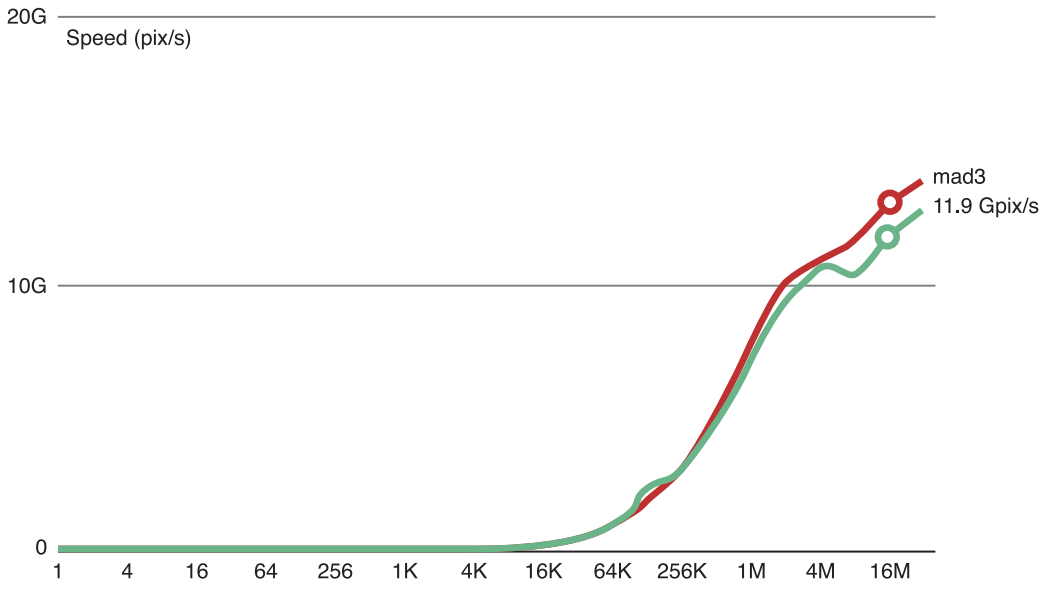


Figure 14-4 (page 111) shows that when we added six floating point operations to the copy kernel code (the top (red) line), we are still able to process 11.8 GP/s. This indicates that with six flops, processing is still memory-bound.

Figure 14-4 MAD/copy kernel performance with 6 flops

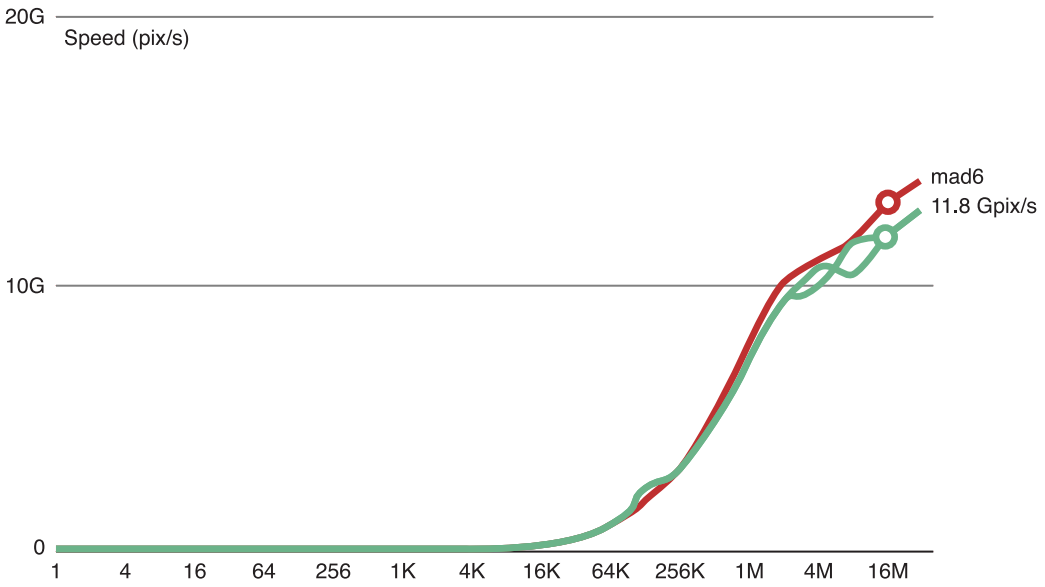
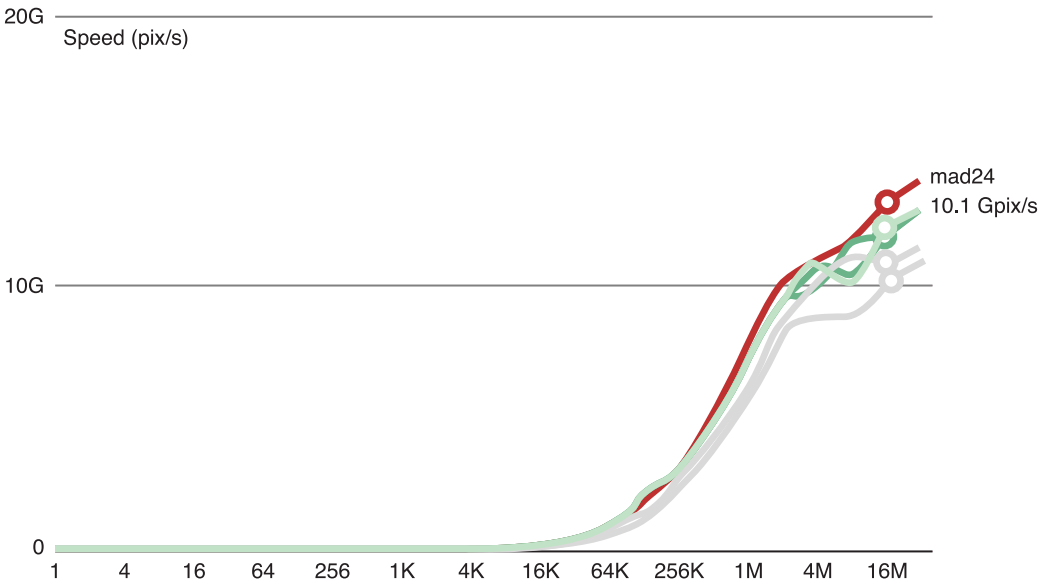


Figure 14-5 (page 111) shows that after we added 24 floating point operations to the copy kernel code (the red line), processing slowed to 10.1 GP/s. Because the reduction of processing speed is large enough to be considered significant, this result indicates that this kernel is a good benchmark for computational processing for this GPU.

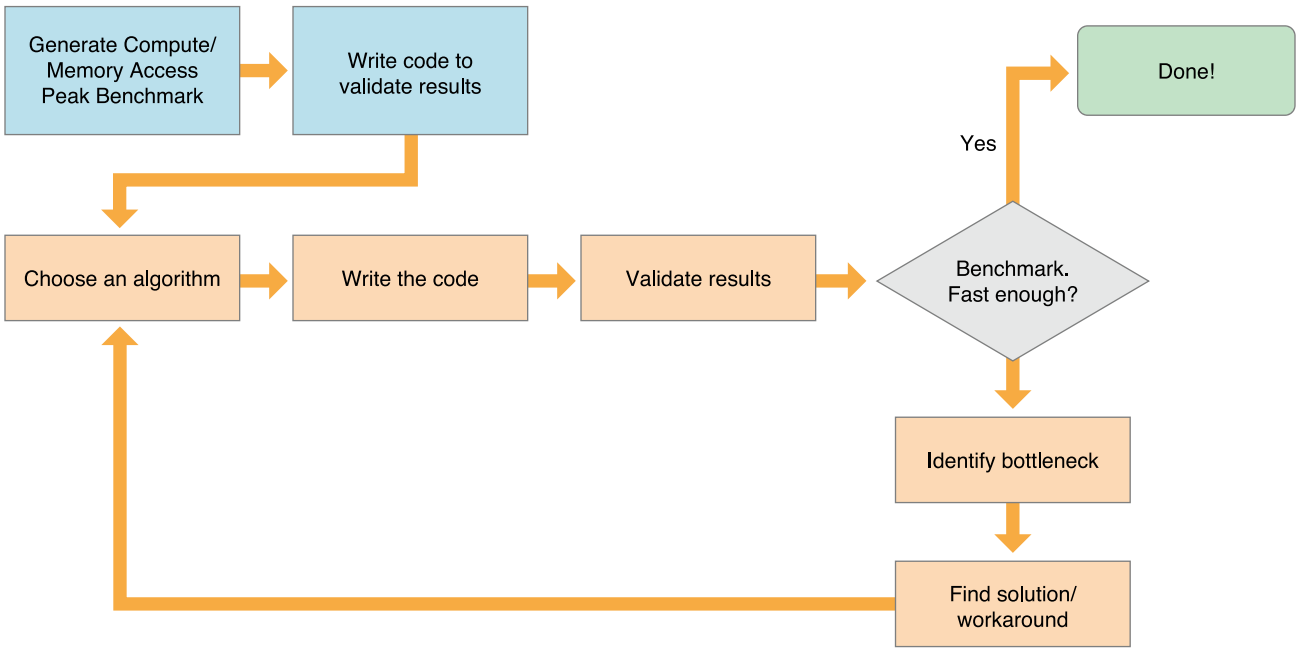
Figure 14-5 MAD/copy kernel performance with 24 flops



Tuning Procedure

Figure 14-6 (page 112) shows a typical process for optimizing a kernel that runs efficiently on the GPU:

Figure 14-6 Tuning procedure



1. Choose an efficient algorithm. OpenCL runs most efficiently if the algorithm is optimized to take advantage of the capabilities of all devices it runs on. See "Choosing An Efficient Algorithm" (page 113) for suggestions about how to evaluate potential algorithms.
2. Write code that runs efficiently on all target device(s). Each family of GPUs has a unique architecture. To get optimal performance from a GPU, you need to understand that GPU's architecture. For example, some GPU families perform best when memory access blocks are set to certain sizes, other GPU families work best when the number of items in a workgroup is a multiple of a particular number, and so on. Consult the manufacturer's literature for any GPU you wish to support to get details about that GPU's architecture. This document provides only general principles that should apply to most GPUs.

   See Table 14-1 (page 118) for suggestions.

   It's usually best to write scalar code first. In the second iteration, parallelize it. Next, create a version that minimizes memory usage.
3. Make sure to validate the results generated by each code version.
4. Benchmark. You can use the techniques described in "Measuring Performance On Devices" (page 107) to measure the speed of the benchmark code and your application code. If the performance is good enough, you are done.

5. Identify bottlenecks.
6. Find a solution or workaround.
7. Repeat this process until your performance approaches the optimization target.

Choosing An Efficient Algorithm

Consider the following when choosing an algorithm for your OpenCL application:

- The algorithm should be massively parallel, so that computation can be carried out by a large number of independent work items. For data parallel calculations on a GPU, OpenCL works best where many work items are submitted to the device.
- Minimize memory usage. The GPU has huge computing power; kernels are usually memory-bound. Consequently, algorithms with the fewest memory accesses or algorithms with a high compute-to-memory ratio are usually best for OpenCL applications. The compute-to-memory ratio is the ratio between the number of floating-point operations and the number of bytes transferred to and from memory.

  Try to maximize the number of independent dependency chains by grouping the computation of several output elements into one single work item.
- Moving data to or from OpenCL devices is expensive. OpenCL is most efficient on large datasets.

  OpenCL gives you complete control over memory allocation and host-device memory transfers. Your program will run much faster if you allocate memory on the OpenCL device, move your data to the device, do as much computation as possible on the device, then move it off—rather than repeatedly going through write-compute-read cycles.

  To avoid slow host-device transfers:

  - Whenever possible, aggregate several transfers into a single, larger transfer.
  - Design algorithms to keep the data on the device as long as possible.
- For data-parallel calculations on a GPU, OpenCL works best where there are a lot of work items submitted to the device; however, some algorithms are much more efficient than others.
- Use OpenCL built-in functions whenever possible. Optimal code will be generated for these functions.
- Balance precision and speed. GPUs are designed for graphics, where the requirements for precision are lower. The fastest variants are exposed in the OpenCL built-ins as `fast_`, `half_`, `native_` functions. The program build options provide control of some speed optimizations.
- Allocating and freeing OpenCL resources (memory objects, kernels, and so on) takes time. Reuse these objects whenever possible instead of releasing them and recreating them repeatedly. Note, however, that image objects can be reused only if they are the same size and pixel format as needed by the new image.
- Take advantage of the memory subsystem of the device.

When using memory on an OpenCL device, the local memory shared by all the work items in a single workgroup is faster than the global memory shared by all the workgroups on the device. Private memory, available only to a single work item, is even faster.

On the GPU, the memory access pattern is the most important factor. Use faster memory levels (local memory, registers) to counter the effects of a sub-optimal pattern and to minimize accesses to the slower global memory.

- Experiment with your code to find the kernel size that works best.

  Using smaller kernels can be efficient because each tiny kernel uses minimal resources. Breaking a job down into many small kernels allows for the creation of very large, efficient workgroups. On the other hand, it may require between 10-100 µs to start each kernel. When each kernel exits, the results must be stored in global memory. Because reading and writing to global memory is expensive, concatenating many small kernels into one large kernel may save considerable overhead.

  To find the kernel size that provides optimal performance, you will need to experiment with your code.

- OpenCL events on the GPU are expensive.

  You can use events to coordinate execution between queues, but there is overhead to doing so. Use events only where needed; otherwise take advantage of the in-order properties of queues.

- Avoid divergent execution:

  All threads scheduled together on a GPU must execute the same code. As a consequence, when executing a conditional, all threads execute both branches but their output is disabled when they are false branch. It is best to avoid conditionals (replace them with `a?x:y` operators) or use built-in functions.

Example: Tuning Performance Of a Gaussian Blur

The following example steps through the process of optimizing an application that performs a Gaussian blur on an image on a GPU. You can follow a similar protocol when tuning your GPU code.

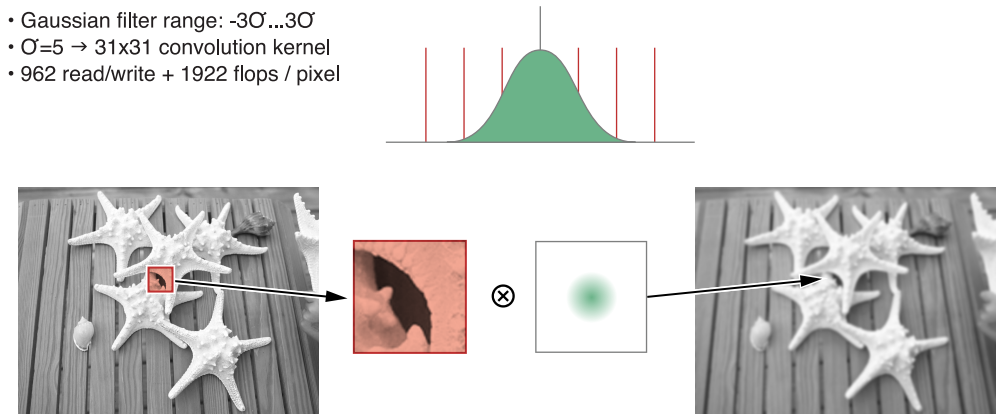
1. Estimate optimal performance.
2. Generate test code. It's probably easiest to write a reference version of the code on the host, save the result, then write code to compare the verified output to the output generated by your optimized code.
3. Choose an algorithm to implement our Gaussian blur:

   There are three possibilities:

   - Classic Two-Dimensional Convolution

Figure 14-7 (page 115) depicts the creation of a two-dimensional convolution using a 31 x 31 kernel for sigma=5. This translates to 31 times 31, or 961 input pixels for each pixel output. One addition and one multiplication is used for each input for a total of 961+1 I/O or 2 times 961 flops per pixel. These results are shown in the second row of Table 14-1 (page 118).

Figure 14-7 Classic two-dimensional convolution

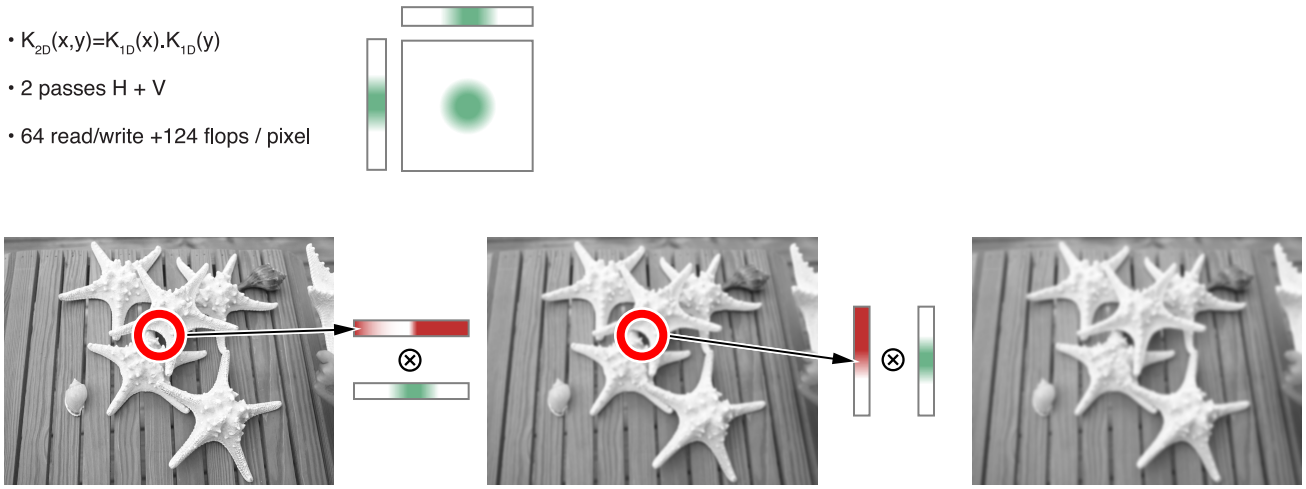


- Separable Two-Dimensional Convolution

  In this case, the algorithm is separable. It can be divided into two one-dimensional filters-one horizontal and one vertical, as shown in Figure 14-8 (page 115). By separating the dimensions, you reduce the cost in memory and processing goes down to 64 read/write operations and 124 flops per pixel. These results are shown in the third row of Table 14-1 (page 118).

  The 1D convolution with a kernel of size 31 that requires reading 31 input values for each output pixel, then performing 1 addition and 1 multiplication for each input. That's 31 + 1 I/O and 2 times 31 = 62 flops. Double this to get the numbers for the two passes. (This is specific to sigma=5.)

Figure 14-8 Separable two-dimensional convolution



- Recursive Gaussian Filter

This algorithm does not compute the exact Gaussian blur, only a good approximation of it. As shown in Figure 14-9 (page 117), it requires four passes (two horizontal, two vertical), but reduces processing to 10 read/write operations and 64 flops per pixel. These results are shown in the fourth row of Table 14-1 (page 118).

Figure 14-9 Recursive Gaussian filter passes

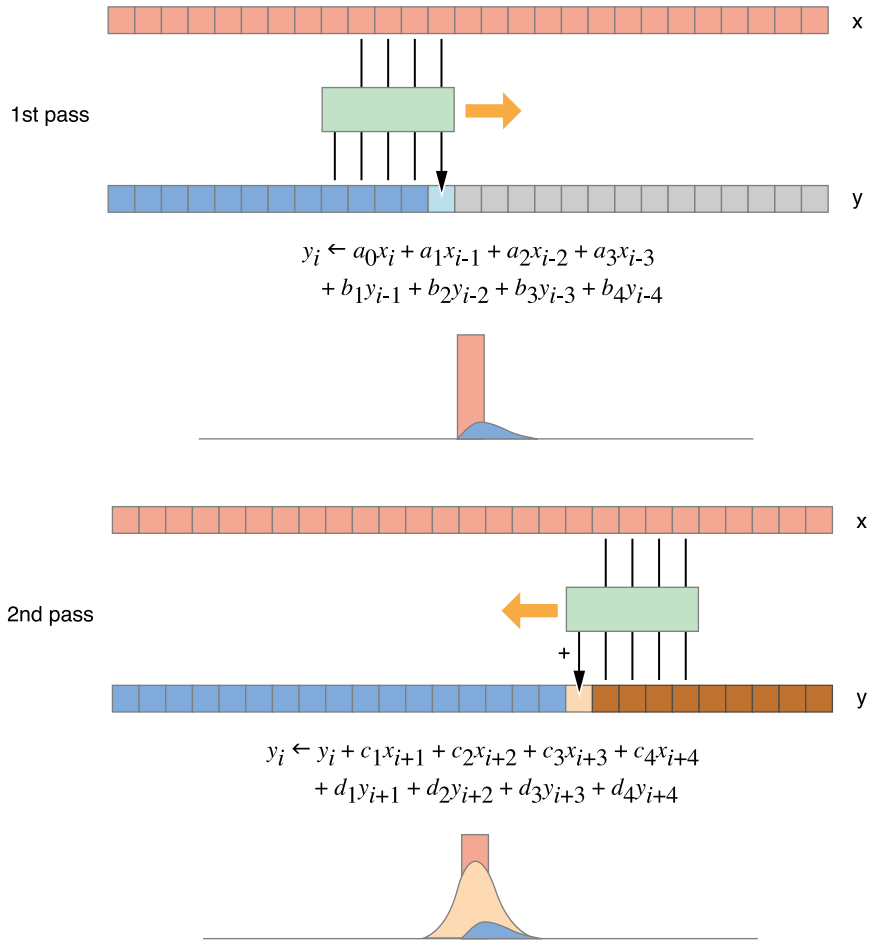


Figure 14-10 Recursive Gaussian filter

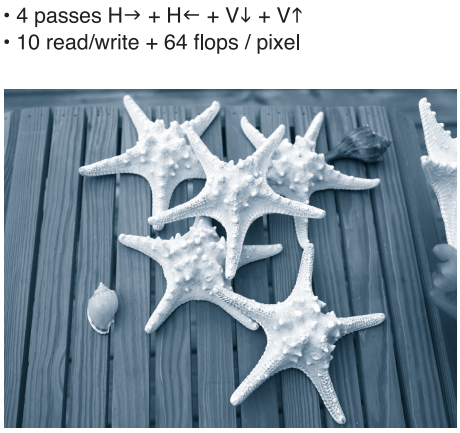


Table 14-1 (page 118) compares the compute-to-memory ratio results of the 2D Convolution, Separable Convolution, and Recursive Gaussian iterations. (The top row shows the results of a simple copy.) It looks like Recursive Gaussian algorithm performs best:

Table 14-1 Comparing algorithms

Algorithm	Memory (float R+W)	Compute (flops)	C/M Ratio	Estimate (MP/s)
Copy	2	0	0	14,200
2D Convolution	962	1,922	2	30
Separable Convolution	64	124	2	443
Recursive Gaussian	10	64	6	2,840

The first column depicts the number of memory accesses per pixel. The second column depicts the number of flops per pixel. The third column depicts the compute:memory ratio. The last column shows the number of megapixels each algorithm can be expected to process per second; numbers were obtained by taking the ratio of I/O with respect to the copy kernel. The copy kernel processes 14,200 MP/s with 2 I/O per pixel. A kernel with 64 I/O per pixel will be 32 times slower, so it will process 14200/32 = 443 MP/s.

4. The first version of code that performs the Gaussian blur using the recursive Gaussian algorithm looks like Listing 14-4 (page 119).

Listing 14-4 Recursive Gaussian implementation, version 1

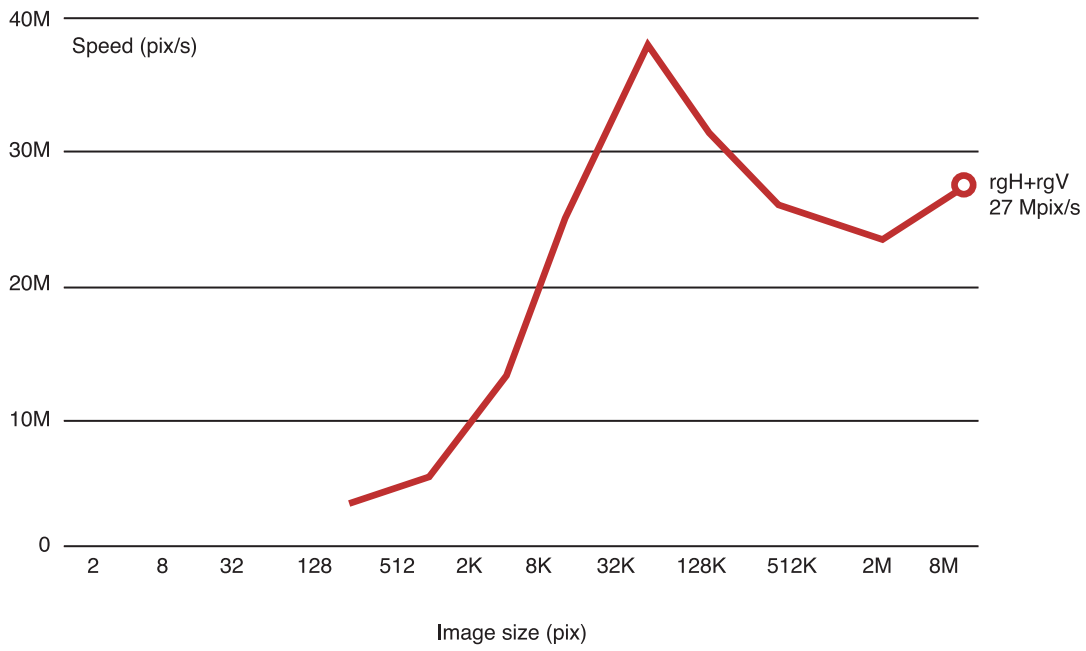
```
// This is the horizontal pass.
// One work item per output row
// Run one of these functions for each row of the image
// (identified by variable y).
kernel void rgH(global const float * in,global float * out,int w,int h)
{
  int y = get_global_id(0); // Row to process
  // Forward pass
  float i1,i2,i3,o1,o2,o3,o4;
  i1 = i2 = i3 = o1 = o2 = o3 = o4 = 0.0f;

  // In each iteration of the loop, read one input value and
  // store one output value.
  for (int x=0;x<w;x++)
  {
    float i0 = in[x+y*w]; // Load
    float o0 = a0*i0 + a1*i1 + a2*i2 + a3*i3
               - c1*o1 - c2*o2 - c3*o3 - c4*o4; // Compute new output
    out[x+y*w] = o0; // Store
    // Rotate values for next pixel.
    i3 = i2; i2 = i1; i1 = i0;
    o4 = o3; o3 = o2; o2 = o1; o1 = o0;
  }
  // Backward pass
  ...
}
```

```
// This is the vertical pass.
// One work item per output column
// Run one of these functions for each column of the image
//   (identified by variable x).
kernel void rgV(global const float * in,global float * out,int w,int h)
{
```

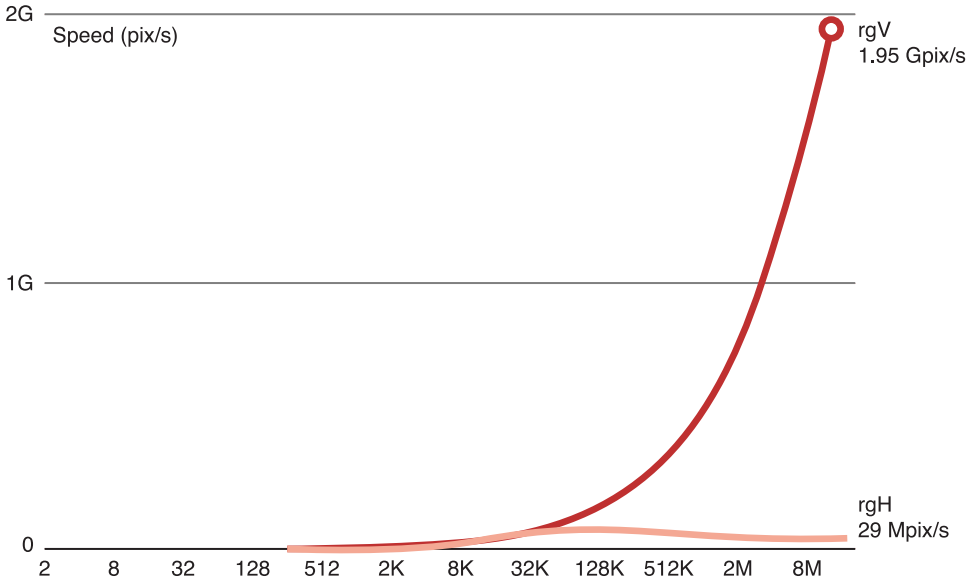
```
  int x = get_global_id(0); // Column to process
  // Forward pass
  float i1,i2,i3,o1,o2,o3,o4;
  i1 = i2 = i3 = o1 = o2 = o3 = o4 = 0.0f;
  for (int y=0;y<h;y++)
  {
    float i0 = in[x+y*w]; // Load
    float o0 = a0*i0 + a1*i1 + a2*i2 + a3*i3
        - c1*o1 - c2*o2 - c3*o3 - c4*o4;
    out[x+y*w] = o0; // Store
    // Rotate values for next pixel
    i3 = i2; i2 = i1; i1 = i0;
    o4 = o3; o3 = o2; o2 = o1; o1 = o0;
  }
  // Backward pass
  ...
}
```

This iteration produces results like those shown in Figure 14-11 (page 120).

Figure 14-11 Benchmark of Recursive Gaussian implementation, version 1



The vertical pass is fast, but the horizontal pass is not:

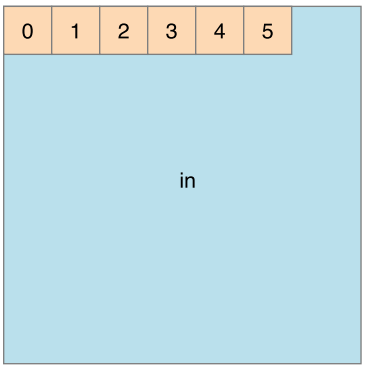


The problem is that inside the GPU we have scheduled about 16 million functions to be called in groups of about 300 work items at the same time, each simultaneously requesting a memory access with a different address. This is an example of a **memory access pattern**. The GPU hardware is optimized for certain kinds of memory accesses. Other kinds of accesses are conflicting. These will be serialized and will run much slower.

Specifically, in image processing, when consecutive work items access consecutive pixels in the same row, as in Figure 14-12 (page 121), processing is very fast:

Figure 14-12 Consecutive work items accessing consecutive addresses

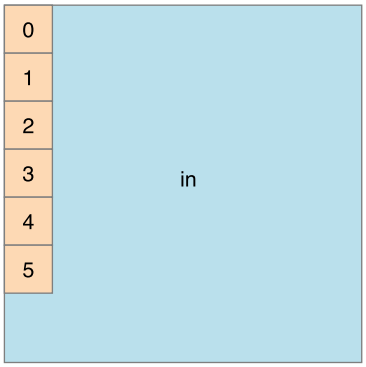
```
global float * in;
int i = get_global_id (0);
float v = in [i];   // FAST
```



However, in cases where memory accesses end up in the same bank, as in Figure 14-13 (page 122) (in image processing this is where consecutive work items access consecutive pixels in the same column) processing is slow:

Figure 14-13 Where memory accesses end up in the same bank, processing is slow

```
global float * in;
int i = get_global_id (0);
float v = in [1024*i];   // SLOW
```

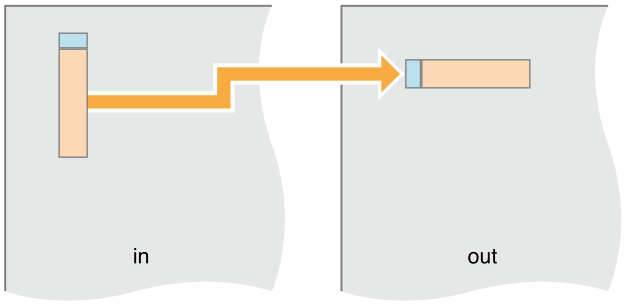


The solution is to transpose the array so that what was horizontal becomes vertical. We can process the transposed image, then transpose the result back into the proper orientation:

```
rgV + transpose + rgV + transpose = rgV + rgH
```

To transpose, we copy the pixels being transposed:

Figure 14-14 A transpose is really a copy



The transpose should be almost as fast as the copy kernel. However, although access to the input buffer is fast, access to the output buffer is slower:

```
kernel void transposeG(global const float * in,
                       global float * out,
                       int w,int h)
{
  int x = get_global_id (0);
  int y = get_global_id (1);

  out[y+x*h] = in[x+y*w];   // (x,y) in input = (y,x) in output
}
```

Slow Fast

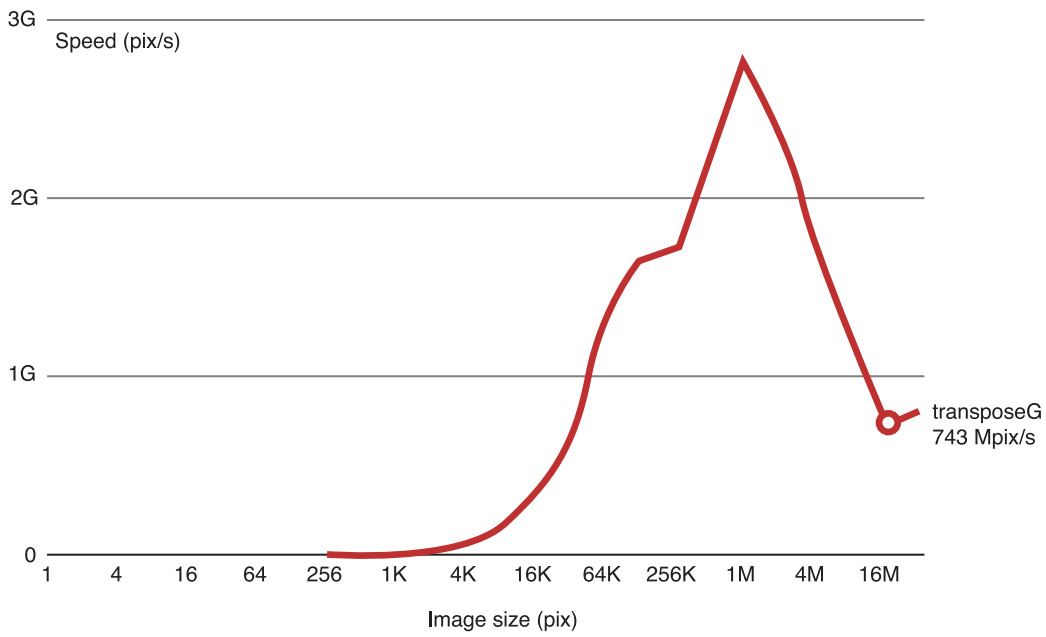
We estimate the performance of the transpose kernel by adding two I/O operations for the transpose for each pass. That comes to 10 + 2 * 2 = 14.

Table 14-2 Estimated results of transpose kernel

Algorithm	Memory (float R+W)	Compute (flops)	C/M Ratio	Estimate (MP/s)
V+T+V+T	14	64	4.6	2,030

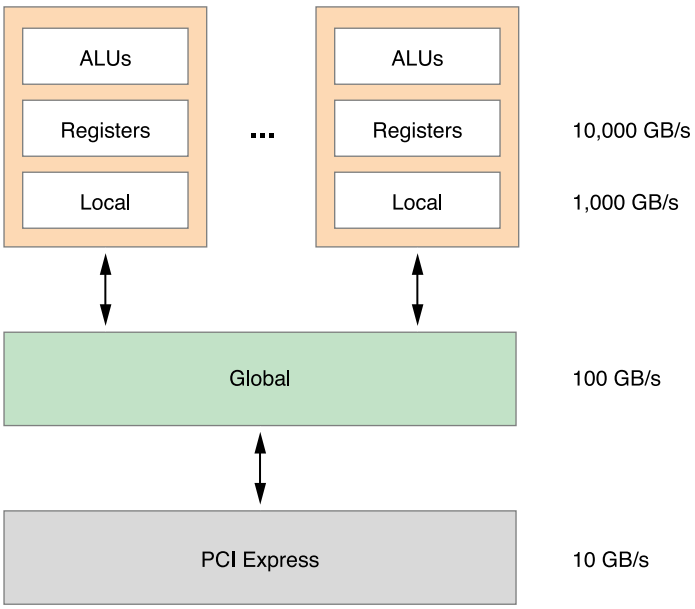
When we run the code, we see that as the image height gets larger, processing gets slower:

Figure 14-15 Results of benchmarking the transpose kernel



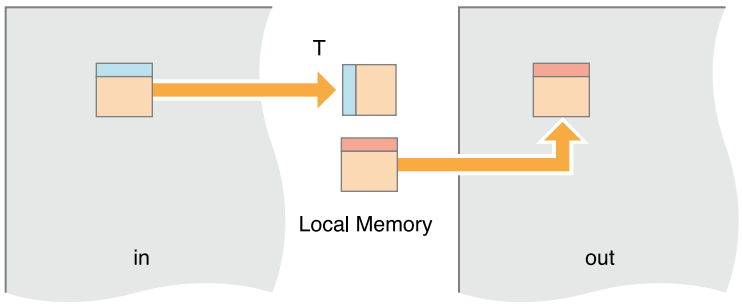
To speed this up, we can move the processing to faster memory. Inside the GPU are **processing cores** (the top boxes in Figure 14-16 (page 124)). Each GPU processing core has Arithmetic Logic Units (ALUs), registers, and local memory. The processing core is connected to the global memory. The global memory is connected to the host. Each layer of memory is about ten times faster than the one below it.

Figure 14-16 GPU memory hierarchy



In this iteration, we will move processing to the local memory. We'll have a work group—a block of work items—loading a small block of the image, storing it in local memory, then when all the work items in the group are finished performing the Gaussian recursion on the pixels in local memory, we move all of them out to the output buffer.

Figure 14-17 Moving blocks of the image to local memory



The code to do this looks like Listing 14-5 (page 124):

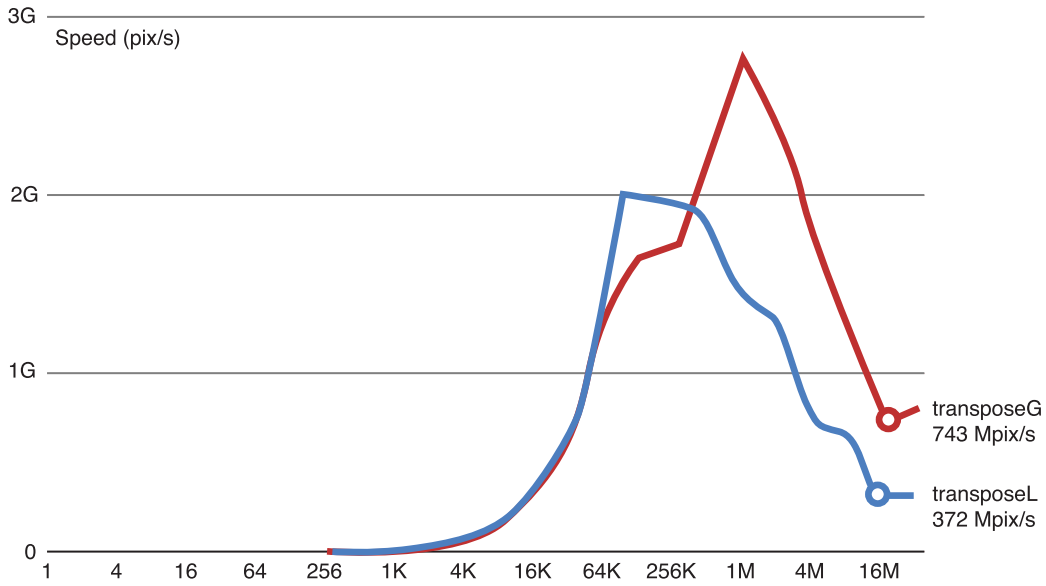
Listing 14-5 Move the work items to local memory then transpose

```
kernel void transposeL(global const float * in,
```

```
                                global float * out,
                                int w,int h)
{
  local float aux[256];           // Block size is 16x16

  // bx and by are the workgroup coordinates.
  // They are mapped to bx and by blocks in the image.
  int bx = get_group_id(0),       // (bx,by) = input block
      by = get_group_id(1);

  // ix and iy are the pixel coordinates inside the block.
  int ix = get_local_id(0),       // (ix,iy) = pixel in block
      iy = get_local_id(1);
  in += (bx*16)+(by*16)*w;        // Move to origin of in,out blocks
  out += (by*16)+(bx*16)*h;

  // Each work item loads one value to the temporary local memory,
  aux[iy+ix*16] = in[ix+w*iy];    // Read block

  // Wait for all work items.
  // This barrier is needed to make sure all work items in the workgroup
  // have executed the aux[…] = in[…] instruction, and that all values
  // in aux are correct. Then we can proceed with the out[…] = aux[…].
  // This is needed because each work item will set one value of aux
  // and then read another one, which was set by another item.
  // If we don't synchronize at this point, we may read an aux value that
  // has not yet been set.
  barrier(CLK_LOCAL_MEM_FENCE);    // Synchronize

  // Move the value from the local memory back out to global memory.
  // Because copying to consecutive memory, the writes are fast.
  out[ix+h*iy] = aux[ix+iy*16];   // Write block
}
```

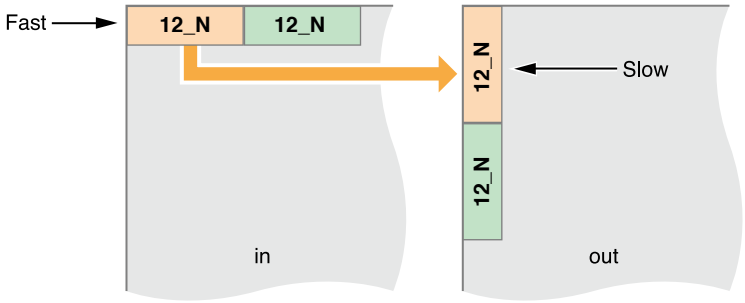
Unfortunately, this change did not make the code run faster.

Figure 14-18 Results of moving the work to local memory and then transposing.



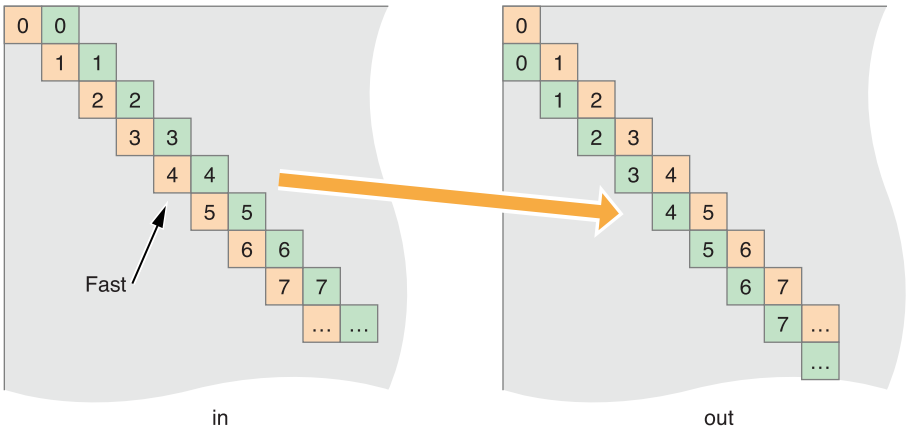
The problem is that now we have created another memory access pattern when we copy the results from rows in local memory to columns in output (global) memory.

Figure 14-19 Memory Access Pattern now occurs on the output side



To solve this, change the work groups to map pixels to copy diagonally:

Figure 14-20 Skew block mapping



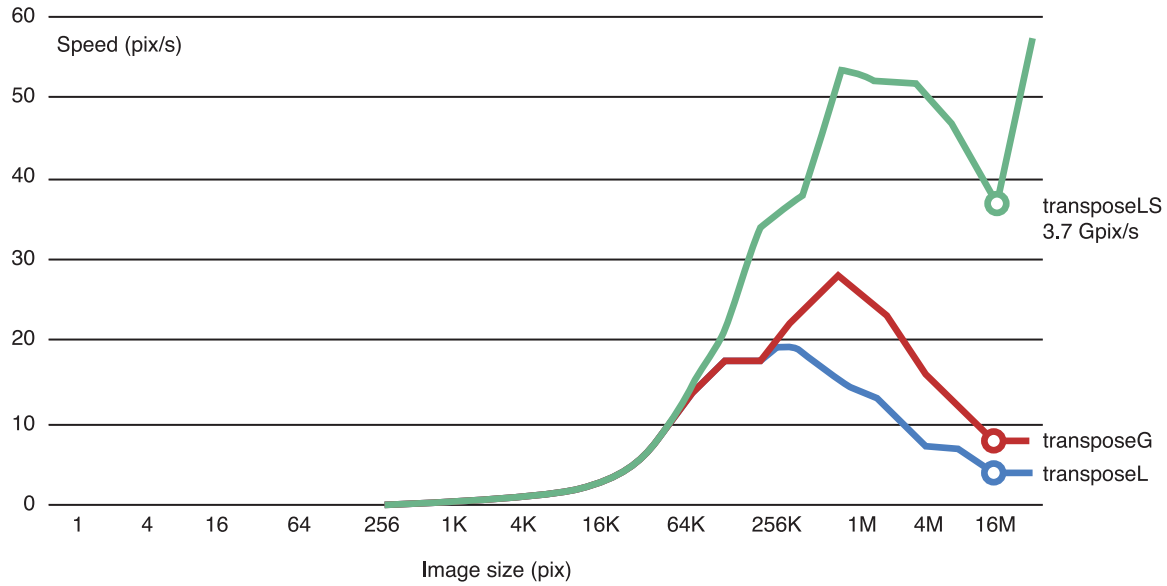
To convert the code to skew the input and output copy, just change one line:

Listing 14-6 Change the code to move diagonally through the image

```
kernel void transposeLS(global const float * in,
                        global float * out,
                        int w,int h)
{
  local float aux[256];           // Block size is 16x16
  int bx = get_group_id(0),       // (bx,by) = input block
  by = get_group_id(1);
  int ix = get_local_id(0),       // (ix,iy) = pixel in block
  iy = get_local_id(1);
  // This is the line we changed:
  by = (by+bx)%get_num_groups(1); // Skew mapping

  in += (bx*16)+(by*16)*w; // Move to origin of in,out blocks
  out += (by*16)+(bx*16)*h;
  aux[iy+ix*16] = in[ix+w*iy];    // Read block
  barrier(CLK_LOCAL_MEM_FENCE);   // Synchronize
  out[ix+h*iy] = aux[ix+iy*16];   // Write block
}
```

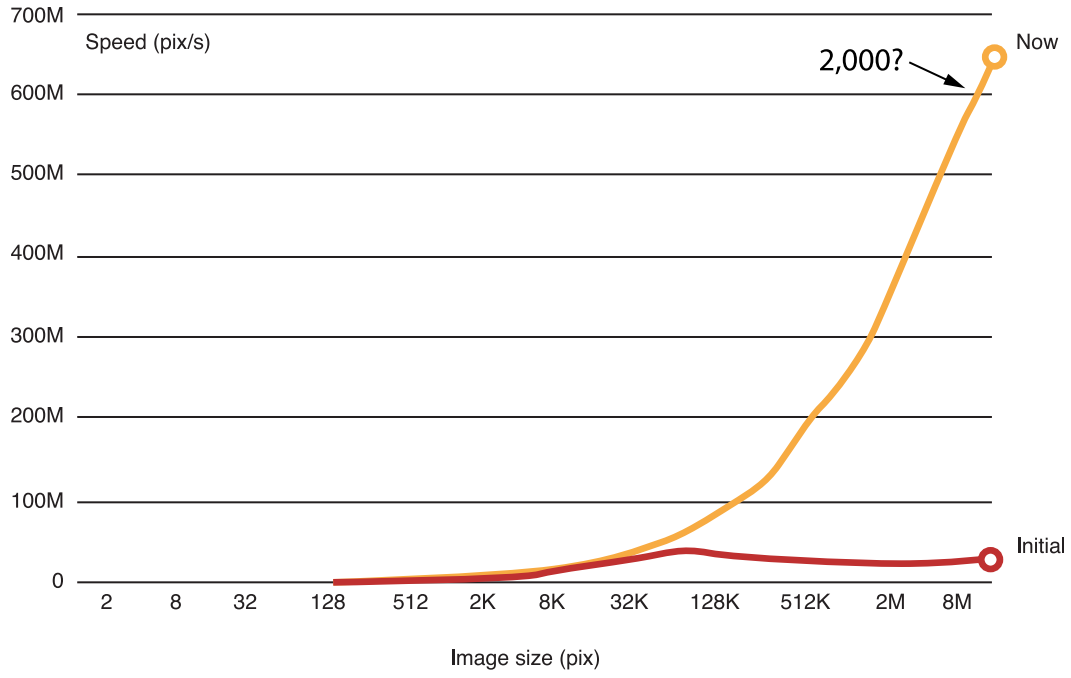
Benchmarking proves that this version is faster:

Figure 14-21 Benchmark of the skewed code



Running the transposed code in local memory does make the Gaussian blur significantly faster:

Figure 14-22 Benchmark of the transposed, skewed code



Still, processing is not occuring as quickly as our original speed estimate would indicate. The problem is that because of the sequential nature of the recursive Gaussian loop, we don't have enough work groups to saturate the GPU. We would need to change the algorithm to increase the parallelism level in order to increase performance to meet our original estimate.

Guidelines For Reducing Overhead On the GPU

Some general principles for improving the efficiency of your OpenCL code running on a GPU:

- Building an OpenCL program is computationally expensive and should ideally occur only once in a process.

  Be sure to take advantage of tools in OS X v10.7 or later that allow you to compile once and then run many times. If you do choose to compile a kernel during runtime, you will need to execute that kernel many times to amortize the cost of compiling it. You can save the binary after the first time the program is run and reuse the compiled code on subsequent invocations, but be prepared to recompile the kernel if the build fails because of an OpenCL revision or a change in the hardware of the host machine.

  You can also use bitcode generated by the OpenCL compiler instead of source code. Using compiled bitcode will increase processing speed and alleviates the need for you to ship source code with your application.
- Use OpenCL built-in functions whenever possible. Optimal code will be generated for these functions.
- Balance precision and speed. GPUs are designed for graphics, where the requirements for precision are lower. The fastest variants are exposed in the OpenCL built-ins as `fast_`, `half_`, `native_` functions. The program build options allow control of some speed optimizations.
- Avoid divergent execution. On the GPU, all threads scheduled together must execute the same code. As a consequence, when executing a conditional, all threads execute both branches, with their output disabled when they are in the wrong branch. It is best to avoid conditionals (replace them with `a?x:y` operators) or use built-in functions.
- Try to use image objects instead of buffers. In some cases (for certain memory access patterns), the different hardware data path the GPU uses when accessing images may be faster than if you use buffers. Using images rather than buffers is especially important when you use 16-bit floating-point data (`half`).

Improving Performance On the CPU

When optimizing code to run for a GPU or a CPU, it is important to take into consideration the strengths and limitations of the device you are writing for. This chapter focuses on optimizing for the a CPU. CPUs have fewer processing elements and more memory (both a large cache and a much larger amount of RAM) than GPUs, which have more processing elements and comparatively less memory. CPU memory access is fastest when data is in cache.

The chapter describes how to benchmark the speed of OpenCL code running on a CPU and how to set performance objectives. It provides tips for writing efficient OpenCL code. It also provides an example of an iterative process in which performance of a simple image filter application is tuned for best performance on a CPU.

Important: Before you manually optimize code to run on CPUs, try the autovectorizer. The autovectorizer frees you to write simple scalar code. It then vectorizes that code for you so that performance on the CPU is maximized. See "Autovectorizer" (page 101).

See "Tuning Performance On the GPU" (page 105) for a description of how to optimize code that will run on GPUs.

Before Optimizing Code

Before you optimize code:

1. Decide whether the code really needs to be optimized. Optimization can take significant time and effort. Weigh the costs and benefits of optimization before starting any optimization effort.
2. Estimate optimal performance. Run some simple kernels on your CPU device to estimate its capabilities. You can use the techniques described in "Measuring Performance On Devices" (page 107) to measure how long kernel code takes to run. See "Generating the Compute/Memory Access Peak Benchmark" (page 108) for examples of code you can use to test memory access speed and processing speed.
3. Generate or collect sample data to feed through each iteration of optimization. Run the unoptimized original code through the sample code and save the results. Then run each major iteration of the optimized code against the same data and compare the results to the original results to ensure your output has not been corrupted by the changed code.

Reducing Overhead

Here are some general principles you can follow to improve the efficiency of OpenCL code intended to run on a CPU:

- Choose an efficient algorithm.

  OpenCL can take advantage of all the devices in the system, but only if the algorithms in your program are written to allow parallel processing.

  Consider the following when choosing an algorithm:

  - When sending work to a CPU, which typically has fewer cores than a GPU, it is important to match the number of work items to the number of threads the CPU can effectively support.
  - OpenCL is most efficient when working with large datasets. If possible, select an algorithm that works on large chunks of data or merge several smaller tasks into one.
- Building an OpenCL program is computationally expensive and should ideally occur only once in a process.

  Be sure to take advantage of tools in OS X v10.7 or later that allow you to compile once and then run many times. If you instead choose to compile a kernel during runtime, you will need to execute that kernel many times to amortize the cost of compiling it. You can save the binary after the first time the program is run and reuse the compiled code on subsequent invocations, but be prepared to recompile the kernel if the build fails because of an OpenCL revision or a change in the hardware of the host machine.

  You can also use bitcode generated by the OpenCL compiler instead of source code; if you do this, compilation speed will be much faster and you won't have to ship source code with your application.
- Allocating and freeing OpenCL resources (memory objects, kernels, and so on) takes time. Reuse these objects whenever possible instead of releasing them and recreating them repeatedly. Note, however, that image objects can be reused only if they are the same size and pixel format as needed by the new image.
- Experiment with your code to find the kernel size that works best.

  Using smaller kernels can be efficient because each tiny kernel uses minimal resources and breaking a job into many small kernels can allow for the creation of very large and efficient workgroups. On the other hand, starting each kernel does take between 10-100 µs. When each kernel exits, the the results must be stored in global memory. Because reading and writing to global memory is expensive, concatenating many small kernels into one large kernel may save considerable overhead.

  To determine the ideal kernel size for your application, experiment with your code to find the kernel size that provides optimal performance.
- Use OpenCL's built-in functions whenever possible.

  Optimal code is generated for these functions.
- Take advantage of the memory subsystem of the device.
  - When writing for the CPU, take advantage of the memory subsystem: reuse data while it's still in L1 or L2 cache. To achieve this, use loop blocking and access memory in a cache-friendly pattern.

- Avoid divergent execution.
  - The CPU predicts the result of conditional jump instructions (corresponding to `if`, `for`, `while`, and so on) and starts processing the selected branch before knowing the effective result of the test. If the prediction is wrong, the entire pipeline needs to be flushed, and you lose cycles. If possible, use conditional assignment instead.
- Write simple scalar code first. The compiler and the autovectorizer work best on scalar code and can generate near-optimal code with no effort required from you. If the autovectorizer provides sub-optimal results, add vectors to the code by hand.
- Use the `-cl-denorms-are-zero` option in clBuildProgram, unless you need to use denormals (denormals are very small numbers with a slightly different floating-point representation). Denormals handling can be extremely slow (100x slower) and can lead to puzzling benchmark results.
- CPUs are not optimized for graphics processing. Avoid using images. CPUs provide no hardware acceleration for images and image access is slower than the equivalent buffer access.
- CPU access to global memory is not as expensive as GPU access to global memory.

Estimating Optimal Performance

Before optimizing code, it is best to know what kind of performance is achievable. See "Measuring Performance On Devices" (page 107) for information about how to set a timer to measure the speed at which a kernel runs.

The main factor determining the execution speed of an OpenCL kernel is memory usage; this is the case for both CPU and GPU devices. Benchmarking the speed of the kernel function in Listing 15-1 (page 132) provides a way to estimate the memory speed of an OpenCL device.

Listing 15-1 Kernel for estimating optimal memory access speed

```
kernel void copyBuffer(global const float * in, global float * out) {
  int i = get_global_id(0);
  out[i] = in[i];  // R+W one float
}
```

Important: OpenCL becomes more efficient as data size increases. Try to process larger problems in fewer kernel calls.

The asymptotic (maximum) value memory speed can be used to estimate the speed of a memory-bound algorithm on large data.

Take the box average kernel before it has been optimized, shown in Listing 15-2 (page 133) for example. This kernel accepts a single channel floating point image as input and computes a single channel floating point image where each output pixel (x,y) is the average value of all pixels in a square box centered at (x,y). A `w` by `h` image is stored in a buffer `float * A`, where pixel (x,y) is stored in `A[x+w*y]`.

Listing 15-2 The boxAvg kernel, first version

```
constant int RANGE = 2;
kernel void boxAvg1(int w, int h, global const float * in, global float * out) {
  int x = get_global_id(0);  // Pixel to process is (x,y)
  int y = get_global_id(1);
  float sumA = 0.0f;         // Sum of pixel values
  float sum1 = 0.0f;         // Number of pixels
  for (int dy=-RANGE;dy<=RANGE;dy++)
    for (int dx=-RANGE;dx<=RANGE;dx++) {
      int xx = x + dx;
      int yy = y + dy;
      // Accumulate if inside image
      if (xx>=0 && xx<w && yy>=0 && yy<h) { sumA += in[xx + w*yy]; sum1 += 1.0f;
}
    }
  out[x+w*y] = sumA/sum1;
}
```

For each pixel, we compute the average of 25 input values (RANGE=2). That's 26*4=104 bytes per pixel. The maximum speed of this kernel should be near (36e9 B/s) / (104 B/pix) = 346 MP/s on the GPU, and 12e9/104 = 115 MP/s for the CPU. The actual numbers are 129 MP/s and 103 MP/s respectively.

Important: Use memory benchmarks to estimate the speed of kernels.

The following two sections show how to tune the code to compute boxAvg on a CPU. CPU and GPU tuning have a lot in common, but at some point the optimization techniques differ, and reaching the best performance usually requires two versions of the kernel—one for the CPU and one for the GPU.

Tuning OpenCL Code For the CPU

To make the best use of the full computing and memory potential of a modern CPU:

- Try the autovectorizer first.

  The autovectorizer can transform scalar OpenCL code into suitable vector code automatically. See "Autovectorizer" (page 101). The autovectorizer works on a restricted class of code, and in some cases you may find it necessary to vectorize the code by hand. But the first step is to try scalar code and let the autovectorizer vectorize for you. Vectorize manually only if really needed.

  > **Important:** Write simple scalar code first. The compiler and the autovectorizer work better on scalar kernels and can generate near-optimal code with no additional effort required by you.
  >
  > Use OpenCL's built-in functions whenever possible. The autovectorizer generates optimal code for these functions.

- Use all cores of the processor.

  OpenCL automatically ensures that all processor cores are used. Work items are scheduled in different tasks submitted to Grand Central Dispatch (GCD) and then executed on all available CPU cores. All that's needed is to have enough work items to run on all threads.

- Use the whole width (16 bytes for SSE or 32 bytes for AVX) of the SIMD execution units.

- Use the memory cache hierarchy efficiently.

  OpenCL execution speed on CPUs is related to the optimal usage of the various levels of data cache. Tuning the code to match the cache levels requires effort, but may provide substantial acceleration.

  Each core of the CPU (the values shown correspond to the Intel Core i7 CPU found in our test machine) has a bank of registers (typically a few hundreds of bytes, 1 cycle of latency), an L1 data cache (32 KB, 3 cycles), and an L2 cache (256 KB, 12 cycles). All cores of the CPU share a large L3 cache (6 MB, 40 cycles). External memory access has a latency in the ~250 cycles range. Each core has several hardware prefetcher units able to identify regular data streams, and move the data closer to the core before it is needed (prefetch).

  Efficient OpenCL kernels should take advantage of this architecture by adopting regular memory access patterns and reusing the data before it is evicted from the cache. This is usually done through the use of several levels of loops (blocking) matching the various cache levels. The OpenCL framework, executing work items sequentially on several threads, provides the highest loop level.

  > **Important:** When using the same data multiple times, try to keep it in the L1 or L2 cache. The cache lines accessed least recently are dropped first. To keep data in the L1/L2 caches, use it soon after its last use.

Example: Tuning a Kernel To Optimize Performance On a CPU

In the example that follows, we tune our sample boxAvg kernel in several ways and check how each modification affects performance on the CPU:

1. We divide the computation into two passes. See "Putting the Horizontal and Vertical Passes Together" (page 143).
2. We modify the horizontal pass to compute one *row* per work item instead of one single *pixel*. See "Putting the Horizontal and Vertical Passes Together" (page 143).
3. We modify the algorithm to read fewer values per pixel and to incrementally update the sum rather than computing it each time. See Table 15-2 (page 137).
4. We modify the horizontal pass by moving the division and conditionals out of the inner loop. See Table 15-3 (page 138).
5. We modify the vertical pass to combine rows; each work item computes a block of row. See "Optimizing the Vertical Pass" (page 141).
6. We ensure that the image width (`w`) is a multiple of four so we can use faster 16-byte I/O functions on float4 data. See Listing 15-7 (page 141).
7. We ensure that the code works for any image width. See "Example: Tuning a Kernel To Optimize Performance On a CPU" (page 135).
8. Fuse the kernels. See "Putting the Horizontal and Vertical Passes Together" (page 143).

Dividing Kernel Computation Into Two Passes

The computational work of the boxAvg kernel can be broken into two passes. The first pass will compute the horizontal average of the input pixels; the second pass will compute a vertical average of the horizontal averages:

Listing 15-3 The boxAvg kernel in two passes

```
// Horizontal pass v1. Global work size: w x h
kernel void boxAvgH1(int w,int h,global const float * in,global float * out) {
  int x = get_global_id(0);  // Pixel to process is (x,y)
  int y = get_global_id(1);
  float sumA = 0.0f;         // Sum of pixel values
  float sum1 = 0.0f;         // Number of pixels
  for (int dx=-RANGE;dx<=RANGE;dx++) {
    int xx = x + dx;
    // Accumulate if inside image
```

```
    if (xx>=0 && xx<w) { sumA += in[xx+w*y]; sum1 += 1.0f; }
  }
  out[x+w*y] = sumA/sum1;
}

// Vertical pass v1. Global work size: w x h
kernel void boxAvgV1(int w,int h,global const float * in,global float * out) {
  int x = get_global_id(0);  // pixel to process is (x,y)
  int y = get_global_id(1);
  float sumA = 0.0f;         // sum of pixel values
  float sum1 = 0.0f;         // number of pixels
  for (int dy=-RANGE;dy<=RANGE;dy++) {
    int yy = y + dy;
    // Accumulate if inside image
    if (yy>=0 && yy<h) { sumA += in[x + w*yy]; sum1 += 1.0f; }
  }
  out[x+w*y] = sumA/sum1;
}
```

In both cases, memory transfers take 24 B/pix (5 read + 1 write), and the estimated speed of each pass is 500 MP/s for the CPU. Actual values are shown in the right column of Table 15-1 (page 136):

Table 15-1 Comparing estimated and actual memory transfer speeds

Kernel	Estimate	Actual
boxAvgH1	500 MP/s	523 MP/s
boxAvgV1	500 MP/s	563 MP/s

We can see some effects of the cache hierarchy. The copyBuffer value of 12 GB/s corresponds to external memory speed. In the case of boxAvg, we reuse each input value five times, and the data is found in the cache, giving speeds better than our estimate based on external memory speed only.

We can compute an absolute upper bound: we need to load each input value once from external memory, and store each output value once, and we assume that the subsequent input accesses are in cache and instantaneous. That's only 8 B/pix, giving an upper bound of 1,500 MP/s (the speed of image copy).

Optimizing the Horizontal Pass

To take better advantage of the cache, we can have each work item process several pixels instead of only one pixel. Since we need only a few threads to saturate the CPU (this is not the case for GPUs), we can actually have only one work item per row (for the horizontal pass) or per column (for the vertical pass), like this:

Listing 15-4 Modify the horizontal pass to compute one row per work item instead of one pixel

```
// Horizontal pass v2. Global work size: H
kernel void boxAvgH2(int w,int h,global const float * in,global float * out)
{
  int y = get_global_id(0);       // Row to process is y
  // Process the row, loop on all pixels
  for (int x=0; x<w; x++) {
    float sumA = 0.0f;            // Sum of pixel values
    float sum1 = 0.0f;            // Number of pixels
    for (int dx=-RANGE;dx<=RANGE;dx++) {
      int xx = x + dx;
      // Accumulate if inside image
      if (xx>=0 && xx<w) { sumA += in[xx+w*y]; sum1 += 1.0f; }
    }
    out[x+w*y] = sumA/sum1;      // Store output value (x,y)
  }
}
```

As shown in Table 15-2 (page 137), for the horizontal pass we see an increased benefit from using fast cache memory. The input values used in one iteration of the x loop are reused immediately in the next iteration, and will be in L1 or L2 cache. For the vertical pass, on the other hand, we now have different threads processing different columns at the same time. Each iteration of the y loop now accesses a different cache line. This *slows* execution speed.

Table 15-2 Comparing optimal and actual speeds of work item row and column processing

Kernel	Upper Bound	Actual
boxAvgH2	1500 MP/s	931 MP/s
boxAvgV2	1500 MP/s	456 MP/s

In this version of the code, the autovectorizer can be of real help. Let's look at the execution times for different workgroup sizes, corresponding more or less to the block size used to vectorize the code. More precisely, the kernel code is vectorized up to the actual vector size (`float4` for SSE, or `float8` for AVX), and then when the workgroup size increases, several vector variants are executed together.

For example, Table 15-3 (page 138) shows that with a workgroup size of 4, we execute `float4` variants of the kernel, merging together 4 work items, and with a workgroup size of 32 we would execute 8 of these `float4` variants.

Table 15-3 Effect of work group size on execution time

Workgroup size	Execution time (ms)
1	260
2	143
4	37
8	45
16	56
32	66
64	66
128	67

The best performance was achieved using the float4 vectorized variant; it is 7x faster than the scalar variant.

In addition to giving vector instructions more bandwidth, the autovectorizer virtually adds blocking (or tiling) to the outer loop (the scheduling loop in the framework) and it can be used to better match the CPU cache structure.

We can make the horizontal kernel better. Instead of computing the sum of `2*RANGE+1` values at each iteration of the x loop, we can simply update the sum between two consecutive iterations, as illustrated in Listing 15-5 (page 138):

Listing 15-5 Modify the algorithm to read fewer values per pixel and to incrementally update the sum

```
// Horizontal pass v3. Global work size: H
kernel void boxAvgH3(int w,int h,global const float * in,global float * out) {
```

```
  int y = get_global_id(0);                                 // Row to process is y
  global const float * inRow = in + y*w;                    // Beginning of input row
  float sumA = 0.0f;                                       // Sum of values in moving
 segment
  float sum1 = 0.0f;                                        // Number of values in
moving segment
  // Initialize the first sums with segment 0..RANGE-1
  for (int x=0; x<RANGE; x++) { sumA += inRow[x]; sum1 += 1.0f; }

  for (int x=0; x<w; x++) {
    // Here, sums are set to segment x-RANGE-1..x+RANGE-1
    // update them to x-RANGE..x+RANGE.
    // Remove x-RANGE-1.
    if (x-RANGE-1 >= 0) { sumA -= inRow[x-RANGE-1]; sum1 -= 1.0f; }
    if (x+RANGE < w) { sumA += inRow[x+RANGE]; sum1 += 1.0f; } // insert x+RANGE
    // Store current value
    out[x+w*y] = sumA/sum1;
  }
}
```

This variant reduces the number of memory accesses from 6 to 3 per iteration of the x loop. The execution speed of this variant is now 1366 MP/s (and only 822 MP/s without the autovectorizer). This is 91% of our upper bound.

We can move the conditionals and the division out of the x loop by splitting it into three parts:

Listing 15-6 Modify the horizontal pass by moving division and conditionals out of the inner loop

```
// Horizontal pass v4. Global work size: H
kernel void boxAvgH4(int w, int h, global const float * in, global float * out) {
  int y = get_global_id(0);                                 // Row to process is y
  global const float * inRow = in + y*w;                    // Beginning of input row
  global float * outRow = out + y*w;                        // Beginning of output row
  float sumA = 0.0f;
  float sum1 = 0.0f;

  // Left border
```

```
  int x = -RANGE;
  for (; x<=RANGE; x++) {
    // Here, sumA corresponds to segment 0..x+RANGE-1, update to 0..x+RANGE.
    sumA += inRow[x+RANGE]; sum1 += 1.0f;
    if (x >= 0) outRow[x] = sumA/sum1;
  }
  // x is RANGE+1 here

  // Internal pixels
  float k = 1.0f/(float)(2*RANGE+1);                  // Constant weight for
internal pixels
  for (; x+RANGE<w; x++) {
    // Here, sumA corresponds to segment x-RANGE-1..x+RANGE-1,
    // Update to x-RANGE..x+RANGE.
    sumA -= inRow[x-RANGE-1];
    sumA += inRow[x+RANGE];
    outRow[x] = sumA * k;
  }
  // x is w-RANGE here

  // Right border
  for (; x < w; x++) {
    // Here, sumA corresponds to segment x-RANGE-1..w-1, update to x-RANGE..w-1.
    sumA -= inRow[x - RANGE - 1]; sum1 -= 1.0f;
    outRow[x] = sumA/sum1;
  }
  // x is w here, and we are done
}
```

This variant runs at 1457 MP/s, or 97% of the upper bound.

Important: Avoid conditionals and high latency operations in the most intensive parts of the code. If possible, try to move all conditional statements out of inner loops.

Optimizing the Vertical Pass

Now we optimize the vertical kernel. To make better use of the cache, we can combine entire rows together. Each work item will be responsible for computing one block of rows. The code in Listing 15-7 (page 141) enqueues a small number of work items: one per CPU thread, which is enough.

Listing 15-7 Modify vertical pass to combine rows; each work item computes a block of rows

```
// Vertical pass v3. Global work size: >= number of CPU cores.
kernel void boxAvgV3(int w,int h,global const float * in,global float * out) {
  // Number of rows to process in each work item (rounded up)
  int rowsPerItem = (h+get_global_size(0)-1)/get_global_size(0);
  int y0 = rowsPerItem * get_global_id(0);         // Update the range Y0..Y1-1
  int y1 = min(h, y0 + rowsPerItem);
  for (int y=y0; y<y1; y++) {
    // Accumulate into row y all rows in y-RANGE..y+RANGE intersected with 0..h-1
    int ya0 = max(0, y-RANGE);
    int ya1 = min(h, y+RANGE+1);
    float k = 1.0f/(float)(ya1-ya0);              // 1/(number of rows)
    global float * outRow = out + w*y;            // Output row
    for (int x=0; x<w; x++) outRow[x] = 0.0f;     // Zero output row
    for (int ya=ya0; ya<ya1; ya++) {
      global const float * inRow = in + w*ya;     // Input row
      for (int x=0; x<w; x++) outRow[x] += k * inRow[x];
    }
  }
}
```

This kernel is executed at 891 MP/s on our test machine. If we can ensure that the image width is always a multiple of four, we can vectorize all x loops to `float4`:

Listing 15-8 Ensure the image width is always a multiple of 4

```
...
```

```
    global float4 * outRow = (global float4 *)(out + w*y);          // Output row
    for (int x=0; x<w/4; x++) outRow[x] = 0.0f;
      for (int ya=ya0; ya<ya1; ya++) {
        global const float4 * inRow = (global float4 *)(in + w*ya); // Input row
        for (int x=0; x<w/4; x++) outRow[x] += k * inRow[x];
      }
    …
```

This code runs at 1442 MP/s, 96% of the upper bound. When accessing data through a pointer to `float4`, the compiler assumes the pointer is aligned to multiples of the size of the type, here 16 bytes, and generates an efficient aligned move instruction (`movaps`).

Important: Make sure the inner loops are vectorized (as much as possible) using the autovectorizer or by hand.

This code is not safe, since we can only assume in and out are four-byte aligned (the size of `float`). A safer variant that will work for any image width would look something like:

Listing 15-9 A safer variant that will work for any image width

```
// Zero output row using aligned memory access
global float * outRow = out + w*y;
int x = 0;
// Iterate by 1 until 16-byte aligned
for (; x<w && ((intptr_t)&outRow[x] & 15); x++) outRow[x] = 0.0f;
// Iterate by 4 on aligned address while we can
// outRow4 is 16-byte aligned
global float4 * outRow4 = (global float4 * outRow4)(outRow + x);
for (; x+4<w; x+=4,outRow4++) outRow4[0] = 0.0f;
// Iterate by 1 until w is reached
for (; x<w; x++) outRow[x] = 0.0f;
```

We now have near-optimal versions of both horizontal and vertical kernels. We can call them in sequence to compute the result, and run at a speed near 700 MP/s.

Putting the Horizontal and Vertical Passes Together

We can now fuse the two kernels together, and loop over the entire input image only once. This is illustrated in Listing 15-10 (page 143):

Listing 15-10 Fused kernel

```
// Fused H+V variant. Global work size: any, >= number of CPU cores.
// AUX[w*global_size(0)] is temporary storage, 1 row for each work item.
kernel void boxAvg2(int w, int h, global const float * in,
                        global float * out, global float * aux) {
  // Number of rows to process in each work item (rounded up)
  int rowsPerItem = (h+get_global_size(0)-1)/get_global_size(0);
  int y0 = rowsPerItem * get_global_id(0);          // Update the range Y0..Y1-1
  int y1 = y0 + rowsPerItem;
  aux += get_global_id(0) * w;                      // Point to our work item's
row of temporary storage
  float k = 1.0f/(float)(2*RANGE+1);               // Constant weight for internal
 pixels

  // Process our rows. We need to process extra RANGE rows before and after.
  for (int y=y0-RANGE; y<y1+RANGE; y++) {
    // Zero new row entering in our update scope before we start
    // accumulating values in it.
    if (y+RANGE < min(h, y1)) {
      global float4 * outRow4 = (global float4 *)(out + w*(y + RANGE));
      for (int x=0; x<(w/4); x++) outRow4[x] = 0.0f;
    }
    if (y<0 || y>=h) continue; // Out of range
    // Compute horizontal pass in AUX.



    //   The boxAvg4 code goes here on input row y
    //   The output is stored in AUX[W].
    // Accumulate this row on output rows Y-RANGE..Y+RANGE
    for (int dy=-RANGE; dy<=RANGE; dy++) {
      int yy = y + dy;
```

```
      if (yy < max(0, y0) || yy >= min(h, y1)) continue; // Out of range
      // Get number of rows accumulated in row YY, to get the weight
      int nr = 1 + min(h-1, yy+RANGE)-max(0, yy-RANGE);
      float u = 1.0f/(float)nr;
      // Accumulate AUX in row YY
      global float4 * outRow4 = (global float4 *)(out + w*yy);
      global float4 * aux4 = (global float4 *)(aux);
      for (int x=0; x<(w/4); x++) outRow4[x] += u * aux4[x];
    }
  }
}
```

This fused version runs at 1166 MP/s. Some rows will be processed twice, since we have to compute horizontal filters on rows y0-RANGE to y1+RANGE-1 to update output rows y0 to y1-1. At any given time during the execution, we will access one row in aux, one input row, and 2*RANGE+1 output rows. For a 4096x4096 image, each row is 16 KiB, and all 7 rows fit in L2 cache.

Important: Merging two kernels called one after the other can reduce memory accesses, and works on smaller data chunks fitting in faster cache levels, instead of of forcing the two kernels to resort to communicate via via full round trips to external memory.

Binary Compatibility Of OpenCL Kernels

Like other frameworks, OpenCL guarantees backwards binary compatibility for API calls. OpenCL, however, does not guarantee that OpenCL kernel binaries (binaries obtained by calling the `clGetProgramInfo(CL_PROGRAM_BINARIES)` function) will continue to work on future OpenCL revisions. In fact, these binaries are likely to break frequently, possibly with each new OpenCL minor and major release.

Handling Runtime Errors

If you plan to get kernel binaries with `clGetProgramInfo(CL_PROGRAM_BINARIES)` and save them to disk or other long term storage device for later reuse, you must handle the possibility that the `clBuildProgram` function will fail with error `CL_INVALID_BINARY` when you try to reuse the saved binary later. If this happens, your application should:

1. Create a new OpenCL program by calling the `clCreateProgramWithSource` function, passing the OpenCL C source code in the `strings` parameter.
2. Recompile the program using the `clBuildProgram` function.
3. Extract the new binary using `clGetProgramInfo(CL_PROGRAM_BINARIES)` and replace the old one on disk.

Avoiding Build Errors

Binary OpenCL kernel images saved to disk may also fail to build for the current machine if the image was created by another machine and saved to a network volume, if the user installs new hardware such as an additional GPU or replaces existing hardware, or changes some hardware settings (for example, turns on and off the discrete graphics card on MacBook Pro with both discrete and integrated GPUs).

The best way to avoid shipping binary OpenCL code is to use the offline OpenCL compiler to produce LLVM bitcode.

To generate LLVM IR (on OS X v10.8), compile a bitcode file for each of the architectures available: i368 32b, x86_64 bit, and 32b GPU. For each architecture, call the openclc compiler:

```
/System/Library/Frameworks/OpenCL.framework/Libraries/openclc -Os -arch i386
-emit-llvm-bc clFileName -o outputFileName

/System/Library/Frameworks/OpenCL.framework/Libraries/openclc -Os -arch x86_64
-emit-llvm-bc clFileName -o outputFileName

/System/Library/Frameworks/OpenCL.framework/Libraries/openclc -Os -arch gpu_32
-emit-llvm-bc clFileName -o outputFileName
```

The output file will be an LLVM bit-code object file, which can be used with the `clCreateProgramWithBinary` function.

Document Revision History

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

Date	Notes
2013-08-08	Created Tuning Performance On the GPU and Improving Performance On the CPU chapters. Other minor updates throughout.
2012-07-23	Updated Introduction and IOSurfaces chapters. Made minor corrections throughout.
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.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010