Introduction to GPU Computing with CUDA and OpenCL

James Shearer, University of California, Davis
Wei-Chao Chen, SDI Corporation and National Taiwan University

March 2, 2010
Contents

1 Introduction 3
  1.1 GPU Architecture Overview ........................................ 3
  1.2 About This Document ............................................. 4

2 CUDA 7
  2.1 The 10,000 Foot View ............................................ 7
    2.1.1 CUDA versus GPGPU ...................................... 7
    2.1.2 Scientific Computation .................................. 7
  2.2 A CUDA Crash Course ........................................... 8
  2.3 Tips on Optimization ............................................ 10
  2.4 Additional Resources ........................................... 10

3 OpenCL 11
  3.1 The 10,000 Foot View ........................................... 11
  3.2 The OpenCL Workflow ............................................ 12
  3.3 The Platform Layer .............................................. 13
  3.4 The Runtime ..................................................... 13
  3.5 The OpenCL Programming Language .............................. 14
  3.6 Other Sources .................................................. 14

4 OpenCL in Practice 16
  4.1 Maximum Luminance Filtering .................................. 16
  4.2 The CPU Implementation ....................................... 16
  4.3 OpenCL Conversion .............................................. 19
    4.3.1 OpenCL Setup ........................................... 20
    4.3.2 Memory Setup ........................................... 21
    4.3.3 Kernel setup ........................................... 22
    4.3.4 Kernel Execution ....................................... 23
    4.3.5 Performance ............................................ 24
  4.4 Image Object Upgrade .......................................... 24
    4.4.1 Kernel Changes .......................................... 25
    4.4.2 Memory Changes ......................................... 26
    4.4.3 Performance ............................................ 26
  4.5 Sharing with OpenGL ........................................... 27
    4.5.1 Setting the Stage ........................................ 27
    4.5.2 Context Creation ......................................... 27
    4.5.3 Memory Object Creation ................................. 28
    4.5.4 Kernel Execution ....................................... 28
    4.5.5 Performance ............................................ 29
    4.5.6 Bonus Round: VBO Sharing ................................ 29
Chapter 1

Introduction

Computer graphics hardware, during the past several decades, have greatly evolved and advanced in several fronts. In terms of performance, in addition to following the ubiquitous Moore’s law curve, their peak performance numbers are growing at a faster rate than mainstream CPUs owing to the natural parallelism of raster graphics algorithms. In addition, modern graphics systems are highly integrated processors, leading to the wide availability of low-cost graphics processing units (GPUs). Perhaps more importantly, the need to support various types of rendering effects and operations has given rise to customizable and, more recently, programmable shading pipelines where small shader programs can be executed for each individual graphics primitive or pixel.

Because GPUs are looking more like data-centric, massively-parallel streaming processors, many researchers have begun to take advantage of their processing capabilities by retrofitting algorithms to GPUs. This field, appropriately called General-Purpose computing on GPU (GPGPU), has grown considerably during the past several years. As expected, one of the biggest obstacles faced by GPGPU researchers is in transforming and mapping algorithms to GPU functionalities using graphics APIs such as OpenGL or DirectX. In addition to the unnecessary complexities involved in this process, it also became evident that GPUs lack some very useful hardware features for many parallel algorithms. For example, GPUs are designed to process one primitive independently of each other, making it extremely inefficient to share data between different instances of the same shader program.

Introduced in 2007, NVIDIA’s CUDA (Compute Unified Device Architecture) is one of the first widely-available software environment that enables developers to harness the computational resources in GPUs through high-level programming languages. It is designed in conjunction with hardware, and provides a minimal extension to the C programming language. CUDA’s success is closely followed by OpenCL (Open Computing Language), an open standard API administered by the Khronos Group. OpenCL sits on top of CUDA; its purpose is to provide an interface to effectively manage the pool of increasingly heterogeneous computational resources in a modern computer, such as GPUs, DSPs, and multi-core CPUs.

1.1 GPU Architecture Overview

Before the programming interface discussions, it is beneficial to look at the architecture design of GPUs. A simple raster graphics pipeline parses the input stream from a program, which consists of both instructions and their associated data (Figure 1.1(a)). Graphics primitives, usually in the form of streams of triangles, are passed down to the geometry stage where transform, clipping and lighting occurs on the primitives. The pixel stage takes the primitives, rasterizes and colorizes them before forwarding to the last stage of the pipeline for various framebuffer operations (ROP, or Raster OPerations) such as visibility determination (z-buffering), transparency, and anti-aliasing handling. This design is often referred to as immediate mode rendering where everything, including instructions and data, go into the graphics pipeline in-band.

Obviously it can be very inefficient to push all the data through the pipeline. For example, a vertex
Figure 1.1: A simple raster graphics pipeline. (a) A very simplified graphics pipeline. (b) Enhancing (a) with multiple geometry/pixel processors and a framebuffer arbiter (FB Mux) for external memory accesses.

should preferably be sent down only once even if it is shared by multiple triangles or instances of objects. It is also not possible to determine data such as texture values in advance - texture coordinates are usually computed in the pixel stage. To resolve this issue, units would need access to the framebuffer memory (Figure 1.1(b)). In this figure we also include a framebuffer mux to arbitrate accesses to multiple memory banks and clients (units). After adding distribution mechanisms between different stages of the pipeline, we can scale up the performances by putting additional geometry and pixel processing units in parallel.

The need to support various geometry and pixel operations makes it both logical and economical to transform both the geometry and pixel pipeline stages into general-purpose ALUs where individual programs, or shaders, can be run on these ALUs (Figure 1.2(a)). Both of these stages look increasingly similar, therefore it becomes evident to combine them into a single, unified shader design (Figure 1.2(b)). In addition to the obvious benefit of design cost reduction, a unified shader allows dynamic balancing between geometry and pixel stages of the rendering process. Previously, it is possible to saturate the geometry units while keeping the pixel units almost entirely idle by rendering many tiny triangles; a properly designed GPU with unified shaders does not suffer from this shortcoming.

For the shader processor to sustain a high throughput rate, hundreds or thousands of threads often need to be issued concurrently. Traditionally, however, each threads has its own local scope of registers and input/output data (Figure 1.3(a), Figure 1.4(a)), and output of each thread is only available through framebuffer interfaces which incur hundreds, if not thousands, cycles of latency. This means that it would be difficult to map algorithms as simple as a parallel matrix multiplication efficiently to a classic graphics pipeline(Figure 1.4(b)).

CUDA-enabled GPUs made an important modification (Figure 1.4(b)) that allows two types of data sharing. First, different threads within the same ALU can be made to share their registers through clever register renaming. Second, different ALUs can share these registers with proper synchronization and hazard detection. The actual designs of the shared register file may vary from a multi-threaded register file to a specialized cache. Irregardless of the actual design, it adds a crucial memory hierarchy between registers and framebuffer from a developer’s point of view.

1.2 About This Document

This document is meant as the written accompaniment to the tutorial GPU Computing with CUDA and OpenCL, presented on March 2 at IEEE Pacific Visualization 2010. Attendees can forego note-taking during
Figure 1.2: Programmable graphics pipeline. (a) Similar to Figure 1.1(b) but the geometry and pixel units are now closer to general-purpose processors. (b) The geometry and pixel ALUs in (a) are unified into one single design. Multiple processing cores can be added for scalability.

Figure 1.3: Thread execution model. (a) In traditional SIMD or graphics pipeline, each instance (or thread) of the same program is executed independently of the other. (b) Many algorithms need threads to share data between each other. Green arrows indicate data forwarding.
Figure 1.4: Memory in shader processors. (a) A straightforward design. (b) Shared register file design. Registers can be shared between different ALUs, or different threads within the same ALU. The register file can be allocated as shared cache, dedicated thread registers, or a combination of both.

The rest of the document is organized as follows. We provide a brief introduction on CUDA (Chapter 2) and OpenCL (Chapter 3). Chapter 4 walks through a real-life image processing problem using OpenCL. As this tutorial is meant as a high-level overview to be completed within a few hours, we ask the readers to continue exploring additional resources listed at the end of each chapter.
Chapter 2

CUDA

Our goal in this chapter is to give a very quick overview to programming with CUDA so that readers with prior C language experience can start programming after completing the tutorial. For in-depth treatments, please refer to the end of this chapter for additional references and resources.

2.1 The 10,000 Foot View

NVIDIA’s CUDA [1], from software development point of view, provides a minimal extension to C and C++ languages and a set of compiler and profiling tools that enables parallel programming on the GPUs. Developers writes a traditional serial version of the program with C/C++ and write subroutines called kernels that are designed for, and executed on, GPU shader multiprocessors. The serial part of the program launches the kernels and determines the number of instances, or threads, of the kernel to be executed. Threads corresponding to each kernel are organized into two levels - a thread belongs to a thread block, and the thread blocks are organized into a grid. Because threads within the same block have shared memory access within the block but not outside of the block, this organization effectively eliminated costly long-latency memory accesses. This model of execution is called SIMT(single-instruction, multiple-thread) in CUDA terminology.

2.1.1 CUDA versus GPGPU

Compared to GPGPU programming using OpenGL or DirectX, CUDA has a much friendlier learning curve than GPGPU, and the source code written with CUDA is cleaner and easier to maintain. In addition, because it executes in a hardware compute mode, some features such as shared memory can not be accessed using traditional graphics APIs. It is also easier to optimize CUDA’s performance across different GPUs - each family of GPUs fall within a revision of compute capability, and within each family most tuning parameters can be queried during run-time (through cudaGetDeviceProperties() function).

On the other hand, because CUDA is mostly mapped to the GPU shader multiprocessors, it do not have access to graphics-specific units such as triangle rasterizer, several texture mapping modes and most raster-op (ROP) functionalities including z-buffer and blending. Therefore it is beneficial to apply a mixture of both techniques in order to maximize performance. Fine-grain context switching between graphics and compute mode in a GPU is often costly and is best approached with caution.

2.1.2 Scientific Computation

CUDA-enabled GPUs with compute capability 1.3+ supports double-precision floating-point numbers. All compute devices follow the IEEE-754 standard with some deviations in specials such as NaN, denorm and rounding modes.
There exist several default CUDA libraries for matrix computation, such as CUBLAS (a CUDA BLAS implementation) and CUFFT (a CUDA FFT implementation). MATLAB users can access GPUs through the MEX API interface [4] which requires rewriting functions in CUDA, compiling them into MEX and modify the MATLAB code to use the new function. To run the original MATA LB code unmodified, a commercial solution is available (Jacket by Accelereyes, http://www.accelereyes.com) as well as an open-source project (GPUmat, http://www.gp-you.org).

2.2 A CUDA Crash Course

In this section we introduce readers to a “Hello World” example for CUDA. We setup an empty buffer on host memory, copies it to the GPU (or device) memory where a kernel writes the string “Hello CUDA!” in parallel into the device memory. The string is then copied back to the host memory and printed on the screen. To compile and run the code, refer to the “Getting Started” guide on CUDA website [1] for up-to-date platform-specific instructions. Once relevant tools are installed, simply type nvcc hello_cuda.cu to compile and build the example.

```c
// A hello world for CUDA
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>

// A simple kernel that copies a constant character to the output buffer
__global__ void hello_kernel(char *odata, int num)
{
    char hello_str[12]="Hello CUDA!";
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    if(idx < num)
        odata[idx] = hello_str[idx];
}

int main(void)
{
    // Initialize input/output buffers
    char *h_data, *d_data;
    const int strlen = 12;
    size_t strsize = strlen * sizeof(char);
    h_data = (char *) malloc(strlen);
    memset(h_data, 0, strlen);
    cudaMemcpy(void **&d_data, strsize);
    cudaMemcpy(d_data, h_data, strsize, cudaMemcpyHostToDevice);

    // Compute block size and launch the kernel
    int blocksize = 8;
    int nblock = strlen / blocksize + (strlen % blocksize == 0 ? 0 : 1);
    hello_kernel<<<nblock,blocksize>>>(d_data, strlen);

    // Get the results from GPU and display
    cudaMemcpy(h_data, d_data, sizeof(char)*strlen, cudaMemcpyDeviceToHost);
    printf("%%s\n", h_data);

    // Cleanup
    free(h_data);
    cudaFree(d_data);
}
```

Listing 2.1: A “Hello World” for CUDA (hello_cuda.cu).

We now describe the example in more details; relevant topics listed next to these steps are optional until readers are familiar with the code.

Memory Allocation (lines 16-22) and Cleanup (lines 30-35). Use cudaMalloc() to allocate memory on device and copy data between host and device memory through the cudaMemcpy() function. Specify
cudaMemcpyHostToDevice to copy from host to GPU, and cudaMemcpyDeviceToHost vice versa. When finished, delete the allocated device buffer by calling cudaFree().

**Memory Transfer** GPU device communicates with the host CPU through the slower PCI-E bus; minimize calls to cudaMemcpy() by finishing all the work on device before switching back to GPU. Using page-locked memory can improve the bandwidth between host to device (via cudaMemcpyHost(), cudaMemcpyHostToDevice()).

**Memory Types** Device memory allocated in this example are called *linear memory*. To allocate memory optimized for the texturing hardware on GPUs, use CUDA arrays instead (through type cudaArray, function cudaMemcpyHostToDevice()).

**2D and 3D Arrays** Use cudaMemcpy() and cudaMemcpy3D() to allocate 2D and 3D arrays respectively. This ensures proper stride are padded to meet the alignment requirement.

**Kernel Execution (lines 25-27).** Before launching the kernel on GPU, we need to determine (1) the number of threads per block (blocksize, and (2) the total number of blocks (nblocks). The total number of threads, which equals to (blocksize × nblocks) need to be equal or more than the number of characters in the output string - each thread copies a single char into the output buffer.

**Thread Block and Grid Size** Use cudaMemcpyHostToDevice() to obtain the maximum size of a thread block and grid supported by a particular device. A thread block can be a one-, two-, or three-dimensional block.

**Kernel (lines 6-12).** Use __global__ qualifier when declaring a CUDA kernel. Each kernel thread is uniquely identified by the built-in variables blockIdx, threadIdx. This kernel computes its own 1D thread number (idx, line 9) and copies a corresponding character to the output buffer odata located on the device memory. Note that kernels do not have access to the host memory.

**Divergence and Concurrency** Threads in a block is split up and executed in multiple ALUs simultaneously. Each ALU, or Shader Processor(SP) in NVIDIA terminology, executes a subset of the block in a pipelined fashion. This subset, called warp in NVIDIA GPUs, does not diverge from each other, which means that if threads of a warp diverge, each branch taken need to be executed serially while disabling the threads not on that path [2].

**Shared Memory** Threads within the same thread block can share variables declared with __shared__ qualifier. Multiple ALUs within the same shader multiprocessor may be executing parts of the same block, and it is necessary to use a lightweight barrier __syncthreads() to synchronize all threads within the block.

**Kernel Subroutines** The __global__ function type qualifier declares a function as a kernel which is executed on the device and called from the host. The __device__ qualifier declares a function executed and called both from the device, therefore functions called by a kernel should have a __device__ qualifier.

The CUDA-specific identifiers and functions mentioned in this section should provide a good baseline for readers to experiment with various parallel algorithms. We encourage readers to look at relevant SDK code examples provided by NVIDIA [5], such as transpose and matrixMul for shared memory usage, simpleCUBLAS, simpleCUFFT for usages on CUBLAS and CUFFT, reduction on various optimization techniques.
2.3 Tips on Optimization

Although porting application over to CUDA can often provide significant speedup out-of-box, even greater performance improvements can be made by understanding and tuning to specific GPU architecture. Here is our recommended order of optimization strategies. For more detailed treatment, refer to the Performance Guidelines chapter in “NVIDIA CUDA Programming Guide”.

1. Algorithm Optimization. Maximize parallelism and observe the complexity of the algorithm, including both time complexity and cost complexity. Notice that depending on the location of the memory location, it is sometimes be faster to recompute than to store/load the same result. Try to increase the math operations per bandwidth consumed.

2. Fill the Pipelines. A GPU is a large machine and needs thousands of threads to fill it in order to run at full efficiency. Merely increasing the number of threads may be counter-productive - the register file is evenly divided between all active threads, and a block may simply fail to launch due to register pressure.

3. Maximize Effective Memory Bandwidth. Memory bandwidth is precious resource in GPU. Different threads may cause bank conflicts while accessing shared or device memory. Unaligned memory access can also cause wasted bandwidth. Accessing device memory incur a very long latency and coalescing memory access is recommended to maximize bandwidth.

4. Maximize Instruction Throughput. Ancillary instructions can cause significant overhead and reduce the effective arithmetic throughput. Loop unrolling often helps - use \#pragma unroll to enable automatic unrolling by the compiler.

2.4 Additional Resources

NVIDIA’s official CUDA Website [1] provides a wealth of information. We recommend looking at the “Getting Started” guide for system-specific setup, “NVIDIA CUDA Programming Guide” for programming references, and SDK code examples [5]. Visual Profiler included in the CUDA Toolkit is particularly valuable for performance optimization.

GPGPU.org at http://www.gpgpu.org. GPGPU/CUDA Courses from Supercomputing 2007 [6] and other SIGGRAPH courses are extremely well organized. The very useful CUDPP library implements a wide variety of data-parallel algorithms and is worth exploring.

GPU Gems 3 book, also available online now [7]. The first post-CUDA book in the GPU Gems series.

Course Notes COMP790-058 from the University of North Carolina at Chapel Hill [8] and ECE498 from University of Illinois at Urbana-Champaign [9, 10] both offer excellent course notes online.

Papers For more detailed description of the NVIDIA Tesla architecture, refer to the IEEE Micro [11]. For the design of CUDA, see [12, 2]. These papers are authored by some of the key contributors to the design and implementation of Tesla and CUDA.
Chapter 3

OpenCL

This chapter is meant as the written accompaniment to the OpenCL portions of the tutorial *GPU Computing with CUDA and OpenCL*. We encourage readers to download the PDF of this document, available at painted.cs.ucdavis.edu/pv2010/gpgpu as the colorized source listings will be easier to read. Inlined source listings are highlighted according to Table 3 to help focus on the OpenCL-specific code.

3.1 The 10,000 Foot View

OpenCL is a royalty-free specification for a technology stack targeted at writing parallel programs. Though originally developed by Apple, Inc., OpenCL is a standard managed by the Khronos Group, the consortium also responsible for OpenGL and OpenGL ES. Its development is driven by the combined efforts of members of the Compute Working Group, which includes Apple, Intel, AMD, and NVIDIA. The broad industry support for OpenCL is good news for programmers in need of a parallel computing API, because OpenCL is not tied to the success or failure of one company or one line of hardware.

The royalty-free nature of OpenCL is key: this means that any hardware vendor can provide an implementation of the OpenCL specification without paying exorbitant sums of money to some third party. This is good for you, the hacker, because code you write against the OpenCL API today is likely to run on future hardware, provided that the vendor provides the appropriate CL driver. This is in counterpoint to vendor-specific stacks like CUDA, which are wholly controlled by one company, and support the hardware of one company.

OpenCL certainly supports GPU computing, but is not specific to GPU computing. Rather, it takes a higher-level view of the computer, and treats all many-core devices as potential vehicles for your parallel code. This means that the multi-core CPU setups now common in commodity machines are also targets for OpenCL code. For a large class of problems, a properly written OpenCL program can switch between CPU-based execution and GPU-based execution with minimal programmatic effort. Further, OpenCL provides the ability for programs to take advantage of both types of devices simultaneously. With OpenCL, you can keep all your hardware busy.

<table>
<thead>
<tr>
<th>Token</th>
<th>Color</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenCL type</td>
<td>Green</td>
<td>cl_mem, cl_device_type</td>
</tr>
<tr>
<td>OpenCL function</td>
<td>Red</td>
<td>clCreateBuffer()</td>
</tr>
<tr>
<td>OpenCL constant</td>
<td>Blue</td>
<td>CL_MEM_READ_ONLY</td>
</tr>
<tr>
<td>OpenCL keyword</td>
<td>Orange</td>
<td>global, kernel</td>
</tr>
</tbody>
</table>

Table 3.1: Color-coding for the embedded source listings.
3.2 The OpenCL Workflow

The hardest of attacking a problem with OpenCL really has nothing to do with OpenCL and everything to do with parallelizing your algorithm. Some problems, like the image filtering example in the next chapter, have an obvious data-parallel structure. But many problems you’ll want to accelerate will require significant algorithmic reworking in order to find the parallelism. The challenge of making efficient use of specific underlying hardware further complicates matters. A thorough treatment of parallel programming is beyond the scope of this tutorial.

Once you find the parallelism in your problem, it’s time to turn to OpenCL. Several of the documents listed in Section 3.6 provide detailed, entirely technical descriptions how OpenCL views your machine. Rather than rewrite these descriptions, we offer the following: Think of OpenCL as your personal robot army that specializes in attacking data. Each robot in your army belongs to a group, the size of which is up to you, the commander. Based on his unique id – and perhaps his group id – each robot grabs data from a gigantic, globally visible storage center. He wails and whacks on the data according to some recipe, and then puts the freshly mangled data away. Again, he uses his unique id to know where to put the finished product. These robots all run each instruction of the recipe you’ve given them in lockstep – they’re not smart enough to do different things at the same time. If you want one robot to do perform some special activity based on his id, the other robots are going to sit around doing absolutely nothing while he works. Presuming you’ve already figured out the algorithmic parallelism, the only remaining difficulty is figuring out the right number of robots to deploy, and ensuring that the recipe is written such that each robot will grab the correct data based on his id(s).

The robots above are OpenCL work items – independent threads of execution that run on the underlying hardware – and their program is called an OpenCL kernel. You learn about the OpenCL-enabled hardware in your system using the platform layer, and actually do stuff with those resources using the runtime. Most of the code you write against the platform and runtime APIs will be remarkably similar, following this basic formula:

1. Ask the platform layer for an OpenCL device, usually specifying a certain type of device. As we’ve said, OpenCL supports a variety of hardware, so you have to tell it what kind of device you’d like to use. We don’t cover it in this tutorial, but you can use multiple devices to simultaneously process work. This includes workloads that require the same pieces of data, though that requires a pretty detailed understanding of your hardware to work efficiently.

2. Ask the platform layer for an OpenCL context, a data structure that essentially ties all the devices, memory, and kernels you want to use into one nice package. You don’t really do anything directly to the context, but provide it as a parameter to platform and runtime methods.

3. Ask the runtime to compile your OpenCL program, which is typically a text file containing one or more kernels and supporting functions written in the C-derived OpenCL programming language. After compilation, you can obtain references to specific kernel objects which you will later use for actually executing your work.

4. Ask the runtime for a command queue tied to the device. This command queue is exactly what it sounds like: a data structure doing stuff in a certain order. In the case of OpenCL, stuff means operations like allocating memory on the device, reading and writing this memory, and running kernels.

5. Passing your just-acquired command queue, ask the runtime to allocate the memory you need for both the input and output of your kernels. There are two types of memory objects in OpenCL, which we’ll talk about in the hands-on code of Chapter 4.

6. Via the runtime, set your kernel parameters. We’ll see examples later; it’s straightforward.

7. Execute one or more kernels. You must specify the total number of work items and can optionally specify the decomposition of the work items into work groups. To help facilitate the parallel decomposition of your problem, you can specify these number in 1, 2, or 3 dimensions. As we’ve mentioned,
getting this number and decomposition right is tricky, and it’s a part of parallelizing your algorithm. Many people just coming to OpenCL choke on this aspect of programming; for now, just think in terms of starting up the right number of robots to *get all your work done*.

8. Ask the runtime to read back your results, unless you’re going to keep it one the device for some additional task(s).

These steps typically change only in small details from program to program, and once you’ve been coded though it a few times you will not think much about the platform and runtime. Almost all of the meat of OpenCL programming is in the kernels, which are entirely application dependent.

We’re glossing over a lot, particularly regarding the OpenCL view of the memory system. But the above view is about the right level to begin your programming with this new technology. After you’ve written a few kernels and understand where this simple views breaks down, you’ll be ready for the intermediate and advanced guides mentioned in Section 3.6.

### 3.3 The Platform Layer

The platform layer boils down to a handful of function calls that provide the following services to you, the programmer:

**Context Creation** The function `clCreateContext` is a pre-requisite for most of the host-side code you will write.

**Device Information** The platform layer provides the methods `clGetDeviceIDs` and `clGetDeviceInfo`. The former simply returns one or more `cl_device_ids` identifying the hardware in your system that OpenCL can use to execute work for you. The latter function is much more powerful; given a `cl_device_id`, OpenCL can tell you all manner of details about that hardware. You can learn the vendor, the size of local memory, the maximum amount of global or constant memory, the types of supported images – all sorts of useful information for planning a workload. Again, a comprehensive list of obtainable information is in the OpenCL specification.

**Platform Information** There are similar functions for learning about the *platforms* available on your system (i.e. Apple’s OpenCL vs. Intel’s OpenCL), but this isn’t really useful just yet. Most OpenCL implementations only report themselves since the various vendors have not yet figured out the details of this aspect of the system. Expect this part of the platform layer to evolve as more OpenCL implementations become available for a wider variety of hardware.

Aside from context creation, you can think of the platform layer as the part of OpenCL that helps you plan your workload based on details of your system capabilities. This planning might happen as part of development, or might be the result of device inspection during startup (e.g. support a range of GPUs).

### 3.4 The Runtime

While the platform layer tells you about your OpenCL-enabled hardware, the runtime is the part of the software stack that actually lets you *do work* with that hardware. The runtime has API calls to do the following:

**Handle Device Memory** It provides flexible calls to allocate both buffers and images, and provides functionality to get data in and out of these memory objects.

**Compile OpenCL Programs** The runtime includes calls to both compile text-based OpenCL programs and load existing compiled CL binaries.
Deal with Kernels  One you have a compiled program, the runtime provides functions to reference kernels, set arguments on those kernels, and then...

Execute Work  ... launch work using those kernels. You control the number of total work items, as well as the decomposition of the index space using two parameters to the function clEnqueueNDRangeKernel. NDRange stands for n-dimensional range, alluding to the fact that you can decompose your index space into 1, 2 or 3 dimensions to best suit your problem.

Event Management  While the examples we’ll cover are all based on (the default) in-order command queue behavior, more advanced OpenCL code will probably require some level of host-side data synchronization. The runtime provides an event system to allow programmers to develop multi-device control flows of arbitrary complexity. You’ll notice this even if you don’t use it – most of the runtime calls contain three parameters related to event management. We’ll see this later in the hands-on code of Chapter 4.

In addition to the explicit work the runtime does via API calls, it has another, less visible function. The runtime works behind the scenes to provide certain guarantees about data and command ordering. For example, let’s say that you create a command queue on the GPU device. You have two kernels, the first of which produces an image memory object that is used as input to the second kernel. Although the calls the clEnqueueNDRangeKernel return immediately in an asynchronous fashion, the OpenCL runtime make sure that the first kernel runs to completion before the second begins execution. It’s possible to use the event system with both out-of-order queues and multiple queues to create very complex workflows, but most beginner programmers will start with this basic, in-order setup.

3.5 The OpenCL Programming Language

You program your OpenCL kernels in the OpenCL programming language, a C99-derived language that includes some additional types and functions, as well as many restrictions. The language, additional features, and restrictions are covered in much detail in the OpenCL specification (see below). Some highlights:

Types  OpenCL supports all the types you’d expect: (u)char, (u)int, (u)long, and float. It also provides vector versions of each type (e.g. uchar4, uint8). If it fits your algorithm, it can be performance-beneficial to use the vector types as the OpenCL compiler can use any underlying vector hardware (SSE, AltiVec, etc.). CL also include less-precise half types for cases where you wish to sacrifice floating point precision for packing more data on the device.

Memory Spaces  Memory you touch in the kernel lives in one of four available memory spaces: global, private, local, or constant. The first refers to memory in the global pool – typically buffers or images you’ve allocated. The variables local to the kernel (and private to the work item) are in private memory. local memory is the space visible to all work items in one work group, and is usually backed by some kind of very fast, size-limited, on-chip memory. Expert CL programming for the GPU requires an excellent understanding of how best to use this memory. constant memory is a size-limited portion of memory typically optimized for uniform broadcast to all work items.

Built-in Functions  OpenCL contains a significant number of built-in functions. Their use ranges from obtaining work item/group ids to min/max computation to trig functions. Many of the math-oriented functions contain variants that are optimized for performance on certain types (e.g. mad24).

No recursion  This needs little explanation; you cannot use recursive function calls at this time.

3.6 Other Sources

This tutorial is aimed at the new OpenCL initiate. As you are visualization developers, we anticipate that you are familiar with OpenGL and programmable shaders. While such knowledge is certainly helpful
in understanding this tutorial, it’s not required. Our goal is to take users who have no experience using
OpenCL and provide them with sufficient conceptual and working knowledge to begin developing useful code
using this new technology. Yet some of you will undoubtedly crave additional information. Unfortunately,
OpenCL is a technology in its infancy, and materials for self-education are meager. Still, there are a a few
decent sources of information, as indicated below. We’ve included shortened bit.ly URLs to each resource
for convenience.

The OpenCL Specification  The definitive reference is the specification, available from the Khronos OpenCL
API registry (bit.ly/1WwI). This well-maintained document covers the platform layer, runtime, and
kernel programming language in detail with respect to the expected behavior of functions and language
features. It is not the best starting point for a beginner, however, as it does not cover much conceptual
ground and offers few real-world examples. The chapter covering the OpenCL Programming language
is indispensable for kernel-writers.

The Khronos Forums  There are several forums (bit.ly/dwBFXB) dedicated to OpenCL, the most active
of which are ‘OpenCL’ and ‘Coding:Beginner’. If you encounter a strange problem or just need some
implementation advice, we’ve found that a clear writeup usually nets a response in no more than a few
days.

Apple’s Documentation  Apple provides a relatively short OpenCL programming guide (bit.ly/axAWvL)
that is an excellent starting point for an absolute beginner. Apple is also the current best source for
OpenCL code examples. (bit.ly/8GLQmc, filter on ‘OpenCL’).

NVIDIA’s OpenCL Guides  NVIDIA provides several informative guides aimed at CL programmers of
varying levels. The OpenCL Programming Overview (bit.ly/ckHVyh) covers some key aspects of
GPU computing on NVIDIA hardware, but requires basic OpenCL knowledge (this tutorial will be
enough). The OpenCL Programming Guide (bit.ly/ctwxcL) is a more comprehensive examination
of the same topic. The Best Practices Guide (bit.ly/y9LrJ) is for the more seasoned CL programmer
wanting to extract the absolute maximum performance out of her NVIDIA hardware. Finally, the
OpenCL Jumpstart Guide (bit.ly/8BC0IN) is for hackers already familiar with CUDA who want to
learn OpenCL.

AMD Guides  AMD also provides some CL programming help in the form of an introduction (bit.ly/exHb0k)
and programming guide (bit.ly/aQNbrz).

MacResearch Video Tutorials  This is a series of videos covering CL development on the Mac. Even if
you’re not on a mac, the discussion of shared memory will be helpful. (bit.ly/15plQ6)

The OpenCL Quick Reference  Once you have a good grasp of the CL programming, the quick reference
card (bit.ly/19T08T) is a useful document to keep handy.
Chapter 4
OpenCL in Practice

As with any new API, working through example code is one of the best ways to learn the down-and-dirty details. In this Chapter, we present the straight-forward but non-trivial application of OpenCL to an image processing problem: maximum luminance filtering. We provide the initial C code that a typical programmer might write, and then show how to (easily) convert it to OpenCL for execution on both the CPU and GPU. We then show how using OpenCL image objects can benefit this particular application. Finally, we walk through the code and considerations involved in sharing objects between OpenCL and OpenGL. To keep the included code listings as short as possible, we’ve trimmed out some comments and error handling. For the full source code for all examples, please visit painted.cs.ucdavis.edu/pv2010/gpgpu.

We include performance numbers throughout this section. Our test machine is a Mac Pro with two dual-core 2.66GHz Intel Xeon processors, 8GB RAM, and an NVIDIA 8800GT (512MB, PCIe 1.1) graphics board. We use Apple’s OpenCL 1.0 implementation on both the CPU and GPU.

4.1 Maximum Luminance Filtering

Luminance ($l$) in this context is the human-perceived brightness of a given color, computable with the formula:

$$l = 0.30r + 0.59g + 0.11b$$

where $r$, $g$, and $b$ are the color components of an RGB pixel. This formula might appear arbitrary if you haven’t seen it before, but it stems from the human eye’s unequal response to different colors. In addition to a wealth of other fascinating topics, the interested reader can find a detailed discussion of luminance in Glassner [13].

Maximum luminance filtering (MLF) has a common image processing structure. It takes as input an RGB bitmap and a filter width ($w$) and uses them to compute the output image. The width defines a moving, $w$-by-$w$ region of pixels to be considered for each pixel of the input, as depicted in Figure 4.1. For each input pixel, MLF selects the pixel from the filter region with maximum luminance as the output value. The end result is an blurred, almost impressionistic version of the original image.

MLF is a good choice for our first sample application, as it exhibits a structure common to many pixel-processing applications. Further, both the C and OpenCL implementations are easy to understand while still requiring enough of the API to be useful as instruction.

4.2 The CPU Implementation

Below is Listing 4.1, the portion of the C-based implementation of maximum luminance filtering that does the “real” work. The full source code is contained in the folder pure_cpu in the code archive. The transition from algorithm to code is straightforward. There are three key memory buffers, pixels (the original RGBA
Figure 4.1: An example of maximum luminance filter of width 13. The filter first computes the luminance value for each pixel of the input image (a) and stores it in an intermediate buffer (b). The filter iterates over the input image pixel-by-pixel considering the 13x13 grid (c) of surrounding pixels for each. It selects the location from the grid with the maximum luminance (green) and uses that as the output pixel for that particular x,y (orange) location. The resulting image (d) is an almost impressionistic variation of the original wherein brighter areas bleed outward.
data), luminance (the calculated luminance image), and filtered (the output) which are declared and initialized outside the scope of this listing.

```c
for (findex = 0; findex < rgba_size; findex+=4) {
    lindex = (unsigned int)(findex * 0.25);
    luminance[lindex] = pixels[findex+0]*0.30 + pixels[findex+1]*0.59 + pixels[findex+2]*0.11;
}
for (x = 0; x < IMAGE_WIDTH; x++) {
    for (y = 0; y < IMAGE_HEIGHT; y++) {
        maxlum = 0;
        xlo = x - (width >> 1); xhi = x + (width >> 1);
        ylo = y - (width >> 1); yhi = y + (width >> 1);
        xlo = xlo < 0 ? 0 : xlo;
        ylo = ylo < 0 ? 0 : ylo;
        xhi = xhi > IMAGE_WIDTH - 1 ? IMAGE_WIDTH - 1 : xhi;
        yhi = yhi > IMAGE_HEIGHT - 1 ? IMAGE_HEIGHT - 1 : yhi;
        for (xx = xlo; xx <= xhi; xx++) {
            for (yy = ylo; yy <= yhi; yy++) {
                lindex = yy * IMAGE_WIDTH + xx;
                if (luminance[lindex] >= maxlum) {
                    maxlum = luminance[lindex];
                    xmax = xx;
                    ymax = yy;
                }
            }
        }
        maxindex = ymax *(IMAGE_WIDTH*4) + (xmax*4);
        outindex = y * (IMAGE_WIDTH*4) + (x * 4);
        filtered[outindex+0] = pixels[maxindex+0];
        filtered[outindex+1] = pixels[maxindex+1];
        filtered[outindex+2] = pixels[maxindex+2];
        filtered[outindex+3] = pixels[maxindex+3];
    }
}
```

Listing 4.1: ANSI C implementation of maximum luminance filtering.

The first loop (lines 1-7) applies the luminance formula described in Section 4.1 to each pixel of the original image and stores the result to an intermediate luminance buffer. The doubly-nested loop (lines 9-42) executes for each pixel of the input image, using the pre-calculated values to determine the filter location with the maximum luminance. This location is then used to copy the correct pixel from the input image to the output image. Lines 14-20 calculate the filter x,y range from the filter width and input pixel x,y location and clamp these values to image extents.

We fed this implementation a sample 512x512 RGBA image and asked it to apply a 25x25 filter 500 times. We performed this test 10 times, and recorded the best observed time and the average time. When compiled by gcc with -O3 -mtune=native, the best time was 169.64 seconds, and the average was 169.67 seconds. That’s equates to about 3Hz, which is probably acceptable performance for an image editing program. But what if we wanted to – for example – processes video frames in real time? Can we do better using the ample resources in our test machine?
4.3 OpenCL Conversion

Like many image processing problems, MLF contains much potentially independent, data-parallel computation. Both the luminance computation pass and the maximum-finding pass operate on a per-pixel basis, with no need for the results computed by neighboring pixels. This type of structure is an ideal fit for OpenCL, and in this section we present a buffer-based approach to the problem. The full source for this example is in the folder `cl_buffers` in the code archive.

Whenever you find yourself saying something like “for each foo, compute bah,” that is your indicator to write a kernel that computes bah on a single foo and launch it over a range big enough to cover all the foos in your data set. Note that this isn’t always the best parallel approach on all hardware architectures, but it is usually a good start.

Before we deal with the support code needed to use OpenCL, let’s look at a kernel (Listing 4.2). In the case of the luminance computation, we can write a kernel that operates on a single pixel of the input image and launch that kernel over a two-dimensional range equal to the width (x) and height (y) of the image. The kernel can compute a unique pixel id based on the x value, y value, and x-dimension width. This unique id indexes the input image to provide the pixel, and also tell the kernel where to put the result in the output image.

```
void kernel luminance(global uchar4 *orig, global uchar *lum) {
    // index = y * width + x
    uint gid = get_global_id(1) * get_global_size(0) + get_global_id(0);
    uchar4 pixel = orig[gid]; lum[gid] = pixel.x*0.3 + pixel.y*0.59 + pixel.z*0.11;
}
```

Listing 4.2: A kernel which computes the luminance per pixel of an input image.

Though this is a very simple kernel, there are a few points worth noting. First is the use of the `kernel` keyword in the function signature. This is how the CL compiler knows this is a externally-visible kernel, as opposed to a helper function. Also note that the kernel expects two pointers to memory objects in global memory space: `orig` of type `uchar4`, and `lum` of type `uchar`. These correspond, as we’ll see, to memory objects configured on the host side. As discussed above, the kernel uses the x and y work item ids to determine the unique location into the input/output arrays to use for its work. Note that in some older code examples, you might see `_global` or `_kernel`. The double underbar was required for memory space specifiers and the kernel keyword in earlier versions of the CL specification, but is no longer necessary. We suggest avoiding the clutter.

The maximum luminance calculation maps just as easily to an OpenCL kernel, which is presented in Listing 4.3.

```
void kernel max_filter(const uint width, global uchar4 *orig, global uchar *lumi,
                       global uchar4 *result) {
    int i, j, index, maxindex = 0;
    uchar lum, maxlum = 0;
    int4 range;
    int x = get_global_id(0);
    int y = get_global_id(1);
    int xdim = get_global_size(0);
    int ydim = get_global_size(1);

    uint gid = mad24(y, xdim, x);

    int xlo = x - (width >> 1); int xhi = x + (width >> 1);
    int ylo = y - (width >> 1); int yhi = y + (width >> 1);

    xlo = xlo < 0 ? 0 : xlo; xhi = xhi > xdim-1 ? xdim-1 : xhi;
    ylo = ylo < 0 ? 0 : ylo; yhi = yhi > ydim-1 ? ydim-1 : yhi;
}```
for (i = xlo; i <= xhi; i++) {
    for (j = ylo; j <= yhi; j++) {
        index = mad24(j, xdim, i);
        lum = lumi[index];
        maxindex = lum > maxlum ? index : maxindex;
        maxlum = lum > maxlum ? lum : maxlum;
    }
}
result[gid] = orig[maxindex];

Listing 4.3: Searches a neighborhood around the pixel to find the location of maximum luminance.

This kernel takes the filter width, the original image data, the previously computed luminance data, and an output image. Note that width is in constant memory space, while the other three parameters are in global space. On some OpenCL devices, constant memory space is the right choice when you have a limited amount of data to be broadcast to all work items.

Line 13 is the same x,y to 1D array index mapping we’ve seen before, but it uses the built-in `mad24(a, b, c)` (multiply-and-add, \(x = a \cdot b + c\)) to perform the computation. We do this because certain architectures contain multiply-and-add instructions, the use of which can effectively perform two computations for the cost of one. The ‘24’ version of `mad()` is a “fast math” variant useful when the multiplicands \(a, b\) above are both integers under 24 bits.

Lines 15-19 are the same filter range calculation and index clamping as used in the pure C variant. Further, the doubly-nested loop in lines 21-29 is also nearly identical to the previous code. Instead of using an if statement, however, we use the ternary operator to check against the previous maximum luminance. For each filter location we ask: “Is this luminance greater than the max?” If so, we note the new maximum luminance (line 27) and the index where it can be found (line 26). Finally, we use the index of the maximum value to copy a pixel from the input image to the output image (line 31).

Now that we have two kernels which together perform maximum luminance filtering, let’s look at the OpenCL “infrastructure” code required to put them to work. The needed code breaks down into four basic components: OpenCL setup, memory setup, kernel setup, and kernel execution.

### 4.3.1 OpenCL Setup

The code below is called only once during program startup. In line 11, we set the `cl_device_type` used to ask for devices based on whether the user wants to use the CPU or the GPU, as determined by the function parameter `gpu`. Line 13 asks OpenCL for compute devices in the system matching that type. Note that the third parameter to `clGetDeviceIDs` is 1. That because in our particular case, we know we are working on a single GPU system. If you have a system with multiple GPUs you’d like to use, you should pass a value > 1 to this method and provide sufficient room in the `device_id` variable.

```c
// declared above:
static cl_device_id device_id;
static cl_device_type device_type;
static cl_context context;
static cl_command_queue commands;

static int initialize_compute(int gpu) {
    cl_int err;
    device_type = gpu ? CL_DEVICE_TYPE_GPU : CL_DEVICE_TYPE_CPU;
    err = clGetDeviceIDs(NULL, device_type, 1, &device_id, NULL);
    if (err != CL_SUCCESS) error_and_exit("clGetDeviceIDs", err);
    context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &err);
    if (err != CL_SUCCESS) error_and_exit("clCreateContext", err);
    commands = clCreateCommandQueueWithProperties(context, device_id, CL_QUEUE_PROFILER_ENABLE
```
context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &err);
if (!context) error_and_exit("clCreateContext", err);

commands = clCreateCommandQueue(context, device_id, 0, &err);
if (!commands || err != CL_SUCCESS) error_and_exit("clCreateCommandQueue", err);

return CL_SUCCESS;
}

Listing 4.4: An excerpt from cl_buffers/main.c – illustrates basic OpenCL context and command queue setup.

Line 16 creates an OpenCL compute context based on the device discovered above. Note that while we’re not using the fourth parameter to clCreateContext (PFN_notify), but it can be a useful debugging aid. This parameter is a callback function that the user can register. The OpenCL implementation can then use this function to report errors that occur in function calls using this context. Apple, for example, provides the callback function clLogMessagesToStdoutAPPLE, which can be used like this:

context = clCreateContext(NULL, 1, &device_id, clLogMessagesToStdoutAPPLE, NULL, &err);

This will cause any errors related to this OpenCL context to show up on standard output, which can be a development timesaver. Line 19 creates a command queue using both the newly created context and the device identified by device_id. This is a critical point: contexts are broad and could manage several devices at the users request, while command queues are specific to a single device. If you want to schedule work on multiple devices, you must create multiple command queues. But in our filtering example, we’re only interested in using either the GPU or the CPU for kernel execution, but not both at the same time.

4.3.2 Memory Setup

The memory requirements for our OpenCL program are conceptually identical to the C version. We need three OpenCL memory objects: one buffer to store the RGBA pixels of the input image, one buffer to store each result from the per-pixel luminance computation, and one buffer to store the output image pixels.

cl_mem original, luminance, result;
static int initialize_memory() {
    cl_int err;
    int size = IMAGE_WIDTH * IMAGE_HEIGHT * 4;

    unsigned char* data = (unsigned char*)calloc(sizeof(unsigned char), size);
    FILE* orig = fopen(IMAGE_FILE, "rb");
    fread((void*)data, sizeof(unsigned char), size, orig); fflush(orig); fclose(orig);

    original = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(unsigned char)
    * size, NULL, &err);
    err = clEnqueueWriteBuffer(commands, original, CL_TRUE, 0, size, data, 0, NULL, NULL);
    if (err != CL_SUCCESS) error_and_exit("WriteBuffer", err);

    luminance = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(unsigned char)
    * IMAGE_WIDTH * IMAGE_HEIGHT, NULL, &err);
    if (!luminance) error_and_exit("clCreateBuffer", err);

    result = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(unsigned char)
    * size, NULL, &err);
    if (!result) error_and_exit("clCreateBuffer", err);

    free(data);

    return CL_SUCCESS;
}

Listing 4.5: An excerpt from cl_buffers/main.c. Despite the verbosity this code simply creates three buffers and initializes one of them using data loaded from disk.
Lines 6-8 load the input RGBA pixels from disk. Note that this could easily be a call to a PNG or JPEG loading routine, but we’ve kept it simple for clarity. Line 10 creates an OpenCL buffer object to hold these pixels. Note that we use the memory flag `CL_MEM_READ_ONLY` which tells OpenCL that the memory object is read-only from OpenCL’s point of view. This means that kernels are not expected to write to this buffer, and the CL runtime might be able to utilize this knowledge to optimize access to this memory based on a quirk or feature of the underlying architecture.

Line 12 enqueues a write to the newly allocated memory object, passing in the RGBA pixels. We specify a blocking write (CL_TRUE as the third parameter) so that in line 23 we can delete the temporary buffer used to grab the pixels from disk. Note that the buffer creation requires only a CL context, but the actual write is tied to a specific command queue - a specific device.

The luminance buffer creation (line 15) uses the flag `CL_MEM_READ_WRITE` since the first kernel will write to this buffer, and the second kernel will read from it. Note that we do not provide initial data for this buffer, as it is unnecessary. Line 19 creates the output buffer, and uses the memory flag `CL_MEM_WRITE_ONLY` since the last kernel will write into this buffer, but never read from it.

### 4.3.3 Kernel setup

In order to use our kernels we must first provide them to the OpenCL runtime for compilation\(^1\). Listing 4.7 provides an example of creating an OpenCL program object from a source file on disk, and obtaining kernel objects from that program.

```c
static cl_kernel klum, kmaxf;
static cl_program program;

static int initialize_kernels() {
    cl_int err;
    char *source = NULL;
    size_t length = 0, len;
    textFromFile("kernels/kernels.cl", &source, &length);
    program = clCreateProgramWithSource(context, 1, (const char**)&source, NULL, &err);
    if (!program) error_and_exit("clCreateProgramWithSource", err);
    err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
    if (err != CL_SUCCESS) {
        char buffer[2048];
        fprintf(stderr, "Failed to build CL source. Error log:\n");
        clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG,
                                sizeof(buffer), buffer, &len);
        fprintf(stderr, "%s\n", buffer);
        return EXIT_FAILURE;
    }
    klum = clCreateKernel(program, "luminance", &err);
    if (!klum || err != CL_SUCCESS) error_and_exit("clCreateKernel", err);
    kmaxf = clCreateKernel(program, "max_filter", &err);
    if (!kmaxf || err != CL_SUCCESS) error_and_exit("clCreateKernel", err);
    return CL_SUCCESS;
}
```

Listing 4.6: An excerpt from `clbuffers/main.c` – kernel creation.

\(^1\)Note that it is possible to use pre-built binaries and in situations where initial startup time is important, this might be a good choice. However, using pre-built binaries is very fragile in the sense that your must provide a binary for the exact hardware and driver combination to ensure valid results. For in-house code that run only on known hardware this might be plausible. For software aimed at the wide world, however, this isn’t likely feasible.
We load the file from disk into a character buffer (line 9) and then provide it to the OpenCL runtime via `clCreateProgramWithSource` (line 11). One under-appreciated aspect of OpenCL is this just-in-time compilation, which permits any changes we desire to the source code before submission to the CL compiler (e.g., programatically unrolling loops based on runtime parameters). The call to `clBuildProgram` in Line 14 actually performs the compilation and linking. Users can pass options to the CL compiler via the fourth parameter, and can register a callback function that is triggered when the compilation is complete. Note that when this parameter is `NULL`, as in our example, the compilation is blocking. If the compilation fails, we print out the build log. Otherwise, we obtain kernel objects for both the luminance kernel (line 25) and the filtering kernel (line 28).

### 4.3.4 Kernel Execution

At this point we’ve loaded our image data and fed it to OpenCL, allocated other needed memory, compiled our CL program, and obtained references to the needed kernels. The last step is execution, detailed in Listing 4.7.

```c
static int execute_kernel(void) {
    unsigned int i, t, width = FILTER_WIDTH;
    cl_event event = NULL;
    cl_int err = CL_SUCCESS;

    unsigned char *result_host = (unsigned char*)malloc(sizeof(unsigned char) * IMAGE_WIDTH * IMAGE_HEIGHT * 4);

    unsigned int arg = 0;
    err = clSetKernelArg(klum, arg++, sizeof(cl_mem), &original);
    err = clSetKernelArg(klum, arg++, sizeof(cl_mem), &luminance);
    arg = 0;
    err = clSetKernelArg(kmaxf, arg++, sizeof(unsigned int), &width);
    err = clSetKernelArg(kmaxf, arg++, sizeof(cl_mem), &original);
    err = clSetKernelArg(kmaxf, arg++, sizeof(cl_mem), &luminance);
    err = clSetKernelArg(kmaxf, arg++, sizeof(cl_mem), &result);

    if (err != CL_SUCCESS) error_and_exit("clSetKernelArg", err);

    size_t global_dim[2] = { IMAGE_WIDTH, IMAGE_HEIGHT };
    err = clEnqueueNDRangeKernel(commands, klum, 2, NULL, global_dim, NULL, 0, NULL, &event);
    err = clEnqueueNDRangeKernel(commands, kmaxf, 2, NULL, global_dim, NULL, 0, NULL, &event);
    err |= clEnqueueReadBuffer(commands, result, CL_TRUE, 0, sizeof(unsigned char) * IMAGE_WIDTH * IMAGE_HEIGHT * 4, result_host, 1, &event, NULL);

    // do whatever you’d like with the results, then:
    free(result_host);

    return CL_SUCCESS;
}
```

Listing 4.7: An excerpt from `cl_buffers/main.c` – kernel execution.

OpenCL requires that we explicitly set each kernel argument via a call to `clSetKernelArg`, which we can see in lines 10-17. This function is straightforward – it requires the kernel, the argument number, the size of the argument, and the argument itself. A common mistake here is a mismatch between the host-side argument setup and the CL kernel code.

Once the arguments are set, we can fire away. We choose to execute our two kernels per pixel via a two-dimensional global range (lines 23 and 25), which creates `IMAGE_WIDTH x IMAGE_HEIGHT` work items per kernel. In this particular example, we allow OpenCL to choose the local range, passing `NULL` as the
sixth parameter of `NDRangeKernel`. Note that the second kernel uses output from the first as input (the luminance image). This is perfectly fine since we’re using the default in-order behavior on our command queue. OpenCL will ensure that the data is correct. It is possible to specify an out-of-order queue, or use objects on queues for different devices. In that case, it’s up to the programmer to maintain synchronization and ordering using events. Line 25 shows how to obtain an event for a CL call – even though we don’t really need it here – and line 28 waits on that event before reading back the result of the filter.

### 4.3.5 Performance

Though it may seem like a lot of code compared to the pure C example, the bulk of the above code listings is “boilerplate” code that changes very little from one application to the the next. The main elements are always the same: obtain your command queue(s), allocate and possibly initialize memory, compile your program and obtain kernel references, and finally execute your workload. As previously stated, the most difficult part of converting to OpenCL is deciding how to parallelize. Some algorithms – like image convolution – are parallelizable in an obvious way. Other algorithms are more difficult to adapt to SIMD-style computing, particularly when it comes to efficient utilization of the device memory hierarchy.

So the big question is, “what do we get for our coding efforts?” Recall from Listing 4.4, line 11 that we can flip between the CPU device and GPU device very easily in this program. We tested each device using the same method as our test for the pure CPU implementation: 500 iterations of a 25x25 maximum luminance filter on 512x512 image, reporting the best and average of 10 runs. Running on the CPU, we saw a best time of 86.9 seconds (1.94x speedup) and an average time of 87.1 seconds. On the GPU device, we saw a best time of 57.3 seconds (2.96x speedup), and an average of 57.4 seconds. The GPU test includes a memory read back per iteration.

### 4.4 Image Object Upgrade

While a threefold increase in performance is good, it’s not really in keeping with our expectations of the NVIDIA 8800GT. The problem with our buffer-based implementation is its inefficient use of the 8800’s memory system. On current GPUs, global memory access is very slow compared to local memory access, and our filter makes repeated accesses to the same locations as part of the per-pixel neighborhood lookup. We could rework our filtering algorithm to use a clever decomposition of the index space such that we could move each piece of global memory to fast, on-chip local memory once, and then make our repeated queries into that lower-latency space. This would almost certainly be much faster on the GPU, but requires changes at an algorithmic level that are likely beyond a novice parallel programmer.

OpenCL image objects, on the other hand, offer a alternative for obtaining improved memory performance that require no algorithmic changes for a certain class of problems. If your algorithm exhibits spatial locality of global memory access across work items, especially for inherently two-dimensional index spaces, then image objects can likely improve performance with little programmer effort.

Image objects provide cached access to global memory at the expense of some flexibility. Images are either 2D or 3D, and are limited to a maximum size in each dimension based on the underlying hardware. Also, images support a hardware-defined range of underlying data types (UINT_8, FLOAT, etc.). Contrast this to buffer objects, which are one dimensional, support all primitive types, and are not size-limited (up to the total memory on the device).

Image objects can be used to store images that are not related to the execution of kernel computations. The modifications required to leverage images objects in our code are non-trivial, but not overwhelming. Context and command queue setup, program compilation, kernel setup, and kernel execution are essentially identical to the buffer-based code. What does change? Only two areas of code need to be modified: the host-side memory object creation calls, and the CL kernel code.
4.4.1 Kernel Changes

Both revised kernels are showing in Listing 4.8. The first obvious change is the addition of the sampler objects at the top of the file. Samplers should be familiar to anyone who has worked with textures in OpenGL. They can be viewed as a lens through which image data is accessed, and provide features like range clamping and interpolation “for free.” We declare two samplers, both of which use non-normalized coordinates and use nearest-neighbor filtering. Non-normalized in this context means we can access our data using the work item x and y coordinates. If it was applicable to our problem, we could use 0.0 - 1.0 normalized coordinates instead. The first sampler, sampler_plain, does not perform address clamping while the second sampler, sampler_clamp clamps image access to the extents.

```c
const sampler_t sampler_plain = CLK_NORMALIZED_COORDS_FALSE | CLK_ADDRESS_NONE |
                                 | CLK_FILTER_NEAREST;
const sampler_t sampler_clamp = CLK_NORMALIZED_COORDS_FALSE | CLK_ADDRESS_CLAMP_TO_EDGE |
                             | CLK_FILTER_NEAREST;

void kernel luminance(const image2d_t input, write_only image2d_t luminance) {
    int2 coord = (int2)(get_global_id(0), get_global_id(1));
    uint4 pixel = read_imageui(input, sampler_plain, coord);
    uint4 lum = (uint4)(pixel.x*0.30 + pixel.y*0.59 + pixel.z*0.11);
    write_imageui(luminance, coord, lum);
}

void kernel max_filter(const uint width, const read_only image2d_t input, read_only image2d_t luminance, write_only image2d_t output) {
    uint maxlum = 0;
    uint4 lum;
    int i, j;

    int2 coord = (int2)(get_global_id(0), get_global_id(1));
    int2 maxcoord;

    int xlo = coord.x - (width >> 1); int xhi = coord.x + (width >> 1);
    int ylo = coord.y - (width >> 1); int yhi = coord.y + (width >> 1);

    uint count = 0;
    for (i = xlo; i <= xhi; i++) {
        for (j = ylo; j <= yhi; j++) {
            lum = read_imageui(luminance, sampler_clamp, (int2)(i,j));
            if (lum.x > maxlum) {
                maxlum = lum.x;
                maxcoord = (int2)(i,j);
            }
        }
    }
    uint4 value = read_imageui(input, sampler_clamp, maxcoord);
    write_imageui(output, coord, value);
```

Listing 4.8: The two kernels after image conversion.

As before, the luminance calculation kernel (lines 7-15) is dead simple. We obtain the x and y work item ids and use them as coordinates to read from the input image. After calculating the luminance, we use the same coordinates to write to the output image. One small detail of note is the read-only designation of the input image, and the write-only designation of the output image. This is another small restriction of images; they can be either read or written, but not both in the same kernel. Note that the same image can
be `read-only` in one kernel and `write-only` in another, as is the case with the `luminance` image in our example.

The maximum luminance filtering kernel is arguably simplified by our use of images. We no longer need to perform manual clamping, since the call to `read_imageui` uses the clamping sampler. One detail that bears explanation is the `ui` suffix on the read/write methods, which causes the data at the given coordinate location to be treated as unsigned integer data. There are also variants for reading float data and signed integer data, as detailed in the OpenCL specification.

### 4.4.2 Memory Changes

The code for creating and initializing the needed memory objects is given in Listing 4.9. Though it’s quite similar to the buffer-based code, there are a few differences.

```c
unsigned int *data; // ... read in from disk during init ...
cl_mem original, result, luminance;
static int initialize_memory() {
    cl_int err;

    // read in orig image:
    int size = IMAGE_WIDTH * IMAGE_HEIGHT;
    cl_image_format format;
    format.image_channel_order = CL_RGBA;
    format.image_channel_data_type = CL_UNSIGNED_INT8;

    original = clCreateImage2D( context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, 
                               &format, IMAGE_WIDTH, IMAGE_HEIGHT, IMAGE_WIDTH*4, data, &err);
    result = clCreateImage2D( context, CL_MEM_WRITE_ONLY, &format, IMAGE_WIDTH, 
                              IMAGE_HEIGHT, 0, NULL, &err);
    luminance = clCreateImage2D( context, CL_MEM_READ_WRITE, &format, IMAGE_WIDTH, 
                                 IMAGE_HEIGHT, 0, NULL, &err);

    if (!original || !result || !luminance || err != CL_SUCCESS)
        error_and_exit("clCreateImage2d", err);

    return CL_SUCCESS;
}
```

Listing 4.9: Excerpt from `cl_images/main.c` – image object creation.

Note that we’re assuming that the C integer array `data` has been populated with our image data before the call to `initialize_memory`. Lines 10-12 declare and initialize an image format structure, used to tell OpenCL about the type of image data we’re using. Three calls to `clCreateImage2D` follow, each of which tells OpenCL to allocate space on the device for an `IMAGE_WIDTH`-by-`IMAGE_HEIGHT` image memory object containing four channel RGBA pixels with 8-bits per pixel. The call that creates the image object `original` is unique in that it uses the memory flag `CL_MEM_COPY_HOST_PTR`. This flag instructs OpenCL to use the buffer specified in the second to last parameter as the initial data for the image. Effectively, it avoids explicitly writing the data with a subsequent call to `clEnqueueWriteImage`. Note that while OpenCL guarantees the data will be on the device before the first kernel that uses it runs, it might not copy it right away. You can have that behavior – if you’d like to free the host side data buffer, for example – by using a blocking `WriteImage` call. This is similar to our population of the buffer memory object back in Listing 4.5.

### 4.4.3 Performance

Ameliorating the memory bottleneck dramatically improves performance. Our 500 pass filtering test now runs in 5.1 seconds, on average. This is more than a 33x improvement over the original pure CPU implementation!
This is a powerful demonstration of how a knowledge of the underlying hardware is critical for maximum performance.

4.5 Sharing with OpenGL

To ground our test of repeated filter application in something practically useful, this section discusses an optional portion of the OpenCL specification: CL/GL sharing. This is a mechanism – with platform dependent components – by which OpenCL can read and write directly from OpenGL objects like textures and vertex buffer objects. This section presumes that the reader is familiar with these OpenGL structures, and has a working knowledge of GLUT. Note that due to the platform-specific nature of this feature, the examples in this section will only work using Apple’s OpenCL implementation. The concepts, however, are applicable across vendors.

4.5.1 Setting the Stage

The code listings in this section are from a simple GLUT-based program (provided in the clgl folder) which displays our 512x512 test image after processing by our maximum luminance kernel. The user can change the filter width using keyboard controls. The image is displayed by simply rendering a textured quad. The bound texture is given no initial data during creation (Listing 4.10), but is instead treated as an OpenCL image object and written in a kernel that runs once per frame.

```c
GLuint tex; // global
void setupTextures() {
    glGenTextures(1, &tex);
    glBindTexture(GL_TEXTURE_2D, tex);

    glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, TWIDE, THIGH, 0, GL_RGBA,
                 GL_UNSIGNED_BYTE, NULL);
    glBindTexture(GL_TEXTURE_2D, 0);
}
```

Listing 4.10: Excerpt from clgl/main.c – OpenGL texture creation.

This is standard OpenGL texture creation. Note that we do not provide the actual texture data in the call to `glTexImage2D` on line 11. The pixels for the texture instead come as the output of our filtering kernel, since this texture will be shared with OpenCL as an image memory object.

4.5.2 Context Creation

Sharing GL objects with CL requires surprisingly few changes to our previous image-based code. The first – and most significant – change is how we create the CL context, showing in Listing 4.11. We’ve removed error checking from this listing for compactness, but it is included in the actual source code. While this code differs somewhat from the previous examples, the basic logic flow is the same: obtain a context, choose a device, then create a command queue.

Lines 5-10 are implementation specific. On Apple’s platform, we create a `cl_context_properties` object containing a `CGLShareGroupObj`, a type defined in `CGLDevice.h` in the OpenGL framework. This header, as well as all other needed for GL/CL sharing on OS X, is automatically included if you include `OpenCL/OpenCL.h` in lieu of `OpenCL/cl.h`. The underlying concept here is that the CL context is created using a reference to the existing GL context. Regardless of whether you are using GLUT or Qt or Cocoa or something else entirely, you craft a `cl_context_properties` that has a reference to the current GL context.
void setupDevice() {
    cl_int numDevices, device_type, i;
    size_t ret_size;

    CGLContextObj glcontext = CGLGetCurrentContext();
    CGLShareGroupObj sharegroup = CGLGetShareGroup(glcontext);
    cl_context_properties properties[] = {
        CL_CONTEXT_PROPERTY_USE_CGL_SHAREGROUP_APPLE,
        (cl_context_properties)sharegroup, 0
    };

    context = clCreateContext(properties, 0, 0, 0, 0, 0);
    clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &ret_size);
    numDevices = ret_size / sizeof(cl_device_id);
    clGetContextInfo(context, CL_CONTEXT_DEVICES, ret_size, devices, &ret_size);

    for (i = 0; i < numDevices; i++) {
        clGetDeviceInfo(devices[i], CL_DEVICE_TYPE, sizeof(cl_device_type),
                         &device_type, &ret_size);
        if (device_type == CL_DEVICE_TYPE_GPU) {
            device_id = devices[i];
            commands = clCreateCommandQueue(context, device_id, 0, NULL);
            break;
        }
    }
}

Listing 4.11: Excerpt from clgl/main.c – Context creation.

Line 12 actually creates the context with a call to clCreateContext. Notice that we provide no information beyond the previously created properties object; the runtime and driver can tease out enough information from the GL context to configure everything correctly. Armed with this context, we can now ask OpenCL about the available devices. The for loop beginning on line 20 iterates over these devices looking for a GPU device. Once we find it, we create our command queue (lines 26). Note that a system containing more than one GPU device might want to check other device properties using clGetDeviceInfo as part of the device selection. But we have only one GPU in our test system, so the logic above works fine.

4.5.3 Memory Object Creation

The second needed change concerns the image object creation, as shown Listing 4.12. This code snippet is short because only one image object is affected: the final filtered image. Instead of a call to clCreateImage2D, we use clCreateFromGLTexture2D, supplying the OpenGL texture handle. Note that we do not specify a size or format, since the system already knows these details from the initial texture creation.

    filtered = clCreateFromGLTexture2D(context, CL_MEM_WRITE_ONLY, GL_TEXTURE_2D, 0,
                                         tex, &err);

Listing 4.12: Excerpt from clgl/main.c – Image memory object creation from an OpenGL texture.

That’s it. We now have an OpenCL image object that is backed by the previously-allocated OpenGL texture. When we write to this image in a kernel, we directly affect the pixel data associated with the texture. Note that as with “normal” OpenCL images, we have arbitrary scatter ability.

4.5.4 Kernel Execution

The final needed change is simple: we wrap calls to kernel execution in code that acquires and releases the shared objects. This allows the graphics driver to defer any use of the shared objects until the data needed
for display is complete and correct. This makes sense; we wouldn’t want OpenGL to apply the texture before the kernel was done computing its pixel data.

```c
err = clEnqueueNDRangeKernel(commands, kernel.luminance, 2, NULL, gdim, NULL, 0, NULL, NULL);
err = clEnqueueAcquireGLObjects(commands, 1, &filtered, 0, NULL, NULL);
err = clEnqueueNDRangeKernel(commands, kernel.filter, 2, NULL, gdim, NULL, 0, NULL, &event);
err = clWaitForEvents(1, &event);
err = clEnqueueReleaseGLObjects(commands, 1, &filtered, 0, NULL, NULL);
```

In the above listing, we first execute the luminance calculation kernel, which does not make use of shared objects. We then acquire the shared texture/image object, execute the filter kernel, wait for completion using an OpenCL event, and then release the shared object. The call to `clWaitForEvents` is not required, since the we’re using an in-order command queue. We found in testing, however, that adding this explicit wait resulted in smoother window-dragging performance on our implementation. Your mileage may vary.

### 4.5.5 Performance

The screenshot in figure 4.2 tells the tale. With a filter width of 25 (performing 625 memory reads per pixel per frame), our test application runs at a respectable 90 fps on a $512^2$ input image.

### 4.5.6 Bonus Round: VBO Sharing

The previous sections show that sharing OpenGL textures with OpenCL isn’t particularly difficult. Fortunately for those interested in procedural generation of vertices and vertex attributes, sharing vertex buffer objects is just as easy. The downloadable examples include `cl_hilbert`, a GLUT-based program that uses OpenCL to compute the 2D x,y hilbert curve mapping based on the global work item ids. It arose out of
a need for fast calculation of Hilbert curves, and is a barebones example of the aforementioned procedural
generation. The basic steps of the Hilbert example are as follows:

1. Initialize GLUT and your VBOs as usual. Just as in the texture example, no special setup is required
   on the OpenGL side.

2. Create your OpenCL context and command queue from the existing GL context, exactly as in the
   previous example.

3. Create your OpenCL buffer objects using clCreateFromGLBuffer.

4. Compile your CL program and reference kernels as normal.

5. Execute your kernel as normal, but wrapped with calls to acquire and release the shared objects, as in
   the previous example.

Listing 4.14 shows the VBO creation, which is entirely typical save for one minor point. Notice that
the calls to glVertexPointer uses the flag GL_STATIC_DRAW. This might seem strange since we’re going to be
procédurally generating both vertex and color data in these buffers per frame; this is decidedly dynamic and
one might think that GL_DYNAMIC_DRAW is appropriate. Yet from OpenGL’s perspective, the buffer is indeed
static – it’s OpenCL that’s doing the dynamic work.


Listing 4.15 shows the OpenCL side of buffer creation. It’s very similar to how we created image objects
from textures in the previous example.

Listing 4.15: Excerpt from clhilbert/main.c – Buffer Creation.

Figure 4.3 shows a screen capture from the hilbert test code. In this example, we’re generating a
Hilbert curve using a GL_LINE_STRIP containing 65,536 vertices and accompanying colors once per frame
entirely on the GPU. No vertex data ever crosses the graphics bus, and we achieve a framerate of more than
700Hz. While this example is somewhat contrived, it’s clear that OpenCL is suitable for dynamic, GPU-side
generation of vertex data. Apple offers a well-crafted, sophisticated code example for procedural grass and
terrain generation on their developer website (bit.ly/9mGiyl). While their example shows the power of
OpenCL/GL generation in a enjoyably spectacular fashion, it does not use any code tricks beyond
those given in our Hilbert curve example.
4.6 Concluding Remarks

We walked though the steps necessary to get a basic CL program up and running using both buffers and image objects, highlighting how images can sometimes help performance. Hopefully, we’ve shown that sharing data between OpenCL and OpenGL is easy, and given your enough knowledge to craft some useful code.

We’d encourage readers to experiment with the included sample code until they feel comfortable with CL basics, and then move onto the topics of local memory and architecture-dependent optimization. The NVIDIA guides mentioned in Section 3.6 would be a great place to start if you’re working with NVIDIA hardware.
Bibliography


