PEARSON

国外信息科学与技术优秀图书系列　计算机科学与技术

OpenCL 编程指南

OpenCL Programming Guide

（英文版）

〔美〕Aaftab Munshi　Benedict R. Gaster
Timothy G. Mattson　James Fung　著
Dan Ginsburg

科学出版社

信息科学与技术优秀图书系列

OpenCL 编程指南

（英文版）

OpenCL Programming Guide

Aaftab Munshi　Benedict R. Gaster
〔美〕　Timothy G. Mattson　James Fung
Dan Ginsburg

科学出版社

北京

图字：01-2012-1770

内 容 简 介

新的 OpenCL 标准有助于充分利用 CPU、GPU 等处理器的丰富资源，已获得 Apple、AMD、Intel、IBM 等公司的认可，在服务器、嵌入式设备、高性能计算等领域有广阔的应用前景。

本书由 OpenCL 的五大技术权威共同撰写，内容涵盖完整的规范。在分析关键用户案例的基础上，说明了如何用 OpenCL 表示各类并行算法，并且提供了完整的 API 和 OpenCL C 语言的参考信息。通过完整的案例学习和代码示例，讲解了编写复杂并行程序的方法，实现在众多不同设备间分解工作量，还介绍了 OpenCL 软件性能优化的要点。

本书是第一本针对 OpenCL 1.1 规范的全面、权威的实践指南，适合信息技术领域的研发人员和软件架构师阅读参考。

图书在版编目(CIP)数据

OpenCL 编程指南 : 英文 /(美) 曼什 (Munshi, A.) 等著. —影印本. —北京：科学出版社，2012

(国外信息科学与技术优秀图书系列)

ISBN 978-7-03-034963-7

Ⅰ. ①O… Ⅱ. ①曼… Ⅲ. ①图形软件-程序设计-英文 Ⅳ. ①TP391.41

中国版本图书馆 CIP 数据核字(2012)第 132813 号

责任编辑：任 静 / 责任印制：张 倩 / 封面设计：刘可红

科 学 出 版 社 出版

北京东黄城根北街16号

邮政编码：100717

http://www.sciencep.com

双 青 印 刷 厂 印刷

科学出版社发行 各地新华书店经销

*

2012 年 7 月第 一 版 开本：B5(720×1000)

2012 年 7 月第一次印刷 印张： 39 1/4

字数：964 000

定价：98.00 元

（如有印装质量问题，我社负责调换）

Foreword

During the past few years, heterogeneous computers composed of CPUs and GPUs have revolutionized computing. By matching different parts of a workload to the most suitable processor, tremendous performance gains have been achieved.

Much of this revolution has been driven by the emergence of many-core processors such as GPUs. For example, it is now possible to buy a graphics card that can execute more than a trillion floating point operations per second (teraflops). These GPUs were designed to render beautiful images, but for the right workloads, they can also be used as high-performance computing engines for applications from scientific computing to augmented reality.

A natural question is why these many-core processors are so fast compared to traditional single core CPUs. The fundamental driving force is innovative parallel hardware. Parallel computing is more efficient than sequential computing because chips are fundamentally parallel. Modern chips contain billions of transistors. Many-core processors organize these transistors into many parallel processors consisting of hundreds of floating point units. Another important reason for their speed advantage is new parallel software. Utilizing all these computing resources requires that we develop parallel programs. The efficiency gains due to software and hardware allow us to get more FLOPs per Watt or per dollar than a single-core CPU.

Computing systems are a symbiotic combination of hardware and software. Hardware is not useful without a good programming model. The success of CPUs has been tied to the success of their programming models, as exemplified by the C language and its successors. C nicely abstracts a sequential computer. To fully exploit heterogeneous computers, we need new programming models that nicely abstract a modern *parallel* computer. And we can look to techniques established in graphics as a guide to the new programming models we need for heterogeneous computing.

I have been interested in programming models for graphics for many years. It started in 1988 when I was a software engineer at PIXAR, where I developed the RenderMan shading language. A decade later graphics

systems became fast enough that we could consider developing shading languages for GPUs. With Kekoa Proudfoot and Bill Mark, we developed a real-time shading language, RTSL. RTSL ran on graphics hardware by compiling shading language programs into pixel shader programs, the assembly language for graphics hardware of the day. Bill Mark subsequently went to work at NVIDIA, where he developed Cg. More recently, I have been working with Tim Foley at Intel, who has developed a new shading language called Spark. Spark takes shading languages to the next level by abstracting complex graphics pipelines with new capabilities such as tesselation.

While developing these languages, I always knew that GPUs could be used for much more than graphics. Several other groups had demonstrated that graphics hardware could be used for applications beyond graphics. This led to the GPGPU (General-Purpose GPU) movement. The demonstrations were hacked together using the graphics library. For GPUs to be used more widely, they needed a more general programming environment that was not tied to graphics. To meet this need, we started the Brook for GPU Project at Stanford. The basic idea behind Brook was to treat the GPU as a data-parallel processor. Data-parallel programming has been extremely successful for parallel computing, and with Brook we were able to show that data-parallel programming primitives could be implemented on a GPU. Brook made it possible for a developer to write an application in a widely used parallel programming model.

Brook was built as a proof of concept. Ian Buck, a graduate student at Stanford, went on to NVIDIA to develop CUDA. CUDA extended Brook in important ways. It introduced the concept of cooperating thread arrays, or thread blocks. A cooperating thread array captured the locality in a GPU core, where a block of threads executing the same program could also communicate through local memory and synchronize through barriers. More importantly, CUDA created an environment for GPU Computing that has enabled a rich ecosystem of application developers, middleware providers, and vendors.

OpenCL (Open Computing Language) provides a logical extension of the core ideas from GPU Computing—the era of ubiquitous heterogeneous parallel computing. OpenCL has been carefully designed by the Khronos Group with input from many vendors and software experts. OpenCL benefits from the experience gained using CUDA in creating a software standard that can be implemented by many vendors. OpenCL implementations run now on widely used hardware, including CPUs and GPUs from NVIDIA, AMD, and Intel, as well as platforms based on DSPs and FPGAs.

By standardizing the programming model, developers can count on more software tools and hardware platforms.

What is most exciting about OpenCL is that it doesn't only standardize what has been done, but represents the efforts of an active community that is pushing the frontier of parallel computing. For example, OpenCL provides innovative capabilities for scheduling tasks on the GPU. The developers of OpenCL have have combined the best features of task-parallel and data-parallel computing. I expect future versions of OpenCL to be equally innovative. Like its father, OpenGL, OpenCL will likely grow over time with new versions with more and more capability.

This book describes the complete OpenCL Programming Model. One of the coauthors, Aaftab, was the key mind behind the system. He has joined forces with other key designers of OpenCL to write an accessible authoritative guide. Welcome to the new world of heterogeneous computing.

—Pat Hanrahan
Stanford University

Preface

Industry pundits love drama. New products don't build on the status quo to make things better. They "revolutionize" or, better yet, define a "new paradigm." And, of course, given the way technology evolves, the results rarely are as dramatic as the pundits make it seem.

Over the past decade, however, something revolutionary has happened. The drama is real. CPUs with multiple cores have made parallel hardware ubiquitous. GPUs are no longer *just* specialized graphics processors; they are heavyweight compute engines. And their combination, the so-called heterogeneous platform, truly is redefining the standard building blocks of computing.

We appear to be midway through a revolution in computing on a par with that seen with the birth of the PC. Or more precisely, we have the *potential* for a revolution because the high levels of parallelism provided by heterogeneous hardware are meaningless without parallel software; and the fact of the matter is that outside of specific niches, parallel software is rare.

To create a parallel software revolution that keeps pace with the ongoing (parallel) heterogeneous computing revolution, we need a parallel software industry. That industry, however, can flourish only if software can move between platforms, both cross-vendor and cross-generational. The solution is an industry standard for heterogeneous computing.

OpenCL is that industry standard. Created within the Khronos Group (known for OpenGL and other standards), OpenCL emerged from a collaboration among software vendors, computer system designers (including designers of mobile platforms), and microprocessor (embedded, accelerator, CPU, and GPU) manufacturers. It is an answer to the question "How can a person program a heterogeneous platform with the confidence that software created today will be relevant tomorrow?"

Born in 2008, OpenCL is now available from multiple sources on a wide range of platforms. It is evolving steadily to remain aligned with the latest microprocessor developments. In this book we focus on OpenCL 1.1. We describe the full scope of the standard with copious examples to explain how OpenCL is used in practice. Join us. *Vive la révolution.*

Intended Audience

This book is written by programmers for programmers. It is a pragmatic guide for people interested in writing code. We assume the reader is comfortable with C and, for parts of the book, C++. Finally, we assume the reader is familiar with the basic concepts of parallel programming. We assume our readers have a computer nearby so they can write software and explore ideas as they read. Hence, this book is overflowing with programs and fragments of code.

We cover the entire OpenCL 1.1 specification and explain how it can be used to express a wide range of parallel algorithms. After finishing this book, you will be able to write complex parallel programs that decompose a workload across multiple devices in a heterogeneous platform. You will understand the basics of performance optimization in OpenCL and how to write software that probes the hardware and adapts to maximize performance.

Organization of the Book

The OpenCL specification is almost 400 pages. It's a dense and complex document full of tediously specific details. Explaining this specification is not easy, but we think that we've pulled it off nicely.

The book is divided into two parts. The first describes the OpenCL specification. It begins with two chapters to introduce the core ideas behind OpenCL and the basics of writing an OpenCL program. We then launch into a systematic exploration of the OpenCL 1.1 specification. The tone of the book changes as we incorporate reference material with explanatory discourse. The second part of the book provides a sequence of case studies. These range from simple pedagogical examples that provide insights into how aspects of OpenCL work to complex applications showing how OpenCL is used in serious application projects. The following provides more detail to help you navigate through the book:

Part I: The OpenCL 1.1 Language and API

- **Chapter 1, "An Introduction to OpenCL"**: This chapter provides a high-level overview of OpenCL. It begins by carefully explaining why heterogeneous parallel platforms are destined to dominate computing into the foreseeable future. Then the core models and concepts behind OpenCL are described. Along the way, the terminology used in OpenCL is presented, making this chapter an important one to read

even if your goal is to skim through the book and use it as a reference guide to OpenCL.

- **Chapter 2, "HelloWorld: An OpenCL Example"**: Real programmers learn by writing code. Therefore, we complete our introduction to OpenCL with a chapter that explores a working OpenCL program. It has become standard to introduce a programming language by printing "hello world" to the screen. This makes no sense in OpenCL (which doesn't include a print statement). In the data-parallel programming world, the analog to "hello world" is a program to complete the element-wise addition of two arrays. That program is the core of this chapter. By the end of the chapter, you will understand OpenCL well enough to start writing your own simple programs. And we urge you to do exactly that. You can't learn a programming language by reading a book alone. Write code.
- **Chapter 3, "Platforms, Contexts, and Devices"**: With this chapter, we begin our systematic exploration of the OpenCL specification. Before an OpenCL program can do anything "interesting," it needs to discover available resources and then prepare them to do useful work. In other words, a program must discover the platform, define the context for the OpenCL program, and decide how to work with the devices at its disposal. These important topics are explored in this chapter, where the OpenCL Platform API is described in detail.
- **Chapter 4, "Programming with OpenCL C"**: Code that runs on an OpenCL device is in most cases written using the OpenCL C programming language. Based on a subset of C99, the OpenCL C programming language provides what a kernel needs to effectively exploit an OpenCL device, including a rich set of vector instructions. This chapter explains this programming language in detail.
- **Chapter 5, "OpenCL C Built-In Functions"**: The OpenCL C programming language API defines a large and complex set of built-in functions. These are described in this chapter.
- **Chapter 6, "Programs and Kernels"**: Once we have covered the languages used to write kernels, we move on to the runtime API defined by OpenCL. We start with the process of creating programs and kernels. Remember, the word *program* is overloaded by OpenCL. In OpenCL, the word *program* refers specifically to the "dynamic library" from which the functions are pulled for the kernels.
- **Chapter 7, "Buffers and Sub-Buffers"**: In the next chapter we move to the buffer memory objects, one-dimensional arrays, including a careful discussion of sub-buffers. The latter is a new feature in

OpenCL 1.1, so programmers experienced with OpenCL 1.0 will find this chapter particularly useful.

- **Chapter 8, "Images and Samplers"**: Next we move to the very important topic of our other memory object, images. Given the close relationship between graphics and OpenCL, these memory objects are important for a large fraction of OpenCL programmers.
- **Chapter 9, "Events"**: This chapter presents a detailed discussion of the event model in OpenCL. These objects are used to enforce ordering constraints in OpenCL. At a basic level, events let you write concurrent code that generates correct answers regardless of how work is scheduled by the runtime. At a more algorithmically profound level, however, events support the construction of programs as directed acyclic graphs spanning multiple devices.
- **Chapter 10, "Interoperability with OpenGL"**: Many applications may seek to use graphics APIs to display the results of OpenCL processing, or even use OpenCL to postprocess scenes generated by graphics. The OpenCL specification allows interoperation with the OpenGL graphics API. This chapter will discuss how to set up OpenGL/OpenCL sharing and how data can be shared and synchronized.
- **Chapter 11, "Interoperability with Direct3D"**: The Microsoft family of platforms is a common target for OpenCL applications. When applications include graphics, they may need to connect to Microsoft's native graphics API. In OpenCL 1.1, we define how to connect an OpenCL application to the DirectX 10 API. This chapter will demonstrate how to set up OpenCL/Direct3D sharing and how data can be shared and synchronized.
- **Chapter 12, "C++ Wrapper API"**: We then discuss the OpenCL C++ API Wrapper. This greatly simplifies the host programs written in C++, addressing automatic reference counting and a unified interface for querying OpenCL object information. Once the C++ interface is mastered, it's hard to go back to the regular C interface.
- **Chapter 13, "OpenCL Embedded Profile"**: OpenCL was created for an unusually wide range of devices, with a reach extending from cell phones to the nodes in a massively parallel supercomputer. Most of the OpenCL specification applies without modification to each of these devices. There are a small number of changes to OpenCL, however, needed to fit the reduced capabilities of low-power processors used in embedded devices. This chapter describes these changes, referred to in the OpenCL specification as the OpenCL embedded profile.

Part II: OpenCL 1.1 Case Studies

- **Chapter 14, "Image Histogram":** A histogram reports the frequency of occurrence of values within a data set. For example, in this chapter, we compute the histogram for R, G, and B channel values of a color image. To generate a histogram in parallel, you compute values over local regions of a data set and then sum these local values to generate the final result. The goal of this chapter is twofold: (1) we demonstrate how to manipulate images in OpenCL, and (2) we explore techniques to efficiently carry out a histogram's global summation within an OpenCL program.
- **Chapter 15, "Sobel Edge Detection Filter":** The Sobel edge filter is a directional edge detector filter that computes image gradients along the *x*- and *y*-axes. In this chapter, we use a kernel to apply the Sobel edge filter as a simple example of how kernels work with images in OpenCL.
- **Chapter 16, "Parallelizing Dijkstra's Single-Source Shortest-Path Graph Algorithm":** In this chapter, we present an implementation of Dijkstra's Single-Source Shortest-Path graph algorithm implemented in OpenCL capable of utilizing both CPU and multiple GPU devices. Graph data structures find their way into many problems, from artificial intelligence to neuroimaging. This particular implementation was developed as part of FreeSurfer, a neuroimaging application, in order to improve the performance of an algorithm that measures the curvature of a triangle mesh structural reconstruction of the cortical surface of the brain. This example is illustrative of how to work with multiple OpenCL devices and split workloads across CPUs, multiple GPUs, or all devices at once.
- **Chapter 17, "Cloth Simulation in the Bullet Physics SDK":** Physics simulation is a growing addition to modern video games, and in this chapter we present an approach to simulating cloth, such as a warrior's clothing, using OpenCL that is part of the Bullet Physics SDK. There are many ways of simulating soft bodies; the simulation method used in Bullet is similar to a mass/spring model and is optimized for execution on modern GPUs while integrating smoothly with other Bullet SDK components that are not written in OpenCL. We show an important technique, called batching, that transforms the particle meshes for performant execution on wide SIMD architectures, such as the GPU, while preserving dependences within the mass/spring model.

- **Chapter 18, "Simulating the Ocean with Fast Fourier Transform"**: In this chapter we present the details of AMD's Ocean simulation. Ocean is an OpenCL demonstration that uses an inverse discrete Fourier transform to simulate, in real time, the sea. The fast Fourier transform is applied to random noise, generated over time as a frequency-dependent phase shift. We describe an implementation based on the approach originally developed by Jerry Tessendorf that has appeared in a number of feature films, including *Waterworld*, *Titanic*, and *Fifth Element*. We show the development of an optimized 2D DFFT, including a number of important optimizations useful when programming with OpenCL, and the integration of this algorithm into the application itself and using interoperability between OpenCL and OpenGL.
- **Chapter 19, "Optical Flow"**: In this chapter, we present an implementation of optical flow in OpenCL, which is a fundamental concept in computer vision that describes motion in images. Optical flow has uses in image stabilization, temporal upsampling, and as an input to higher-level algorithms such as object tracking and gesture recognition. This chapter presents the pyramidal Lucas-Kanade optical flow algorithm in OpenCL. The implementation demonstrates how image objects can be used to access texture features of GPU hardware. We will show how the texture-filtering hardware on the GPU can be used to perform linear interpolation of data, achieve the required sub-pixel accuracy, and thereby provide significant speedups. Additionally, we will discuss how shared memory can be used to cache data that is repeatedly accessed and how early kernel exit techniques provide additional efficiency.
- **Chapter 20, "Using OpenCL with PyOpenCL"**: The purpose of this chapter is to introduce you to the basics of working with OpenCL in Python. The majority of the book focuses on using OpenCL from C/C++, but bindings are available for other languages including Python. In this chapter, PyOpenCL is introduced by walking through the steps required to port the Gaussian image-filtering example from Chapter 8 to Python. In addition to covering the changes required to port from C++ to Python, the chapter discusses some of the advantages of using OpenCL in a dynamically typed language such as Python.
- **Chapter 21, "Matrix Multiplication with OpenCL"**: In this chapter, we discuss a program that multiplies two square matrices. The program is very simple, so it is easy to follow the changes made to the program as we optimize its performance. These optimizations focus

on the OpenCL memory model and how we can work with the model to minimize the cost of data movement in an OpenCL program.

- **Chapter 22, "Sparse Matrix-Vector Multiplication"**: In this chapter, we describe an optimized implementation of the Sparse Matrix-Vector Multiplication algorithm using OpenCL. Sparse matrices are defined as large, two-dimensional matrices in which the vast majority of the elements of the matrix are equal to zero. They are used to characterize and solve problems in a wide variety of domains such as computational fluid dynamics, computer graphics/vision, robotics/kinematics, financial modeling, acoustics, and quantum chemistry. The implementation demonstrates OpenCL's ability to bridge the gap between hardware-specific code (fast, but not portable) and single-source code (very portable, but slow), yielding a high-performance, efficient implementation on a variety of hardware that is almost as fast as a hardware-specific implementation. These results are accomplished with kernels written in OpenCL C that can be compiled and run on any conforming OpenCL platform.

Appendix

- **Appendix A, "Summary of OpenCL 1.1"**: The OpenCL specification defines an overwhelming collection of functions, named constants, and types. Even expert OpenCL programmers need to look up these details when writing code. To aid in this process, we've included an appendix where we pull together all these details in one place.

Example Code

This book is filled with example programs. You can download many of the examples from the book's Web site at www.openclprogrammingguide.com.

Errata

If you find something in the book that you believe is in error, please send us a note at errors@opencl-book.com. The list of errata for the book can be found on the book's Web site at www.openclprogrammingguide.com.

Contents