Quick answers to common problems

OpenCL Parallel Programming Development Cookbook

Accelerate your applications and understand high-performance computing with over 50 OpenCL recipes

Raymond Tay

[PACKT] PUBLISHING

www.EngineeringBooksPdf.com

OpenCL Parallel Programming Development Cookbook

Accelerate your applications and understand high-performance computing with over 50 OpenCL recipes

Raymond Tay

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

www.EngineeringBooksPdf.com

OpenCL Parallel Programming Development Cookbook

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: August 2013

Production Reference: 1210813

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84969-452-0

www.packtpub.com

Cover Image by Suresh Mogre (suresh.mogre.99@gmail.com)

www.EngineeringBooksPdf.com

Credits

Author

Raymond Tay

Reviewers

Nitesh Bhatia

Darryl Gove

Seyed Hadi Hosseini

Kyle Lutz

Viraj Paropkari

Acquisition Editors

Saleem Ahmed

Erol Staveley

Lead Technical Editor

Ankita Shashi

Technical Editors

Veena Pagare

Krishnaveni Nair

Ruchita Bhansali

Shali Sashidharan

Project Coordinator

Shiksha Chaturvedi

Proofreader

Faye Coulman

Lesley Harrison

Paul Hindle

Indexer

Tejal R. Soni

Graphics

Sheetal Aute

Ronak Druv

Valentina D'silva

Disha Haria

Abhinash Sahu

Production Coordinator

Melwyn D'sa

Cover Work

Melwyn D'sa

www.EngineeringBooksPdf.com

About the Author

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

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

---

www.EngineeringBooksPdf.com

About the Reviewers

Nitesh Bhatia is a tech geek with a background in **information and communication technology** (**ICT**) with an emphasis on computing and design research. He worked with Infosys Design as a user experience designer, and is currently a doctoral scholar at the Indian Institute of Science, Bangalore. His research interests include visual computing, digital human modeling, and applied ergonomics. He delights in exploring different programming languages, computing platforms, embedded systems and so on. He is a founder of several social media startups. In his leisure time, he is an avid photographer and an art enthusiast, maintaining a compendium of his creative works through his blog Dangling-Thoughts (`http://www.dangling-thoughts.com`).

Darryl Gove is a senior principal software engineer in the Oracle Solaris Studio team, working on optimizing applications and benchmarks for current and future processors. He is also the author of the books, *Multicore Application Programming*, *Solaris Application Programming*, and *The Developer's Edge*. He writes his blog at `http://www.darrylgove.com`.

Seyed Hadi Hosseini is a software developer and network specialist, who started his career at the age of 16 by earning certifications such as MCSE, CCNA, and Security+. He decided to pursue his career in Open Source Technology, and for this Perl programming was the starting point. He concentrated on web technologies and software development for almost 10 years. He is also an instructor of open source courses. Currently, Hadi is certified by the Linux Professional Institute, Novell, and CompTIA as a Linux specialist (LPI, LINUX+, NCLA and DCTS). High Performance Computing is one of his main research areas. His first published scientific paper was awarded as the best article in the fourth Iranian Bioinformatics Conference held in 2012. In this article, he developed a super-fast processing algorithm for SSR in Genome and proteome datasets, by using OpenCL as the GPGPU programming framework in C language, and benefiting from the massive computing capability of GPUs.

Special thanks to my family and grandma for their invaluable support.
I would also like to express my sincere appreciation to my wife, without her support and patience, this work would not have been done easily.

---

www.EngineeringBooksPdf.com

Kyle Lutz is a software engineer and is a part of the Scientific Computing team at Kitware, Inc, New York. He holds a bachelor's degree in Biological Sciences from the University of California at Santa Barbara. He has several years of experience writing scientific simulation, analysis, and visualization software in C++ and OpenCL. He is also the lead developer of the `Boost.Compute` library – a C++ GPU/parallel-computing library based on OpenCL.

Viraj Paropkari has done his graduation in computer science from University of Pune, India, in 2004, and MS in computer science from Georgia Institute of Technology, USA, in 2008. He is currently a senior software engineer at **Advanced Micro Devices** (**AMD**), working on performance optimization of applications on CPUs, GPUs using OpenCL. He also works on exploring new challenges in big data and **High Performance Computing** (**HPC**) applications running on large scale distributed systems. Previously, he was systems engineer at **National Energy Research Scientific Computing Center** (**NERSC**) for two years, where he worked on one of the world's largest supercomputers running and optimizing scientific applications. Before that, he was a visiting scholar in **Parallel Programming Lab** (**PPL**) at Computer Science Department of University of Illinois, Urbana-Champaign, and also a visiting research scholar at Oak Ridge National Laboratory, one of the premier research labs in U.S.A. He also worked on developing software for mission critical flight simulators at Indian Institute of technology, Bombay, India, and **Tata institute of Fundamental Research** (**TIFR**), India. He was the main contributor of the team that was awarded the HPC Innovation Excellence Award to speed up the CFD code and achieve the first ever simulation of a realistic fuel-spray related application. The ability to simulate this problem helps reduce design cycles to at least 66 percent and provides new insights into the physics that can provide sprays with enhanced properties.

I'd like to thank my parents, who have been inspiration to me and also thank my beloved wife, Anuya, who encouraged me in spite of all the time it took me away from her.

---

www.EngineeringBooksPdf.com

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

www.EngineeringBooksPdf.com

www.EngineeringBooksPdf.com

Table of Contents

Preface 1

Chapter 1: Using OpenCL 7

Introduction 7
Querying OpenCL platforms 14
Querying OpenCL devices on your platform 18
Querying OpenCL device extensions 22
Querying OpenCL contexts 25
Querying an OpenCL program 29
Creating OpenCL kernels 35
Creating command queues and enqueuing OpenCL kernels 38

Chapter 2: Understanding OpenCL Data Transfer and Partitioning 43

Introduction 43
Creating OpenCL buffer objects 44
Retrieving information about OpenCL buffer objects 50
Creating OpenCL sub-buffer objects 54
Retrieving information about OpenCL sub-buffer objects 58
Understanding events and event synchronization 61
Copying data between memory objects 64
Using work items to partition data 71

Chapter 3: Understanding OpenCL Data Types 79

Introduction 79
Initializing the OpenCL scalar data types 80
Initializing the OpenCL vector data types 82
Using OpenCL scalar types 85
Understanding OpenCL vector types 88
Vector and scalar address spaces 100
Configuring your OpenCL projects to enable the double data type 103

www.EngineeringBooksPdf.com

Chapter 4: Using OpenCL Functions 109
Introduction 109
Storing vectors into an array 110
Loading vectors from an array 114
Using geometric functions 117
Using integer functions 120
Using floating-point functions 123
Using trigonometric functions 126
Arithmetic and rounding in OpenCL 129
Using the shuffle function in OpenCL 132
Using the select function in OpenCL 135

Chapter 5: Developing a Histogram OpenCL program 139
Introduction 139
Implementing a Histogram in C/C++ 139
OpenCL implementation of the Histogram 142
Work item synchronization 153

Chapter 6: Developing a Sobel Edge Detection Filter 155
Introduction 155
Understanding the convolution theory 156
Understanding convolution in 1D 157
Understanding convolution in 2D 159
OpenCL implementation of the Sobel edge filter 162
Understanding profiling in OpenCL 168

Chapter 7: Developing the Matrix Multiplication with OpenCL 173
Introduction 173
Understanding matrix multiplication 174
OpenCL implementation of the matrix multiplication 178
Faster OpenCL implementation of the matrix multiplication by thread coarsening 181
Faster OpenCL implementation of the matrix multiplication through register tiling 185
Reducing global memory via shared memory data prefetching in matrix multiplication 187

www.EngineeringBooksPdf.com

Chapter 8: Developing the Sparse-Matrix Vector Multiplication in OpenCL 193
Introduction 193
Solving SpMV (Sparse Matrix-Vector Multiplication) using the Conjugate Gradient Method 195
Understanding the various SpMV data storage formats including ELLPACK, ELLPACK-R, COO, and CSR 199
Understanding how to solve SpMV using the ELLPACK-R format 204
Understanding how to solve SpMV using the CSR format 208
Understanding how to solve SpMV using VexCL 216
Chapter 9: Developing the Bitonic Sort with OpenCL 221
Introduction 221
Understanding sorting networks 222
Understanding bitonic sorting 224
Developing bitonic sorting in OpenCL 230
Chapter 10: Developing the Radix Sort with OpenCL 241
Introduction 241
Understanding the Radix sort 242
Understanding the MSD and LSD Radix sorts 244
Understanding reduction 247
Developing the Radix sort in OpenCL 254
Index 281

www.EngineeringBooksPdf.com

www.EngineeringBooksPdf.com

Preface

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

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

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

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

www.EngineeringBooksPdf.com

What this book covers

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

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

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

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

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

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

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

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

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

www.EngineeringBooksPdf.com

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

What you need for this book

You need to be comfortable working in a Linux environment, as the examples are tested against the Ubuntu 12.10 64-bit operating system. The following are the requirements:

- GNU GCC C/C++ compiler Version 4.6.1 (at least)
- OpenCL 1.2 SDK by AMD, Intel & NVIDIA
- AMD APP SDK Version 2.8 with AMD Catalyst Linux Display Driver Version 13.4
- Intel OpenCL SDK 2012
- CMake Version 2.8 (at least)
- Clang Version 3.1 (at least)
- Microsoft Visual C++ 2010 (if you work on Windows)
- Boost Library Version 1.53
- VexCL (by Denis Demidov)
- CodeXL Profiler by AMD (Optional)
- At least eight hours of sleep
- An open and receptive mind
- A fresh brew of coffee or whatever that works

Who this book is for

This book is intended for software developers who have often wondered what to do with that newly bought CPU or GPU they bought other than using it for playing computer games. Having said that, this book isn't about toy algorithms that works only on your workstations at home. This book is ideally for the developers who have a working knowledge of C/C++ and who want to learn how to write parallel programs that execute in heterogeneous computing environments in OpenCL.

www.EngineeringBooksPdf.com

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We can include other contexts through the use of the `#include` directive."

A block of code is set as follows:

```
[default]
cl_uint sortOrder = 0; // descending order else 1 for ascending order
        cl_uint stages = 0;
        for(unsigned int i = LENGTH; i > 1; i >>= 1)
            ++stages;
        clSetKernelArg(kernel, 0, sizeof(cl_mem),(void*)&device_A_in);
        clSetKernelArg(kernel, 3, sizeof(cl_uint),(void*)&sortOrder);
#ifdef USE_SHARED_MEM
        clSetKernelArg(kernel, 4, (GROUP_SIZE << 1) *sizeof(cl_
uint),NULL);
#elif def USE_SHARED_MEM_2
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
[default]
cl_uint sortOrder = 0; // descending order else 1 for ascending order
        cl_uint stages = 0;
        for(unsigned int i = LENGTH; i > 1; i >>= 1)
            ++stages;
        clSetKernelArg(kernel, 0, sizeof(cl_mem),(void*)&device_A_in);
        clSetKernelArg(kernel, 3, sizeof(cl_uint),(void*)&sortOrder);
#ifdef USE_SHARED_MEM
        clSetKernelArg(kernel, 4, (GROUP_SIZE << 1) *sizeof(cl_
uint),NULL);
#elif def USE_SHARED_MEM_2
```

Any command-line input or output is written as follows:

```
# gcc -Wall test.c -o test
```

New terms and important words are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking on the **Next** button moves you to the next screen".

www.EngineeringBooksPdf.com

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

www.EngineeringBooksPdf.com

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

www.EngineeringBooksPdf.com

1
Using OpenCL

In this chapter, we will cover the following recipes:

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

Introduction

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

www.EngineeringBooksPdf.com

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

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

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

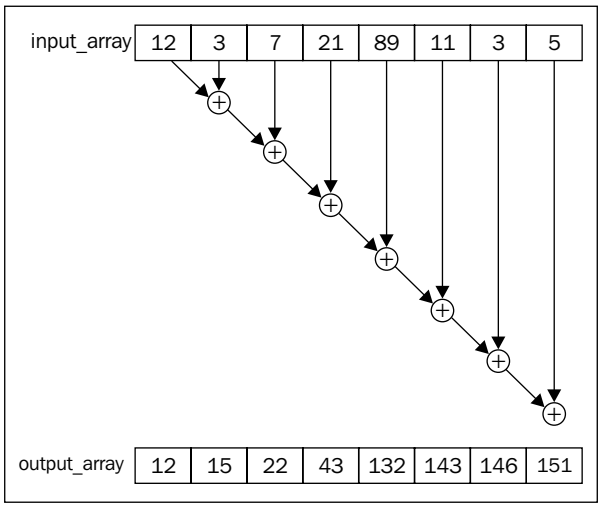
www.EngineeringBooksPdf.com

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

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

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

When attempting to realize data parallelism, the developer needs to readjust the way they think about data and how they can be read and updated concurrently. A common problem found in parallel computation would be to compute the sum of all the elements given in an arbitrary array of values, while storing the intermediary summed value and one possible way to do this is illustrated in the following diagram and the operator being applied there, that is, $\oplus$ is any binary associative operator. Conceptually, the developer could use a task to perform the addition of two elements of that input to derive the summed value.



www.EngineeringBooksPdf.com

Whether the developer chooses to embody task/data parallelism is dependent on the problem, and an example where task parallelism would make sense will be by traversing a graph. And regardless of which model the developer is more inclined with, they come with their own sets of problems when you start to map the program to the hardware via OpenCL. And before the advent of OpenCL, the developer needs to develop a module that will execute on the desired device and communication, and I/O with the driver program. An example example of this would be a graphics rendering program where the CPU initializes the data and sets everything up, before offloading the rendering to the GPU. OpenCL was designed to take advantage of all devices detected so that resource utilization is maximized, and hence in this respect it differs from the "traditional" way of software development.

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

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

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

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

www.EngineeringBooksPdf.com

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

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

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

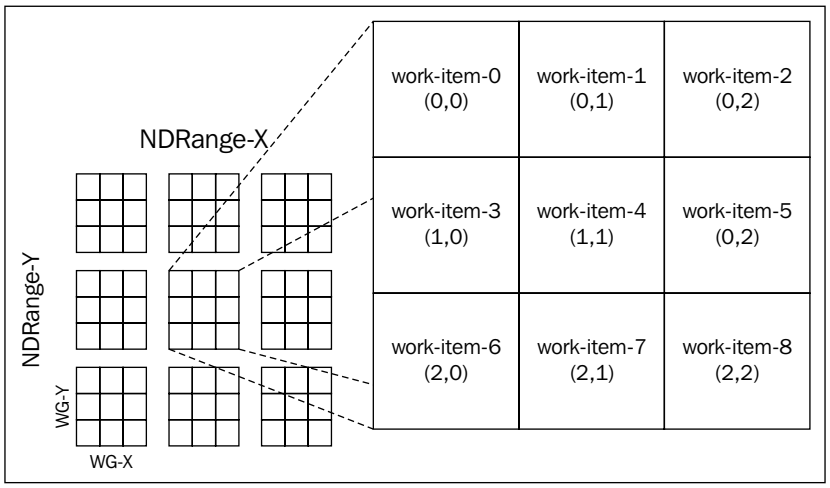
```
__kernel void vector_multiplication(__global int* a,
                                    __global int* b,
                                    __global int* c) {
int threadId = get_global_id(0); // OpenCL function
c[i] = a[i] * b[i];
}
```

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

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

www.EngineeringBooksPdf.com

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



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

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

www.EngineeringBooksPdf.com

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

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

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

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

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

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

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

www.EngineeringBooksPdf.com

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

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

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

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

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

Querying OpenCL platforms

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

Getting ready

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

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

www.EngineeringBooksPdf.com

How to do it...

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

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

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

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

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

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

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


www.EngineeringBooksPdf.com

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

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

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

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

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

         displayPlatformInfo( platforms[i],
                              CL_PLATFORM_EXTENSIONS,
                              "CL_PLATFORM_EXTENSIONS" );
    }
    return 0;
}
```


www.EngineeringBooksPdf.com

To compile it on the UNIX platform, you would run a compile command similar to the following:

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

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

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

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

How it works...

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

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

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

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

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

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

www.EngineeringBooksPdf.com

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

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

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

Querying OpenCL devices on your platform

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

Getting ready

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

How to do it...

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

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

…
void displayDeviceInfo(cl_platform_id id,
                       cl_device_type dev_type) {
    /* OpenCL 1.1 device types */

    cl_int error = 0;
    cl_uint numOfDevices = 0;

    /* Determine how many devices are connected to your
       platform */
```


www.EngineeringBooksPdf.com

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

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

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

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

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

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


www.EngineeringBooksPdf.com

```
void displayDeviceDetails(cl_device_id id,
                          cl_device_info param_name,
                          const char* paramNameAsStr) {
  cl_int error = 0;
  size_t paramSize = 0;

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

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

  switch (param_name) {
    case CL_DEVICE_TYPE: {
            cl_device_type* devType = (cl_device_type*)
                        alloca(sizeof(cl_device_type) * paramSize);
            error = clGetDeviceInfo( id, param_name, paramSize,
devType, NULL );

            if (error != CL_SUCCESS ) {
                perror("Unable to obtain device info for param\n");
                return;
            }

            switch (*devType) {
              case CL_DEVICE_TYPE_CPU :
                   printf("CPU detected\n");break;
              case CL_DEVICE_TYPE_GPU :
                   printf("GPU detected\n");break;
              case CL_DEVICE_TYPE_DEFAULT :
                   printf("default detected\n");break;
            }
            }break;

      // omitted code - refer to source "device_details.c"
    } //end of switch
}
```

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

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


www.EngineeringBooksPdf.com

And a binary executable named `device_details` should be deposited locally on your machine.

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

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

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

How it works...

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

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

www.EngineeringBooksPdf.com

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

Querying for OpenCL device extensions

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

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

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

www.EngineeringBooksPdf.com

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

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

Getting ready

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

How to do it...

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

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

  cl_int error = 0;
  size_t paramSize = 0;

  error = clGetDeviceInfo( id, param_name, 0, NULL, &paramSize );
  if (error != CL_SUCCESS ) {
    perror("Unable to obtain device info for param\n");
    return;
  }
  /* the cl_device_info are preprocessor directives defined in cl.h
```


www.EngineeringBooksPdf.com

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

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

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

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

```
Number of OpenCL platforms found: 1
CL_PLATFORM_PROFILE: FULL_PROFILE
CL_PLATFORM_VERSION: OpenCL 1.0 (Dec 23 2010 17:30:26)
CL_PLATFORM_NAME: Apple
CL_PLATFORM_VENDOR: Apple
CL_PLATFORM_EXTENSIONS:
Number of detected OpenCL devices: 2
GPU detected
  VENDOR ID: 0x1022600
  Maximum number of parallel compute units: 6
  Maximum dimensions for global/local work-item IDs: 3
  Maximum number of work-items in each dimension: ( 512 512 64  )
  Maximum number of work-items in a work-group: 512
  Supported extensions: cl_khr_byte_addressable_store cl_khr_
global_int32_base_atomics cl_khr_global_int32_extended_atomics
cl_APPLE_gl_sharing cl_APPLE_SetMemObjectDestructor cl_APPLE_
ContextLoggingFunctions cl_khr_local_int32_base_atomics cl_khr_local_
int32_extended_atomics
CPU detected
  VENDOR ID: 0x1020400
```


www.EngineeringBooksPdf.com

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

How it works...

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

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

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

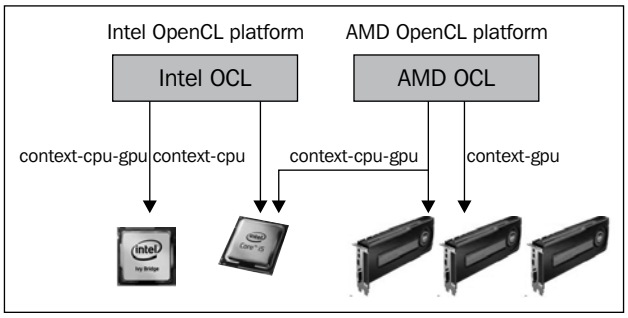
Querying OpenCL contexts

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

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

www.EngineeringBooksPdf.com

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



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

Getting ready

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

How to do it...

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

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


www.EngineeringBooksPdf.com

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

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

/*
 Load the information about your devices into the variable
 'devices'
*/

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

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

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

for(int i = 0; i < numOfDevices; ++ i ) {
    cl_context context = clCreateContext(NULL, 1,
                                         &devices[i],
                                         NULL, NULL,
                                         &error);
    cl_uint ref_cnt = 0;
    if (error != CL_SUCCESS) {
        perror("Can't create a context");
        exit(1);
```


www.EngineeringBooksPdf.com

```
            }

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

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

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

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

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

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

How it works...

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

www.EngineeringBooksPdf.com

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

cl_context_info	Return type	Information returned in `param_name`
`CL_CONTEXT_REFERENCE_COUNT`	`cl_uint`	This variable returns the context reference count
`CL_CONTEXT_NUM_DEVICES`	`cl_uint`	This variable returns the number of devices in context
`CL_CONTEXT_DEVICES`	`cl_device_id[]`	This variable returns a list of devices in context
`CL_CONTEXT_PROPERTIES`	`cl_context_properties`	This variable returns the properties argument specified in `clCreateContext` or `clCreateContextFromType`

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

Querying an OpenCL program

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

Getting ready

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

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


www.EngineeringBooksPdf.com

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

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

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

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

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

How to do it...

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

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


www.EngineeringBooksPdf.com

```
// platform. Build a OpenCL program and do not run it.
for(cl_uint i = 0; i < numOfPlatforms; i++ ) {
    // Get the GPU device
    error = clGetDeviceIDs(platforms[i],
                           CL_DEVICE_TYPE_GPU, 1,
                           &device, NULL);
    if(error != CL_SUCCESS) {
       // Otherwise, get the CPU
       error = clGetDeviceIDs(platforms[i],
                              CL_DEVICE_TYPE_CPU,
                              1, &device, NULL);
    }
     if(error != CL_SUCCESS) {
         perror("Can't locate any OpenCL compliant device");
         exit(1);
     }
     /* Create a context */
     context = clCreateContext(NULL, 1, &device, NULL, NULL,
                               &error);
     if(error != CL_SUCCESS) {
         perror("Can't create a valid OpenCL context");
         exit(1);
     }

     /* Load the two source files into temporary
        datastores */
     const char *file_names[] = {"simple.cl",
                                 "simple_2.cl"};
     const int NUMBER_OF_FILES = 2;
     char* buffer[NUMBER_OF_FILES];
     size_t sizes[NUMBER_OF_FILES];
     loadProgramSource(file_names, NUMBER_OF_FILES, buffer,
                       sizes);

     /* Create the OpenCL program object */
     program = clCreateProgramWithSource(context,
                                         NUMBER_OF_FILES,
                                         (const
                                          char**)buffer,
                                         sizes, &error);
   if(error != CL_SUCCESS) {
     perror("Can't create the OpenCL program object");
     exit(1);
   }
```


www.EngineeringBooksPdf.com

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


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

            /* Clean up */
            for(i=0; i< NUMBER_OF_FILES; i++) { free(buffer[i]); }
            clReleaseProgram(program);
            clReleaseContext(context);
        }
    }
```

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

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


www.EngineeringBooksPdf.com

You'll find the executable file named `build_opencl_program` deposited on the filesystem.

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

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

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

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

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

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

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

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

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

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

How it works...

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

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

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

www.EngineeringBooksPdf.com

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

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

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

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

The following table presents the math options available:

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

The following table highlights the optimization options available:

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

www.EngineeringBooksPdf.com

`-cl-finite-math-only`	This option allows optimizations to assume no floating point argument to take a NaN or an infinite value
`-cl-fast-relaxed-math`	This option combines the `-cl-unsafe-math-optimizations` and the `-cl-finite-math-only` options

The following table here highlights the miscellaneous options available:

`-w`	This option prevents all warning messages
`-Werror`	This option turns all warning messages into errors
`-cl-std=VERSION`	This option builds the program based on the version of the OpenCL compiler (VERSION={CL1.1})

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

Creating OpenCL kernels

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

```
__kernel void simpleAdd(__global float *a,
                        __global float *b,
                        __global float *c) {

  int gid = get_global_id(0);
   c[gid] = a[gid] + b[gid];
}
```

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

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

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


www.EngineeringBooksPdf.com

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

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

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

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

Getting ready

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

How to do it ...

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

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


www.EngineeringBooksPdf.com

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

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

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

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

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

How it works...

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

Finally, we invoke `clGetKernelInfo`, passing to it the retrieved kernel objects, and printing out some information about the kernel functions, such as the kernel function's name and the arity of the function through the `CL_KERNEL_FUNCTION_NAME` and `CL_KERNEL_NUM_ARGS` variables.

www.EngineeringBooksPdf.com

A complete list of details that can be queried from the kernel objects is reflected in the following table:

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

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

Creating command queues and enqueuing OpenCL kernels

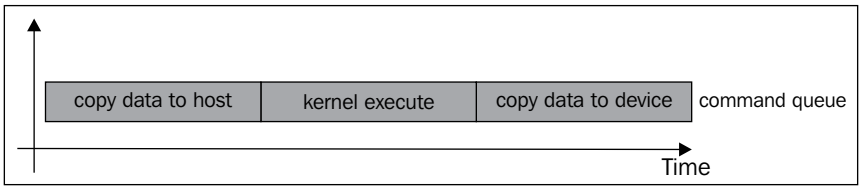
This section will show you how to enqueue OpenCL kernel objects on the device. Before we do that, let's recall that we can create kernels without specifying an OpenCL device and the kernels can be executed on the device via the command queue.

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

However, kernels are not the only objects that can be queued on the command queue. A kernel needs data so that it can perform its operations and data needs to transferred to the device for consumption, and these data could be OpenCL buffer / sub-buffer or image objects. The memory objects that encapsulate the data need to be transported into the device and you have to issue memory commands to the command queue for that to occur; and in many use cases, it is common to hydrate the device with data prior to computation.

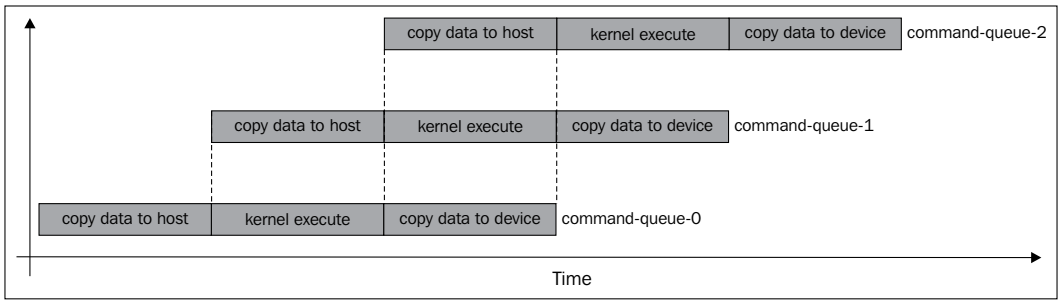
www.EngineeringBooksPdf.com

The following diagram highlights this use case where a kernel is scheduled for in-order execution, assuming that the kernel needs the data to be copied explicitly or memory-mapped, and upon completion of computation, the data is copied from the device's memory to host memory.



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

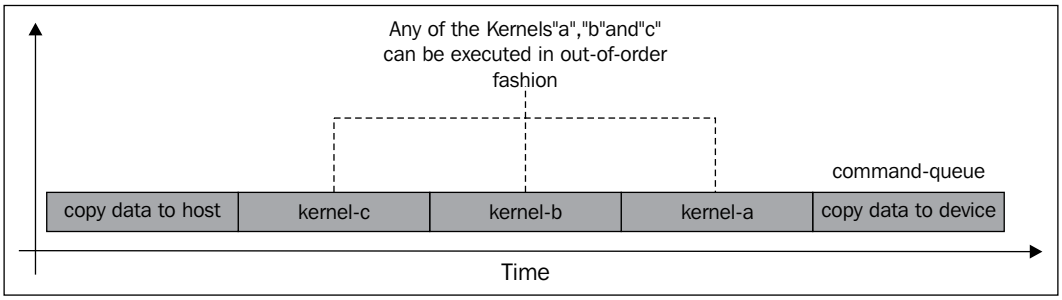
As an example of independent streams of computation, the following diagram assumes that three independent tasks have been identified and they need to execute on a device. Three command queues (in-order execution only) with tasks enqueued in each of them and a pipeline can be formed, such that the device executes the kernel code while I/O is being performed to achieve better utilization by not having the device sit idle waiting for data.



www.EngineeringBooksPdf.com

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

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



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

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

Getting ready

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

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


www.EngineeringBooksPdf.com

How to do it...

You should reference the full code listing alongside this recipe:

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

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

   /*
     Let OpenCL know that the kernel is suppose to receive an
     Argument
   */
   error = clSetKernelArg(kernels[i],
                          0,
                          sizeof(cl_mem),
                          &strObj);
```


www.EngineeringBooksPdf.com

```
    if (error != CL_SUCCESS) {
        perror("Unable to create buffer object");
        exit(1);
    }
    /* Enqueue the kernel to the command queue */
    error = clEnqueueTask(cQ, kernels[i], 0, NULL, NULL);

    if (error != CL_SUCCESS) {
        perror("Unable to enqueue task to command-queue");
        exit(1);
    }
    printf("Task has been enqueued successfully!\n");
    /* Release the command queue */
    clReleaseCommandQueue(cQ);
}
/* Clean up */
for(cl_uint i = 0; i < numOfKernels; i++) {
    clReleaseKernel(kernels[i]);
}
```

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

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

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

```
Number of OpenCL platforms found: 1
Kernel name: hello with arity: 1
About to create command queue and enqueue this kernel...
Task has been enqueued successfully!
```

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

How it works...

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

www.EngineeringBooksPdf.com

2

Understanding OpenCL Data Transfer and Partitioning

In this chapter, we'll cover the following recipes:

- Creating OpenCL buffer objects
- Retrieving information about OpenCL buffer objects
- Creating OpenCL sub-buffer objects
- Retrieving information about OpenCL sub-buffer objects
- Understanding events and event-synchronization
- Copying data between memory objects
- Using work items to partition data

Introduction

In this chapter, we're going to explore how to invoke the OpenCL's data transfer APIs, query memory objects, and data/work partitioning between the GPUs and CPUs.

Be aware that not all OpenCL SDKs support the compilation and execution on both GPUs and CPUs. AMD's OpenCL implementation supports its own AMD and Intel CPUs and GPUs; NVIDIA supports its GPUs and Intel supports its own Intel Core CPUs and Intel HD Graphics. Check with the vendor for supported devices.

www.EngineeringBooksPdf.com

In the **Open Computing Language** (**OpenCL**) development, you would inevitably need data to be processed, and the standard does not permit you to manipulate memory objects directly as you would do when you program in C or C++, because the data memory in the host is ultimately transferred to the devices in a heterogeneous environment for processing, and previously you would use the programming constructs in various libraries or languages to access them which is one of the reasons why OpenCL came about; hence to unify these approaches, the standard added abstractions to shield the developer from these concerns.

With respect to data types, there are a few you need to be aware of other than the one-dimensional data buffer. OpenCL buffer objects can be used to load and store two/three-dimensional data. The next data type in OpenCL is the `image` object; these objects are used to store two or three dimensional images (we won't cover much of using the `image` objects in this book).

The OpenCL 1.1 new data transfer capabilities includes the following:

- Using sub-buffer objects to distribute regions of a buffer across multiple OpenCL devices
- 3-component vector data types
- Using the global work offset which enables kernels to operate on different portions of the NDRange—global work offset refers to the data points in the input data where work items can start processing
- Reading, writing, or copying a 1D, 2D or 3D rectangular region of a buffer object

Creating OpenCL buffer objects

In the previous chapter, we understood the need to create or wrap our host's memory objects into an abstraction that OpenCL can operate on, and in this recipe we'll explore how to create a particular type of memory object defined in the specification that is commonly used for general purpose computation—buffer object. The developer can choose to create a one, two or three dimensional memory object that best fits the computational model.

Creating buffer objects is simple in OpenCL and is akin to the way in which you would use C's memory allocation routines such as `malloc` and `alloca`. But, that's where the similarity ends for the reason that OpenCL cannot operate directly on memory structures created by those routines. What you can do is to create a memory structure that lives on the devices that can be mapped to the memory on the host and the data is transferred to the device by issuing memory transfer commands to the command queue (which you recall is the conduit to the device). What you need to decide is the sort of objects, and how much of these objects you would like the device to compute.

www.EngineeringBooksPdf.com

In this example, we're going to learn how to create buffer objects based on user-defined structures also known as `structs` in the C/C++ language. Before that, let's understand the API:

```
cl_mem clCreateBuffer(cl_context context,
                      cl_mem_flags flags,
                      size_t size,
                      void* host_ptr,
                      cl_int* errcode_ret)
```

You can create a buffer by specifying which `context` it should attach to (recall that contexts can be created with several devices), specify the size of the data, and where to reference it with `size` and `host_ptr` respectively, specify how memory is to be allocated and whether that memory is to be of type read, read-only, read-write, or write only via `flags`; lastly capture the resultant error code in `errcode_ret`. Note that `clCreateBuffer` doesn't queue the command to conduct the memory transfer from host to device memory.

Getting ready

Here's a portion of the code from `Ch2/user_buffer/user_buffer.c` where you will see how to use the `clCreateBuffer` API to allocate memory for a user-defined structure. The problem we are trying to solve in this example is to send a million user-defined structures to the device for computation. The computation encapsulated by the kernel is a simple one—sum of all elements of each user-structure. The astute reader would have noticed we could have demonstrated this data structure with a vector data type in OpenCL, `int4`; the reason why we didn't do it that way is a two fold: (a) it's an example of application domain modeling, (b) because in a few paragraphs from current we wanted to illustrate how you could use the data type alignment construct, and don't fret over the data types now because we'll dive into the various data types in the next chapter. Continuing further, the user-defined structure is as follows:

```
typedef struct UserData {
 int x;
 int y;
 int z;
 int w;
} UserData;
```

What you will need to do is to create a buffer on the host application using standard C/C++ dynamic/static memory allocation techniques such as `new`, `malloc`, and `alloca`. Next, you will need to initialize that data buffer, and finally you will have to invoke `clCreateBuffer` and you should make sure it's done prior to the call to `clSetKernelArg`; recall that we mentioned that kernels get scheduled for execution on the device, well before it executes the kernel code on the device it would need data and values to work against, and you can achieve this by an invocation to `clSetKernelArg` and you typically do this when the buffer object is created.

www.EngineeringBooksPdf.com

The API `clSetKernelArg` looks like the following code and it'll be important for you to understand how it works:

```
cl_int clSetKernelArg(cl_kernel kernel,
                      cl_uint arg_index,
                      size_T arg_size,
                      const void *arg_value);
```

The kernel can take no arguments or at least one and probably more arguments, and how you configure them is simple. The following code snippet should complete the story:

```
// in the kernel code
_kernel void somefunction(__global int* arg1, __global int* arg2) {…}
// in the host code
int main(int argc, char**argv) {
// code omitted
cl_kernel kernel;
// kernel is initialized to point to "somefunction" in the kernel file
clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*) &memoryobjectA);
clSetKernelArg(kernel, 1, sizeof(cl_mem), (void*) &memoryobjectB);
```

Therefore, the kernel arguments are configured programmatically with the understanding that if the kernel function has `n` arguments then the `arg_index` would range from `0` to (`n – 1`).

How to do it...

We've included the main part of this recipe from `Ch2/user_buffer/user_buffer.c`, with the highlighted commentary:

```
/* Defined earlier */
#define DATA_SIZE 1048576
UserData* ud_in = (UserData*) malloc(sizeof(UserData) *
                                     DATA_SIZE); // input to device
/* initialization of 'ud_in' is omitted. See code for details.*/
/* Create a OpenCL buffer object */

cl_mem UDObj = clCreateBuffer(context,
                              CL_MEM_READ_ONLY |
                              CL_MEM_COPY_HOST_PTR,
                              sizeof(UserData) * DATA_SIZE,
                              ud_in, &error);
if (error != CL_SUCCESS) {
  perror("Unable to create buffer object");
  exit(1)
}
```


www.EngineeringBooksPdf.com

On OSX, you would compile the program by running the following command on your terminal:

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

On the Ubuntu Linux 12.04 with Intel OpenCL SDK, the command will be as follows:

```
gcc -std=c99 -Wall -DUNIX -g -DDEBUG -m64 -o user_buffer user_buffer.c
-I . -I /usr/include -L/usr/lib64/OpenCL/vendors/intel -lintelocl -ltbb
-ltbbmalloc -lcl_logger -ltask_executor
```

On the Ubuntu Linux 12.04 with AMD APP SDK v2.8, the command will be as follows:

```
gcc -std=c99 -Wall -DUNIX -g -DDEBUG –m64 -o user_buffer user_buffer.c
-I. –I/opt/AMDAPP/include –L/opt/AMDAPP/lib/x86_64 –lOpenCL
```

Regardless of the platform, a binary executable `user_buffer` would be deposited locally.

Running the application on both platforms, we would get the following result:

```
Number of OpenCL platforms found: 1
Kernel name: hello with arity: 1
About to create command queue and enqueue this kernel...
Task has been enqueued successfully!
Check passed!
```

How it works...

The application created a million of the `UserData` objects on the host. Refer to the following code snippet:

```
/*
  Prepare an array of UserData via dynamic memory allocation
*/
UserData* ud_in = (UserData*) malloc( sizeof(UserData) * DATA_SIZE);
// input to device
UserData* ud_out = (UserData*) malloc( sizeof(UserData) * DATA_SIZE);
// output from device
  for( int i = 0; i < DATA_SIZE; ++i) {
    (ud_in + i)->x = i;
    (ud_in + i)->y = i;
    (ud_in + i)->z = i;
    (ud_in + i)->w = 3 * i;
  }
```


www.EngineeringBooksPdf.com

The application then sends it to the device for computation after the program and kernel objects have been initialized, and we assign the recently created `UDObj` memory object to the kernel as its argument. Refer to the following code snippet:

```
error = clSetKernelArg(kernels[i], 0, sizeof(cl_mem), &UDObj);
  if (error != CL_SUCCESS) {
    perror("Unable to create buffer object");
      exit(1);
  }
```

Next, we issue a kernel execution command to the command-queue, `cQ`, and the code will run against the device, the following code snippet demonstrates the enqueuing of the kernel:

```
/* Enqueue the kernel to the command queue */
error = clEnqueueTask(cQ, kernels[i], 0, NULL, NULL);
  if (error != CL_SUCCESS) {
    perror("Unable to enqueue task to command-queue");
    exit(1);
  }
```

After that's done, the data in the device's memory is read back and we indicated that we wish to read the data back until the device has completed its execution by passing `CL_TRUE` to indicate blocking read which otherwise could result in partial data read back; finally the data is verified, demonstrated by the following code snippet:

```
/* Enqueue the read-back from device to host */
            error = clEnqueueReadBuffer(cQ, UDObj,
                                        CL_TRUE, // blocking read
                                        0, // write from the start
                                        sizeof(UserData) * DATA_SIZE,
// how much to copy
                                        ud_out, 0, NULL, NULL);
    if ( valuesOK(ud_in, ud_out) ) {
      printf("Check passed!\n");
    } else printf("Check failed!\n");
```

Let's explore how we used `clCreateBuffer` further.

In this scenario, you would want to allocate memory on the device as read-only when it comes to providing input to the device and because you want to be sure nothing else is writing to the data store. Therefore, the flag `CL_MEM_READ_ONLY` is passed, but if your input data was meant to be readable and writable then you would need to indicate it using `CL_MEM_READ_WRITE`. Notice that we actually created a data store on the host via `ud_in` and, we wanted our OpenCL memory object to be the same size as `ud_in` and the `C` statement reflects this; finally we wanted OpenCL to know that the new memory object is to copy its values from `ud_in` and we provided the flag `CL_MEM_COPY_HOST_PTR` too, and we use the bitwise `OR` operator that is represented on the standard US keyboard as a pipe symbol, `|`, to merge these two flags.

www.EngineeringBooksPdf.com

Conceptually, you can visualize it to be an 1D-array-of-structs for short or an array-of-structures in general.

```
UserData   UserData   UserData   ..........   UserData
```

Provide the same declaration of the application data type to the OpenCL kernel file (`*.cl`) as well as the host application files (`*.c`, `*.h`, `*.cpp`, `*.hpp`); else the OpenCL runtime will emit errors to reflect that the struct it is looking for does not exist, and the replication is necessary as OpenCL prohibits the `C` header file inclusion mechanism.

Let's spend some time to understand the C `struct` we just used in this example. The C structure we just used, `UserData`, is an example of an application data type. OpenCL makes no requirement about the alignment of OpenCL data types outside of buffers and images; hence developers of OpenCL need to make sure the data is properly aligned. Fortunately, OpenCL has provided attribute qualifiers so that we can annotate our types, functions and variables to suit the algorithm and CPU/GPU architecture with the primary motivation being to improve memory bandwidth. The alignment needs to be a power of two and at least a perfect multiple of the lowest common multiple of all the alignments of all the members of the `struct` or `union`.

Refer to Section 6.11.1 Specifiying Attributes of Types in the OpenCL 1.2 specification

Let's take a look at what is available to developers when it comes to aligning data types such as `enum`, `struct`, or `union`.

Data alignment is a direct result of how various computer systems restrict the allowable addresses for the primitive data types, requiring that the address for some type of object must be a multiple of some value *K* (typically 2, 4, or 8), and this actually simplifies the design of the hardware between the processor and the memory system. For example, if the processor were to always fetch 8 bytes from memory with an address that must be a multiple of 8, then the value can be read or written in a single memory operation otherwise, the processor needs to perform two or more memory accesses.

Alignment is enforced by making sure that every data type is organized and allocated in such a way that every object within the type satisfies its alignment restrictions.

Let's use an example for this illustration. Following is the generic manner in which alignment can be defined for application data type such as `UserData`. While examining the code, you will notice that without the `aligned` attribute, this data structure will be allocated on a 17-byte boundary assuming `int` is 4-bytes and `char` is 1-byte on a 32-bit / 64-bit system architecture. Once this attribute is included, following is the alignment:

```
| __attribute__((aligned))
```


www.EngineeringBooksPdf.com

The alignment is now determined by the OpenCL compiler to be aligned to 32-bytes instead of 17-bytes, that is, summing all the struct member's sizes, and the specification designates the alignment size to be the largest power of 2 and therefore it is 2^5 because, the 2^4 is 1-byte too many; however if you were to change the previous alignment to the following alignment:

```
| __attribute__((aligned (8)))
```

Then the alignment will be at least 8-bytes as shown in the following code:

```
typedef struct __attribute__((aligned)) UserData {
    int x;
    int y;
    int z;
    int w;
    char c;
} UserData;
```

Equivalently, you can also write in more explicit form as follows:

```
typedef struct __attribute__((aligned(32)) UserData {…}
```

In general, the golden rule of designing the data to be memory aligned is still a necessary practice; a rule of thumb I keep in mind is 16-byte aligned for 128-bit access and 32-byte aligned for 256-bit access.

On the other side of the story, you may find yourself wishing that the alignment wasn't that large, and with OpenCL you can indicate that by using the `packed` attribute as in the following code assuming that `LargeUserData` is an imaginary large data structure:

```
typedef struct __attribute__((packed)) LargeUserData {…}
```

When you apply this attribute to a `struct` or `union`, you're effectively applying the attribute to every member of the data; applying to an `enum` means that the OpenCL compiler will select the smallest integral type found on that architecture. You can refer to the `Ch2/user_buffer_alignment/user_buffer_align.c` to review what's done and how to profile the performance of the application via AMD APP SDK in the `readme.txt` file.

Retrieving information about OpenCL buffer objects

To retrieve information about a buffer or sub-buffer object, you'll need to use the API `clGetMemObjectInfo` and its signature as in the following code:

```
cl_int clGetMemObjectInfo(cl_mem memobj,
                          cl_mem_info param_name,
                          size_t param_value_size,
                          void* param_value,
                          size_t* param_value_size_ret)
```


www.EngineeringBooksPdf.com

To query the memory object, simply pass the object to `memobj` specifying the type of information you want in `param_name`, inform OpenCL the size of the returned information in `param_value_size` and where to deposit it in `param_value`; the last parameter, `param_value_size_ret`, is largely optional but it returns the size of the value in `param_value_size`.

Getting ready

Here's an excerpt from the code in `Ch2/buffer_query/buffer_query.c` where it shows how to extract the information about the memory object, `UDObj` is encapsulated into a user-defined function `displayBufferDetails` because, the code can be long depending on how many attributes you wish to extract about a memory object and you would place the invocation to this function after you've created the buffer object or if you have been given a handle to the memory object. The following code illustrates how it would display the information about a memory object by abstracting the OpenCL memory retrieval APIs into the function `displayBufferDetails`:

```
cl_mem UDObj = clCreateBuffer(context, … sizeof(UserData) *
                              DATA_SIZE, ud_in, &error);
/* Extract some info about the buffer object we created */
displayBufferDetails(UDObj);
```

How to do it...

We've included the main part of this recipe, as shown in the following code:

```
void displayBufferDetails(cl_mem memobj) {
  cl_mem_object_type objT;
  cl_mem_flags flags;
  size_t memSize;
  clGetMemObjectInfo(memobj, CL_MEM_TYPE,
                     sizeof(cl_mem_object_type), &objT, 0);
  clGetMemObjectInfo(memobj, CL_MEM_FLAGS, sizeof(cl_mem_flags),
                     &flags, 0);
  clGetMemObjectInfo(memobj, CL_MEM_SIZE, sizeof(size_t),
                     &memSize, 0);
  char* str = '\0';
  switch (objT) {
    case CL_MEM_OBJECT_BUFFER: str = "Buffer or Sub
                                     buffer";break;
    case CL_MEM_OBJECT_IMAGE2D: str = "2D Image Object";break;
    case CL_MEM_OBJECT_IMAGE3D: str = "3D Image Object";break;
  }
  char flagStr[128] = {'\0'};
```


www.EngineeringBooksPdf.com

```
    if(flags & CL_MEM_READ_WRITE) strcat(flagStr, "Read-Write|");
    if(flags & CL_MEM_WRITE_ONLY) strcat(flagStr, "Write Only|");
    if(flags & CL_MEM_READ_ONLY)  strcat(flagStr, "Read Only|");
    if(flags & CL_MEM_COPY_HOST_PTR) strcat(flagStr, "Copy from
                                                    Host|");
    if(flags & CL_MEM_USE_HOST_PTR)  strcat(flagStr, "Use from
                                                    Host|");
    if(flags & CL_MEM_ALLOC_HOST_PTR) strcat(flagStr, "Alloc from
                                                     Host|");
    printf("\tOpenCL Buffer's details =>\n\t size: %lu MB,\n\t object
  type is: %s,\n\t flags:0x%lx (%s) \n", memSize >> 20, str, flags,
  flagStr);
  }
```

On OSX, you will compile the program by running the following command on your terminal:

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

On Ubuntu Linux 12.04 with Intel OpenCL SDK, the command will be as follows:

```
gcc -std=c99 -Wall -DUNIX -g -DDEBUG -m64 -o buffer_query buffer_query.c
-I . -I /usr/include -L/usr/lib64/OpenCL/vendors/intel -lintelocl -ltbb
-ltbbmalloc -lcl_logger -ltask_executor
```

On Ubuntu Linux 12.04 with AMD APP SDK v2.8, the command will be as follows:

```
gcc -std=c99 -Wall -DUNIX -g -DDEBUG –m64 -o buffer_query buffer_query.c
-I. –I/opt/AMDAPP/include –L/opt/AMDAPP/lib/x86_64 –lOpenCL
```

Regardless of the platform, a binary executable `buffer_query` would be deposited locally.

Executing the program on an OSX 10.6 and Ubuntu 12.04 with AMD APP SDK v2.7 would present the following result:

```
Number of OpenCL platforms found: 1
Kernel name: hello with arity: 1
About to create command queue and enqueue this kernel...
  OpenCL Buffer's details =>
    size: 128 MB,
    object type is: Buffer or Sub-buffer,
    flags:0x21 (Read-Write|Copy from Host)
Task has been enqueued successfully!
Check passed!
```


www.EngineeringBooksPdf.com

How it works...

The host application proceeds to first create the buffer that it will send to the device, then the application queries for information about the buffer. The full list of attributes that can be queried is as shown in the following table:

cl_mem_info	Return type	Info. Returned in param_value
CL_MEM_TYPE	cl_mem_object_type	It returns CL_MEM_OBJECT_BUFFER if memobj is created with clCreateBuffer or clCreateSubBuffer.
Cl_MEM_FLAGS	cl_mem_flags	It returns the flags argument specified when memobj is created with clCreateBuffer, clCreateSubBuffer, clCreateImage2D, or clCreateImage3D.
CL_MEM_SIZE	size_t	It returns the actual size of the data associated with memobj in bytes.
CL_MEM_HOST_PTR	void*	If memobj is created with clCreateBuffer or clCreateImage2d, clCreateImage3D, then it returns the host_ptr argument specified when memobj is created. If memobj is created with clCreateSubBuffer, then it returns the host_ptr plus origin specified when memobj was created. See clCreateBuffer for what host_ptr is.
CL_MEM_MAP_COUNT	cl_uint	Map count.
CL_MEM_REFERENCE_COUNT	cl_uint	It returns memobj's reference count.

www.EngineeringBooksPdf.com

cl_mem_info	Return type	Info. Returned in param_value
CL_MEM_CONTEXT	cl_context	It returns the context specified when the memory is created. If memobj is created using clCreateSubBuffer, the context associated with the memory object specified as the buffer argument to clCreateSubBuffer is returned.
CL_MEM_ASSOCIATED_MEMOBJECT	cl_mem	It return memory object from which memobj is created. In clCreateSubBuffer, it returns the buffer argument; else NULL is returned.
CL_MEM_OFFSET	size_t	Applicable to memobj created via clCreateSubBuffer. It returns offset or 0.

Creating OpenCL sub-buffer objects

Sub-buffers are incredibly useful data types and as you continue to explore OpenCL in this chapter, you'll notice that this data type can be used to partition the data and distribute them across your OpenCL devices on your platform.

At the time of this writing, sub-buffer support is not enabled on OpenCL delivered in the OSX 10.6, because the official version is OpenCL 1.0. However, if you have OSX 10.7 then you'll be able to run this code without any problem.

Let's take a look at the method signature and examine it:

```
cl_mem clCreateSubBuffer(cl_mem buffer,
                         cl_mem_flags flags,
                         cl_buffer_create_type bufferType,
                         const void* buffer_create_info,
                         cl_int* errcode_ret)
```

The argument `buffer` refers to the buffer you created via `clCreateBuffer`, the `flags` argument refers to the options you wish this offer to have and if it's zero then the default option is `CL_MEM_READ_WRITE`; this flag can adopt any values from the previous table. The argument `bufferType` is of a data structure:

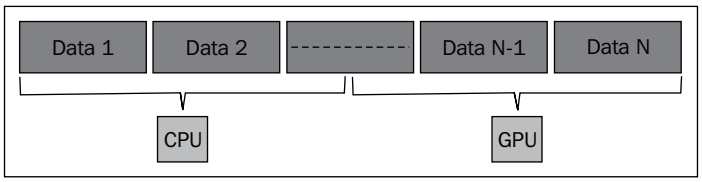
www.EngineeringBooksPdf.com

```
typedef struct _cl_buffer_region {
  size_t origin;
  size_t size;
} cl_buffer_region;
```

Therefore, you indicate where to start creating the region via the `origin` argument and how large it is going to be via the `size` argument.

Getting ready

In the *How to do it...* section of this recipe there is an excerpt from `Ch2/sub_buffers/sub_buffer.c` where we create two sub-buffer objects and each of them holds one-half of the data; these two sub-buffers will be sent to each OpenCL device on my setup, and they're computed and results are checked. Conceptually, here's what the code is doing:



How to do it...

We've included the main part of this recipe as shown in the following code:

`/* Chop up the data evenly between all devices & create sub-buffers */`

```
    cl_buffer_region region;
    region.size   = (sizeof(UserData)*DATA_SIZE) / numOfDevices;
    region.origin = offset * region.size;
    cl_mem subUDObj = clCreateSubBuffer(UDObj,
                                CL_MEM_READ_WRITE, // read-write
                                CL_BUFFER_CREATE_TYPE_REGION,
                                &region, &error);
    if (error != CL_SUCCESS) {
      perror("Unable to create sub-buffer object");
      exit(1);
    }
```

`/* Let OpenCL know that the kernel is suppose to receive an argument */`

```
error = clSetKernelArg(kernels[j], 0, sizeof(cl_mem), &subUDObj);
// Error handling code omitted
```


www.EngineeringBooksPdf.com

As noted earlier, this application doesn't work on OSX 10.6 and hence to compile it using the AMD APP SDK, you will enter the following command:

```
gcc -std=c99 -Wall -DUNIX -g -DDEBUG -m64 -o sub_buffer sub_buffer.c -I.
-I/opt/AMDAPP/include -L/opt/AMDAPP/lib/x86_64 -lOpenCL
```

For the Intel OpenCL SDK, you will enter the following command:

```
gcc -std=c99 -Wall -DUNIX -g -DDEBUG -m64 -o sub_buffer sub_buffer.c -I.
-I/usr/include
-L/usr/lib64/OpenCL/vendors/intel
-lintelocl
-ltbb
-ltbbmalloc
-lcl_logger
-ltask_executor
```

For NVIDIA on Ubuntu Linux 12.04, you will enter the following command:

```
gcc -std=c99 -Wall -DUNIX -g -DDEBUG -m64 -o sub_buffer sub_buffer.c -I.
-I/usr/local/cuda/include -lOpenCL
```

Regardless of the platform, a binary executable `sub_buffer` would be deposited locally.

In the setup I have with Ubuntu Linux 12.04 with a NVIDIA GTX460 graphics chip with both NVIDIA's and Intel's OpenCL toolkit installed, I have the following output:

```
Number of OpenCL platforms found: 2
Number of detected OpenCL devices: 1
Kernel name: hello with arity: 1
About to create command queue and enqueue this kernel...
Task has been enqueued successfully!
Check passed!
```

In the other setup with Ubuntu Linux 12.04 with an ATI 6870x2 graphics chip and AMD APP SDK installed, the difference in the output is only that the number of platforms is one and data is split between the CPU and GPU:

```
Number of OpenCL platforms found: 1
Number of detected OpenCL devices: 2
Kernel name: hello with arity: 1
About to create command queue and enqueue this kernel...
Task has been enqueued successfully!
Check passed!
Kernel name: hello with arity: 1
About to create command queue and enqueue this kernel...
Task has been enqueued successfully!
Check passed!
```


www.EngineeringBooksPdf.com

How it works...

The application basically discovers all the OpenCL compliant devices and keeps tracks of how it discovered. Next, the application uses the prior information to divide the data among the devices before enqueuing the data for execution and the code snippet demonstrates the following:

```
cl_buffer_region region;
region.size   = (sizeof(UserData)*DATA_SIZE) / numOfDevices;
region.origin = offset * region.size;
cl_mem subUDObj = clCreateSubBuffer(UDObj,
                                    CL_MEM_READ_WRITE, // read-write
                                    CL_BUFFER_CREATE_TYPE_REGION,
                                    &region, &error);
```

Finally, the data is checked for sanity after reading the data back from the device memory to the host memory as the following code snippet shows:

```
error = clEnqueueReadBuffer(cQ,
                            subUDObj,
                            CL_TRUE, // blocking read
                            region.origin, // write from the last
offset
                            region.size, // how much to copy
                            ud_out, 0, NULL, NULL);
                /* Check the returned data */
                if ( valuesOK(ud_in, ud_out, DATA_SIZE/numOfDevices){
                  printf("Check passed!\n");
               } else printf("Check failed!\n");
```

What you've just seen is a data partitioning technique also known as the distributed array pattern on a one-dimensional block of data.

Based on the distributed array pattern, there had been three general techniques that were developed, and they are over one-dimensional and two-dimensional blocks of data and finally the block-cyclic pattern.

www.EngineeringBooksPdf.com

Depending on whether you've installed one or more OpenCL toolkits from the vendors, the OpenCL will report the appropriate platforms and the OpenCL **Installable Client Driver** (**ICD**) allows multiple OpenCL implementations to co-exist on the same physical machine. Refer to the URL `http://www.khronos.org/registry/cl/extensions/khr/cl_khr_icd.txt` for more information about ICDs. This explains why your program may display distinct numbers for each installed platforms. The ICD actually identifies the vendors who provided the OpenCL implementation on the machine you have setup and its main function is to expose the platforms to the host code so that the developer may choose to run the algorithm in question against. The ICD has two pieces of information—(a) entry points to the vendor's OpenCL implementation in the library on the filesystem on which it's been installed, (b) the suffix string used to identify the suffix for OpenCL extensions provided by that vendor.

Retrieving information about OpenCL sub-buffer objects

The retrieval of information about OpenCL sub-buffers is very similar to that described in the previous recipe and involves the invocation of `clGetMemObjInfo`. Let's take a look at it.

OSX Caveat—you will need a OpenCL 1.1, at least the implementation to see this build and run; since OSX 10.6 doesn't support that version, you'll have to get a OSX 10.7 to get this code to run.

Getting ready

In the `Ch2/sub_buffer_query/subbuffer_query.c`, you'll find an excerpt of the following code demonstrating how we would pass the sub-buffer memory object to our defined function `displayBufferDetails`:

```
cl_buffer_region region;
region.size = sizeof(UserData)*DATA_SIZE;
region.origin = 0;
cl_mem subUDObj = clCreateSubBuffer(UDObj,
                                    CL_MEM_READ_WRITE, // read-write
                                    CL_BUFFER_CREATE_TYPE_REGION,
                                    &region, &error);
displayBufferDetails(subUDObj);
```


www.EngineeringBooksPdf.com

During my experimentation, I found that the NVIDIA CUDA 5 OpenCL toolkit was stricter in evaluating the attributes in the argument flags that's passed to `clCreateSubBuffer` as compared to AMD's APP SDK v2.7. Take note that the bug may be fixed by the time you read this book. As a concrete example, the following code throws an error using NVIDIA as opposed to AMD when you write:

`clCreateSubBuffer(buffer,CL_MEM_READ_WRITE|CL_MEM_COPY_HOST_PTR,...)` to reflect the fact that `CL_MEM_COPY_HOST_PTR` doesn't make sense.

How to do it...

We've included the main part of this recipe, as shown in the following code:

```
void displayBufferDetails(cl_mem memobj) {
  cl_mem_object_type objT;
  cl_mem_flags flags;
  size_t memSize;
  size_t memOffset;
  cl_mem mainBuffCtx;
  clGetMemObjectInfo(memobj, CL_MEM_TYPE,
                     sizeof(cl_mem_object_type), &objT, 0);
  clGetMemObjectInfo(memobj, CL_MEM_FLAGS, sizeof(cl_mem_flags),
                     &flags, 0);
  clGetMemObjectInfo(memobj, CL_MEM_SIZE, sizeof(size_t),
                     &memSize, 0);
  clGetMemObjectInfo(memobj, CL_MEM_OFFSET, sizeof(size_t),
                     &memOffset, 0); // 'CL_MEM_OFF_SET' new in OpenCL
1.2
  clGetMemObjectInfo(memobj, CL_MEM_ASSOCIATED_MEMOBJECT,
                     sizeof(size_t),
                     &memOffset, 0);
  char* str = '\0';
  if (mainBuffCtx) { // implies that 'memobj' is a sub-buffer
    switch (objT) {
      case CL_MEM_OBJECT_BUFFER: str = "Sub-buffer";break;
      case CL_MEM_OBJECT_IMAGE2D: str = "2D Image Object";break;
      case CL_MEM_OBJECT_IMAGE3D: str = "3D Image Object";break;
    }
  } else {
switch (objT) {
  case CL_MEM_OBJECT_BUFFER: str = "Buffer";break;
  case CL_MEM_OBJECT_IMAGE2D: str = "2D Image Object";break;
  case CL_MEM_OBJECT_IMAGE3D: str = "3D Image Object";break;
```


www.EngineeringBooksPdf.com

```
    }
  }
    char flagStr[128] = {'\0'};
    if(flags & CL_MEM_READ_WRITE) strcat(flagStr, "Read-Write|");
    if(flags & CL_MEM_WRITE_ONLY) strcat(flagStr, "Write Only|");
    if(flags & CL_MEM_READ_ONLY)  strcat(flagStr, "Read Only|");
    if(flags & CL_MEM_COPY_HOST_PTR) strcat(flagStr, "Copy from
                                                    Host|");
    if(flags & CL_MEM_USE_HOST_PTR)  strcat(flagStr, "Use from
                                                    Host|");
    if(flags & CL_MEM_ALLOC_HOST_PTR) strcat(flagStr, "Alloc from
                                                     Host|");
    printf("\tOpenCL Buffer's details =>\n\t size: %lu MB,\n\t object
  type is: %s,\n\t flags:0x%lx (%s) \n", memSize >> 20, str, flags,
  flagStr);
  }
```

On the Ubuntu Linux 12.04 with AMD's APP SDK v2.8, the following command would suffice:

```
gcc -std=c99 -Wall -DUNIX -g -DDEBUG -m64 -o subbuffer _query subbuffer_
query.c -I. -I/opt/AMDAPP/include -L/opt/AMDAPP/lib/x86_64 -lOpenCL
```

For the Intel OpenCL SDK, you would enter the following command:

```
gcc -std=c99 -Wall -DUNIX -g -DDEBUG -m64 -o subbuffer_query subbuffer_
query.c -I. -I/usr/include
-L/usr/lib64/OpenCL/vendors/intel
-lintelocl
-ltbb
-ltbbmalloc
-lcl_logger
-ltask_executor
```

For NVIDIA on Ubuntu Linux 12.04, you would enter the following command :

```
gcc -std=c99 -Wall -DUNIX -g -DDEBUG -m64 -o subbuffer_query subbuffer_
query.c -I. -I/usr/local/cuda/include -lOpenCL
```

Regardless of the platform, a binary executable `subbuffer_query` would be deposited locally.

When you run the program, you should get something similar to the following output:

```
Number of OpenCL platforms found: 2
Kernel name: hello with arity: 1
About to create command queue and enqueue this kernel...
    OpenCL Buffer's details =>
```


www.EngineeringBooksPdf.com

```
    size: 128 MB,
   object type is: Buffer,
    flags:0x21 (Read-Write|Copy from Host|)
    OpenCL Buffer's details =>
    size: 128 MB,
    object type is: Sub-buffer,
    flags:0x1 (Read-Write|)
Task has been enqueued successfully!
Check passed!
```

How it works...

The application could decipher whether it's an OpenCL sub-buffer object because of the two flags introduced in OpenCL 1.2. They are `CL_MEM_OFFSET` and `CL_MEM_ASSOCIATED_MEMOBJECT`; using either one of the flags would reveal whether it's a sub-buffer, but the catch is that `CL_MEM_OFFSET` can be zero for a sub-buffer because that indicates to OpenCL where to start to extract the data from; a better, recommended option is to use `CL_MEM_ASSOCIATED_MEMOBJECT` since the presence implies the argument `memobj` is a sub-buffer. See the earlier recipe, *Retrieving information about OpenCL buffer objects*.

Understanding events and event-synchronization

The previous recipes demonstrated how you can create memory objects that encapsulates the data that is to be transferred from the host memory to the device memory, and discusses how you can partition the input data among the devices via sub-buffers.

In this recipe, we are going to develop an understanding of how the developer can make use of the event system in OpenCL to control execution of kernel commands as well as memory commands. This is beneficial to the developer because it offers myriad ways in which you can control execution flow in a heterogeneous environment.

Events are, generally, passive mechanisms when the developers wish to be notified of an occurrence, and having the choice of conducting processing past that occurrence; contrasting to the say, polling where it's a more active mechanism as the application makes an active enquiry into the current state and decides what to do when a particular condition is met.

Events in OpenCL fall into two categories as follows:

- Host monitoring events
- Command events

www.EngineeringBooksPdf.com

In both the event types, the developer needs to create the events explicitly and associate them with the objects through waitlists; waitlists are nothing more than a container of events that the command must wait upon completion, that is, the event's status is `CL_COMPLETE` or `CL_SUCCESS` before progressing. The difference between these two event types (as we shall soon see) is in the manner in which the next subsequent command in the queue gets executed, host events are updated by the developer and when this is done it is indicative by the program source, command events in the waitlists on the other hand are updated by the OpenCL runtime. Considering that the events held up in the waitlists must be of a certain state before the next command executes means that waitlists are actually synchronization points since no progress can be made without emptying that list.

Let's start by examining the host events. So far, we understood that commands needs to be placed onto the command queue so that they can be scheduled for execution, and what host monitoring events allow the developer is to monitor the state of enqueued command and we can, optionally, attach a callback function to the event so that when it returns with a state we desire, the callback function will execute. This is made possible via the APIs `clCreateUserEvent`, `clSetUserEventStatus`, `clReleaseEvent`, and `clSetEventCallback`. An example in the *How to do it* section would illustrate how this can be achieved.

Getting ready

Assume that a kernel wishes to process two 1D memory objects named `objA` and `objB` and write the result to `objC` (for this example, we can ignore the output of `objC`). We wish that the copying of input data from `objB` should only take place when we have indicated to the host program.

How to do it...

The full source is demonstrated in `Ch2/events/{events.c,sample_kernel.cl}` and we have to first create the necessary data structures as before; next we will create the event object as follows:

```
event1 = clCreateUserEvent(context, &ret);
```

In this event object, we can next assign a call back function to the event and indicate that upon the event's status changes to `CL_COMPLETE`, the callback would execute like the following code:

```
void CL_CALLBACK postProcess(cl_event event, cl_int status, void
*data) {
  printf("%s\n", (char*)data);
}
clSetEventCallback(event1, CL_COMPLETE, &postProcess, "Looks like its
done.");
```


www.EngineeringBooksPdf.com

Then the host program would continue to conduct memory transfers for `objA` and `objB`, but it doesn't proceed to process any more OpenCL commands enqueued on the command queue till the status of the `event1` is set to `CL_COMPLETE`.

```
ret = clEnqueueWriteBuffer(command_queue, objA, CL_TRUE, 0,
4*4*sizeof(float), A, 0, NULL, NULL );
  printf("A has been written\n");
  /* The next command will wait for event1 according to its status*/
  ret = clEnqueueWriteBuffer(command_queue, objB, CL_TRUE, 0,
4*4*sizeof(float), B, 1, &event1, NULL);
  printf("B has been written\n");
clSetUserEventStatus(event1, CL_COMPLETE);
//….code omitted
clReleaseEvent(event1);
```

Another API we will introduce is the `clWaitForEvents` with it's signature:

```
Cl_int clWaitForEvents(cl_uint num_events, const cl_event* event_
list);
```

This is typically used to stall the host thread until all the commands in the event list have completed (the next code snippet demonstrates how).

The next topic we look at are the command events, which are typically used when you wish to be notified of certain happenings associated with commands. A typical use case would be the following where you have a command-queue and you want to be notified of the status of an memory transfer command like `clEnqueueWriteBuffer` and take a particular action depending on that status:

```
cl_event event1;
// create memory objects and other stuff
ret = clEnqueueWriteBuffer(queue, object, CL_TRUE, 0, 1048576,
hostPtrA, 1, &event1, NULL);
clWaitForEvents(&event1); // stalls the host thread until 'event1' has
a status of CL_COMPLETE.
```

You can easily extrapolate the scenario where you have a large heterogeneous computing environment with large numbers of CPUs and GPUs and obviously you wish to maximize your computational power, and the events mechanism in OpenCL allows the developer to design how to sequence those computations and coordinate those computations. However, as a best practice you probably want to clean up the event object associated with the commands, but you need to discover the state of the event you're watching otherwise you might release the event prematurely, and here's how you can do that by polling the API `clGetEventInfo` passing in the event you are watching; the following code demonstrates this idea:

```
int
waitAndReleaseEvent(cl_event* event) {
  cl_int eventStatus = CL_QUEUED;
```


www.EngineeringBooksPdf.com

```
    while(eventStatus != CL_COMPLETE) {
      clGetEventInfo(*event,
                     CL_EVENT_COMMAND_EXECUTION_STATUS,
                     sizeof(cl_int),
                     &eventStatus, NULL);
    }
    clReleaseEvent(*event);
    return 0;
  }
```

There's more...

There are two scenarios that deserve mentioning and they address the situation where (a) you like to receive notification for a group of events (assuming that they are associated to memory objects) and (b) you like to stall the execution of any commands further down the pipeline, that is, command-queue, until this group of events you are watching for have completed. The API `clEnqueueMarkerWithWaitList` is for the former situation whereas `clEnqueueBarrierWithWaitList` suits the latter. You are encouraged to explore them in the OpenCL 1.2 specification.

If you are still using OpenCL 1.1, you can use `clEnqueueMarker` and `clEnqueueBarrier` (which are the older versions of `clEnqueueMarkerWithWaitList` and `clEnqueueBarrierWithWaitList`) but be aware that they are both deprecated in OpenCL 1.2.

Copying data between memory objects

You will quickly realize how useful the event mechanism in OpenCL is in controlling the various parts of your algorithm, and it can be found in the common kernel and memory commands. This recipe will continue from creating memory objects and focus on how those memory objects can be transferred from the host memory to the device memory and vice versa and we'll be fixated on the data transfer APIs `clEnqueueReadBuffer` and `clEnqueueWriteBuffer`, which is for one-dimensional data blocks, and `clEnqueueReadBufferRect` and `clEnqueueWriteBufferRect` for two-dimensional data blocks; we'll also look at `clEnqueueCopyBuffer` for data transfers between memory objects in the device. First, we look at copying data between memory objects.

www.EngineeringBooksPdf.com

There will come times when you have to copy data between distinct memory objects, and OpenCL provides us a convenient way to do this via `clEnqueueCopyBuffer`. It can only take place between two different memory objects (for example, one is a plain buffer and the other is a sub-buffer) or between two similar objects (for example, both are sub-buffers or plain buffers) and the area of copy cannot overlap. Here's the method signature:

```
cl_int clEnqueueCopyBuffer(cl_command_queue command_queue,
                           cl_mem src_buffer,
                           cl_mem dst_buffer,
                           size_t src_offset,
                           size_t dst_offset,
                           size_t cb,
                           cl_uint num_events_in_wait_list,
                           const cl_event* event_wait_list,
                           cl_event* event)
```

The list of functions for copying data between memory objects are as follows:

- `clEnqueueCopyBuffer`
- `clEnqueueCopyImage`
- `clEnqueueCopyBufferToImage`
- `clEnqueueCopyImageToBuffer`
- `clEnqueueCopyBufferRect`

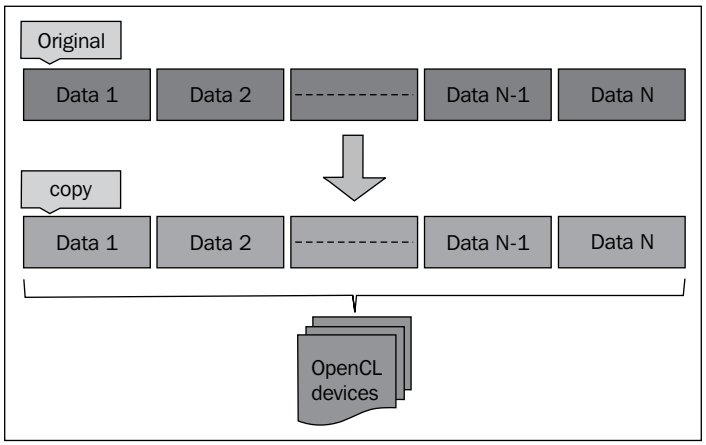
To copy a buffer, you need to indicate the source and destination `cl_mem` objects via `src_buffer` and `dst_buffer`, indicate where to start the copying by indicating the offsets of the `src_buffer` and `dst_buffer` via `src_offset` and `dst_offset` respectively together with the size of data to copy via `cb`. If you wish for the copying of the data to take place after some operations, you need to indicate the number of those operations and a valid array of `cl_event` objects that represent each operation via `num_events_in_wait_list` and `event_wait_list` respectively.

Take note that you can query the device on the status of the copying, when your data array is large, by passing an event object to the `event` argument. Another approach is to enqueue a `clEnqueueBarrier` command.

www.EngineeringBooksPdf.com

Getting ready

The following code is an extract from `Ch2/copy_buffer/copy_buffer.c`, and it illustrates how to enqueue a `clEnqueueCopyBuffer` command to the command queue, and the kernel uses this copy of the data for computation. This process is iterated among the detected OpenCL devices on the machine. The following diagram illustrates how the original data block (previous diagram) is copied to another `cl_mem` object (next diagram) and passed off to the OpenCL devices for computation.



How to do it...

We've included the main part of this recipe, with the highlighted commentary:

```
cl_mem UDObj = clCreateBuffer(context,
                              CL_MEM_READ_WRITE |
                              CL_MEM_COPY_HOST_PTR,
                              sizeof(UserData) * DATA_SIZE,
                              ud_in, &error);
… // code omitted. See the source.
/* Create a buffer from the main buffer 'UDObj' */
cl_mem copyOfUDObj = clCreateBuffer(context, CL_MEM_READ_WRITE,
                                    sizeof(UserData) * DATA_SIZE,
                                    0, &error)
if (error != CL_SUCCESS) {
  perror("Unable to create sub-buffer object");
  exit(1);
}
/* Let OpenCL know that the kernel is suppose to receive an argument
*/
error = clSetKernelArg(kernels[j],
```


www.EngineeringBooksPdf.com

```
                          0,
                          sizeof(cl_mem),
                          &copyOfUDObj);
    if (error != CL_SUCCESS) {
      perror("Unable to set buffer object in kernel");
      exit(1);
    }
    // code omitted. See the source.
    /* Enqueue the copy-write from device to device */
    error = clEnqueueCopyBuffer(cQ,
                                UDObj,
                                copyOfUDObj,
                                0,            // copy from which offset
                                0,            // copy to which offset
                                sizeof(UserData)*DATA_SIZE,
                                0, NULL, NULL);
    printf("Data will be copied!\n");
    // Code for enqueueing kernels is omitted.
    /* Enqueue the read-back from device to host */
    error = clEnqueueReadBuffer(cQ,
                                copyOfUDObj,
                                CL_TRUE,  // blocking read
                                0,        // read from the start
                                sizeof(UserData)*DATA_SIZE,
                                ud_out, 0, NULL, NULL);
```

On OSX, you can run the following command:

```
gcc -std=c99 -Wall -DUNIX -g  -DAPPLE -arch i386 -o copy_buffer copy_
buffer.c   -framework OpenCL
```

On Ubuntu Linux 12.04 with Intel OpenCL SDK installed, you can run the following command:

```
gcc -std=c99 -Wall -DUNIX -g -DDEBUG -m64 -o copy_buffer copy_buffer.c
-I . -I /usr/include -L/usr/lib64/OpenCL/vendors/intel -lintelocl -ltbb
-ltbbmalloc -lcl_logger -ltask_executor
```

On Ubuntu Linux 12.04 with NVIDIA CUDA 5 installed, you can run the following command:

```
gcc -std=c99 -Wall -DUNIX -g -DDEBUG -m64 -o copy_buffer copy_buffer.c
-I. -I/usr/local/cuda/include  -lOpenCL
```

A binary executable named `copy_buffer` will be deposited on the directory.

www.EngineeringBooksPdf.com

Depending on how many OpenCL SDKs are installed on your machine, your output may vary but on my OSX, the following is the output:

```
Number of OpenCL platforms found: 1
Number of detected OpenCL devices: 2
Kernel name: hello with arity: 1
About to create command queue and enqueue this kernel...
Task has been enqueued successfully!
Data will be copied!
Check passed!
Kernel name: hello with arity: 1
About to create command queue and enqueue this kernel...
Task has been enqueued successfully!
Data will be copied!
Check passed!
```

How it works...

The application needed to compute the copied buffer, and you can tell this because `clSetKernelArg` was defined that way by this statement:

```
error = clSetKernelArg(kernels[j], 0, sizeof(cl_mem), &copyOfUDObj);
```

Next, we can perform a copy operation, which takes place in the device's memory, via `clEnqueueCopyBuffer` and finally retrieve the computed values via `clEnqueueReadBuffer`.

The created command queue will default to in-order execution, instead of out-of-order execution so the device will execute the commands in the order of the queueing.

Now, we are going to talk about the one-dimensional and two-dimensional data transfer APIs such as `clEnqueueReadBuffer`, `clEnqueueWriteBuffer`, `clEnqueueWriteBufferRect`, and `clEnqueueReadBufferRect` and we are doing this now because you have seen that most of our examples, so far, we demonstrated the creation of memory objects via `clCreateBuffer` by associating with a memory structure in the host and though that might suffice for some situations, you probably want APIs that gives you more control when memory objects in the device memory are to be written or read from. The control these APIs give you, the developer, is from the fact that they are enqueued onto the command-queue with any events the developer might craft; and that provides a good permutation of strategies and flexibilities for structuring I/O in heterogeneous environments.

www.EngineeringBooksPdf.com

Be aware that there are similar APIs for reading and writing two or three dimensional images to/from host to the device memory. Their names are `clEnqueueReadImage`, `clEnqueueWriteImage`, `clEnqueueReadImageRect`, and `clEnqueueWriteImageRect`. Refer to the OpenCL 1.2 Specifications for more details.

These APIs allows us to indicate to the device when we wish the data transfer to occur, very much like `clEnqueueCopyBuffer`. Let's take a look at their method signatures:

```
cl_int clEnqueueReadBuffer(cl_command_queue command_queue,
                           cl_mem buffer,
                              cl_bool blocking_read,
                              size_t offset,
                              size_t cb,
                              void *ptr,
                              cl_uint num_events_in_wait_list,
                              const cl_event *event_wait_list,
                              cl_event *event)
cl_int clEnqueueWriteBuffer(cl_command_queue command_queue,
                           cl_mem buffer,
                              cl_bool blocking_write,
                              size_t offset,
                              size_t cb,
                              const void *ptr,
                              cl_uint num_events_in_wait_list,
                              const cl_event *event_wait_list,
                              cl_event *event)
```

These two functions are very similar to one another, and they basically say if you wish to read/write to/from a memory buffer , that is, a `cl_mem` object, you need to indicate which command queue is it via `command_queue`, what buffer it is via `buffer`, whether to be a blocking-read/write via `blocking_read/blocking_write`, where to read/write from for what size via `offset` and `cb`, where to read the data or write the data to via `ptr`, should this read/write command occur after some events via `num_events_in_wait_list` and `event_wait-list`. The last argument in the function is `event`, which allows the reading or writing operation to be queried which is described in `clEnqueueCopyBuffer`.

Blocking reads in `clEnqueuReadBuffer` means that the command does not exit until the host pointer has been filled by the device memory buffer; similarly blocking-writes in `clEnqueueWriteBuffer` means that the command doesn't exit until the entire device memory buffer has been written to by the host pointer.

www.EngineeringBooksPdf.com

To see how these calls are used, you can refer to the earlier illustrated code in the recipe *Understanding events and event-synchronization* and for your convenience the following is the relevant code in `Ch2/events/events.c`:

```
ret = clEnqueueWriteBuffer(command_queue, objA, CL_TRUE, 0,
4*4*sizeof(float), A, 0, NULL, NULL );
ret = clEnqueueWriteBuffer(command_queue, objB, CL_TRUE, 0,
4*4*sizeof(float), B, 1, &event1, NULL);
```

Having the capability to model one-dimensional memory objects is fantastic, but OpenCL goes a notch further by facilitating two-dimensional memory object memory transfers.

Here is an example of reading a two-dimensional data blocks from the device's memory to the output buffer in the host memory; extracted from `Ch2/simple_2d_readwrite/simple_2d_readwrite.c`. The code illustrates the usage of the `buffer_origin`, `host_origin`, and `region` as in the API. The application will read from the `UDObj cl_mem` object, which represents the one-dimensional input data, `hostBuffer`, as a 2 x 2 matrix and writes them into the host memory data block represented by `outputPtr`. The application reads back the data from the device to host memory and checks for sanity.

```
cl_int hostBuffer[NUM_BUFFER_ELEMENTS] = {0, 1, 2, 3, 4, 5, 6, 7,
                                          8,9,10,11,12,13,14,15};
cl_int outputPtr[16] = {-1, -1, -1, -1,-1, -1, -1, -1,-1, -1, -1,
                        -1,-1, -1, -1, -1};
for(int idx = 0; idx < 4; ++ idx) {
    size_t buffer_origin[3] = {idx*2*sizeof(int), idx, 0};
    size_t host_origin[3] = {idx*2*sizeof(int), idx, 0};
    size_t region[3] = {2*sizeof(int), 2, 1};
error = clEnqueueReadBufferRect (cQ,
                                 UDObj,
                                 CL_TRUE,
                                 buffer_origin,
                                 host_origin,
                                 region,
                                 0, //buffer_row_pitch,
                                 0, //buffer_slice_pitch,
                                 0, //host_row_pitch,
                                 0, //host_slice_pitch,
                                 outputPtr,
                                 0, NULL, NULL);
}//end of for-loop
```


www.EngineeringBooksPdf.com

In this example, we used the `for` loop and standard array indexing techniques in `C` to model how you might iterate through a two-dimensional array and referencing the elements so that we progressively copy the input. We won't dwell too much into this because, building and running it is very similar to the previous, and you should explore the directory to see how the build and program works via the Makefile.

Using work items to partition data

In the previous chapter, we introduced how work can be partitioned in a one-dimensional array across several work items (you should flip back now if you cannot remember), and also how each work item would obtain an index in which the kernel can use to conduct the computation in the kernel code `vector_multiplication`. In this recipe, we are going to build on that by exploring two-dimensional data partitioning in more detail.

By now, you should realize that one of the cornerstones of OpenCL is getting the data into the device/s for processing via kernels, and you've seen how data can be partitioned among different devices via kernels. In the former, you've seen how we used the distributed array pattern to partition the data among the devices; this refers to coarse grain data-parallelism. The latter refers to the coarse grained task-parallelism that OpenCL provides and it is coarse grained because OpenCL is capable of both data-parallelism and task-parallelism.

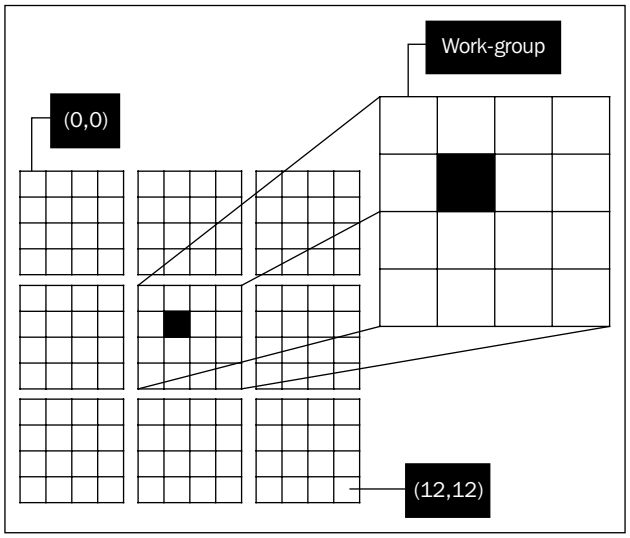
Most of the code you've seen so far have been using `clEnqueueTask` to execute the kernel based on the one-dimensional data blocks and to get your kernel to process two or three dimensional data we need to understand `clEnqueueNDRangeKernel`; and how data can be laid out conceptually in two or three dimensional space.

It is helpful to visualize the two or three dimensional data layout in the device memory to be row-based instead of column-based.

The `NDRange` in `clEnqueueNDRangeKernel` refers to a data indexing scheme that is supposed to span an N-dimensional range of values and hence, the given name. Currently, *N* in this N-dimensional index space can be one, two, or three. Next, we can split each dimensional into chunks of sizes two, three, four, or more till we reached the maximum allowable by the parameter `CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS`. Refer to the `Ch1/device_details/device_details.c` on how to obtain the values. This would decide how many processing groups we can run in parallel, and in OpenCL they are called **work groups**. The work groups would have a number of available processing elements that are called **work items** though I like to think of them as executable threads.

www.EngineeringBooksPdf.com

Let's work through an example using a two-dimensional data size of 12 rows by 12 columns, that is, a 12 x 12 matrix. Let's look at the following diagram to understand how the work groups and work items are related to one another:



In this example, I've decided to partition the two-dimensional space to create nine work groups where each work group is a 4 x 4 matrix. Next, to decide how many work items there should be in each work group, and you have two choices: a) assign one work-item to process each cell in your 4 x 4 matrix, b) assign one work item to process n-cells in your 4 x 4 matrix; in the second option it would be similar to vector processing where n-values are loaded together for the work item to process. Let's assume that we've decided to choose the option a

We'll look at the various data types in the *Chapter 3*, *Understanding OpenCL Data Types*.

At this time, let's take a detailed look at the API `clEnqueueNDRangeKernel` with the following method signature, and understand how to input those values with our example:

```
cl_int
clEnqueueNDRangeKernel(cl_command_queue command_queue,
                       cl_kernel kernel,
                       cl_uint work_dim,
                       const size_t *global_work_offset,
                       const size_t *global_work_size,
                       const size_t *local_work_size,
                       cl_uint num_events_in_wait_list,
                       const cl_event *event_wait_list,
                       cl_event *event)
```


www.EngineeringBooksPdf.com

Let's look at what those variables in `clEnqueueNDRangeKernel` are for; the `command_queue` refers to the particular queue like the `kernel`, to execute on. Next, you need to indicate how many dimensions your input data has via `work_dim`; the next two variables `global_work_size` and `local_work_size` would indicate how many work groups there are and how many work items / work threads can execute in each work group. Recall that the kernel gets scheduled on the device, but it is the work group that gets assign compute units of the device and the work items execute on the processing element in the compute unit. Next, if you need the launch of the kernel to wait on a couple of events in your algorithm, you can indicate them through `num_events_in_wait_list` and `event_wait_list`, and finally if you wish to associate an event to this kernel's state you can pass in an event type to `event` in this API.

The method signature should not look that intimidating to you by now. Given a 12 x 12 matrix partitioned into nine work groups where each work group is a 4 x 4 matrix and each work item will process one data cell, we will code it like the following code snippet:

```
cl_uint work_dim = 2; // 2-D data
size_t global_work_offset[2] = {0,0}; // kernel evals from (0,0)
size_t global_work_size[2] = {12,12};
size_t local_work_size[2]  = {4,4};
clEnqueueNDRangeKernel(command_q, kernel, work_dim,
global_work_offset,global_work_size, local_work_size, 0,
NULL,NULL);
```

To ensure you have got your calculations correct, you can use the following simple formula:

Number of work-groups = (global_work_size[0]*...*global_work_size[n-1]) /
(local_work_size[0]*...*local_work_size[n-1])

Next, we are going to take a look at how we can enable this task-parallelism and data-parallelism to be processed by the CPU and GPU where each device will copy a one-dimensional data array from the input buffer and treat it like a two-dimensional matrix for parallel computing, and finally output the results to a one-dimensional matrix.

www.EngineeringBooksPdf.com

Getting ready

In `Ch2/work_partition/work_partition.c`, we saw an excerpt where we need to copy a million elements from an input buffer to an output buffer using a two-dimensional data format. We proceed to partition the data into a 1024 x 1024 matrix where each work item processes a single cell and we create work groups of the size 64 x 2 matrix.

Caveat—during my experimentation, this program crashed when executing on the OSX 10.6 Intel Core i5 with OpenCL 1.0 as the work group can only be of size one in each dimension. We'll look in the *Chapter 3*, *Understanding OpenCL Data Types* on how to make our programs more portable.

The kernel function, `copy2Dfloat4` is a typical function which is executed on the device and we like to express the idea of transferring a vector of elements from one point to another and once that's done, the application will conduct a data sanity check which will pass or fail the program; Refer to the `Ch2/work_partition/work_partition.cl`.

How to do it...

We've included the main part of this recipe, with the highlighted commentary in the following code:

```
// --------- file: work_partition.cl --------------
#define WIDTH 1024
#define DATA_TYPE float4
/*
  The following macros are convenience 'functions'
  for striding across a 2-D array of coordinates (x,y)
  by a factor which happens to be the width of the block
  i.e. WIDTH
*/
#define A(x,y) A[(x)* WIDTH + (y)]
#define C(x,y) C[(x)* WIDTH + (y)]
__kernel void copy2Dfloat4(__global DATA_TYPE *A, __global DATA_TYPE
*C)
{
    int x = get_global_id(0);
    int y = get_global_id(1);
    // its like a vector load/store of 4 elements
    C(x,y) = A(x,y);
}
// --------- file: work_partition.c ---------
cl_float* h_in = (float*) malloc( sizeof(cl_float4) * DATA_SIZE); //
```


www.EngineeringBooksPdf.com

```
input to device
cl_float* h_out = (float*) malloc( sizeof(cl_float4) * DATA_SIZE); //
output from device
  for( int i = 0; i < DATA_SIZE; ++i) {
    h_in[i] = (float)i;
  }
// code omitted
cl_mem memInObj = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_
COPY_HOST_PTR, sizeof(cl_float4) * (DATA_SIZE), h_in, &error);
cl_mem memOutObj = clCreateBuffer(context,
                                  CL_MEM_WRITE_ONLY ,
                                  sizeof(cl_float4) * (DATA_SIZE),
                                  NULL, &error);
if(error != CL_SUCCESS) {
  perror("Can't create an output buffer object");
  exit(1);
}
/* Let OpenCL know that the kernel is suppose to receive two arguments
*/
error = clSetKernelArg(kernels[j], 0, sizeof(cl_mem), &memInObj);
if (error != CL_SUCCESS) {
  perror("Unable to set buffer object in kernel");
  exit(1);
}
error = clSetKernelArg(kernels[j], 1, sizeof(cl_mem), &memOutObj);
if (error != CL_SUCCESS) {
  perror("Unable to set buffer object in kernel");
  exit(1);
}
/* Enqueue the kernel to the command queue */
size_t globalThreads[2];
globalThreads[0]=1024;
globalThreads[1]=1024;
size_t localThreads[2];
localThreads[0] = 64;
localThreads[1] = 2;
cl_event evt;
error = clEnqueueNDRangeKernel(cQ,
                               kernels[j],
                               2,
                               0,
                               globalThreads,
                               localThreads,
                               0,
```


www.EngineeringBooksPdf.com

```
                                    NULL, &evt);
    clWaitForEvents(1, &evt);
    if (error != CL_SUCCESS) {
      perror("Unable to enqueue task to command-queue");
      exit(1);
    }
    clReleaseEvent(evt);
```

On OSX, you can run the following command:

```
gcc -std=c99 -Wall -DUNIX -g  -DAPPLE -arch i386 -o work_partition work_
partition.c   -framework OpenCL
```

On Ubuntu Linux 12.04 with Intel OpenCL SDK installed, you can run the following command:

```
gcc -std=c99 -Wall -DUNIX -g -DDEBUG -m64 -o work_partition work_
partition.c -I . -I /usr/include -L/usr/lib64/OpenCL/vendors/intel
-lintelocl -ltbb -ltbbmalloc -lcl_logger -ltask_executor
```

On Ubuntu Linux 12.04 with NVIDIA CUDA 5 installed, you can run the following command:

```
gcc -std=c99 -Wall -DUNIX -g -DDEBUG -m64 -o work_partition work_
partition.c -I. -I/usr/local/cuda/include  -lOpenCL
```

A binary executable named `work_partition` will be deposited on the directory.

On Ubuntu Linux 12.04 with AMD APP SDK v2.8 and NVIDIA CUDA 5 installed, I have the following output. If you ran the program using the Intel® OpenCL SDK, then you will not see the output related to the discrete graphics chip. In this example, we have demonstrated both coarse-grained and fine-grained data and task parallelism:

```
Number of OpenCL platforms found: 2
Number of detected OpenCL devices: 1
Running GPU
    => Kernel name: copy2Dfloat4 with arity: 2
    => About to create command queue and enqueue this kernel...
    => Task has been enqueued successfully!
Check passed!
Number of detected OpenCL devices: 1
Running on CPU ........
    => Kernel name: copy2Dfloat4 with arity: 2
    => About to create command queue and enqueue this kernel...
    => Task has been enqueued successfully!
Check passed!
```


www.EngineeringBooksPdf.com

How it works...

The host application allocates two buffers that are capable of storing a million elements of the data type `cl_float4`, which is a OpenCL `vector` data type. Next we proceed to build the program via `clBuildProgramWithSource` (refer to `Ch2/work_partition/work_partition.c`), and pick up all the kernels in the kernel file (`*.cl`). Each detected device will pick up a one-dimensional input buffer, transform it to a two-dimensional matrix, and partition the data among its parallel computing units where each work group will compute the following:

- Obtain the index for the row via `get_global_id(0)`; which can be thought of as the thread's ID in the x-axis
- Obtain the index for the column via `get_global_id(1)`; which can be thought of as the thread's ID in the y-axis
- Together with the row and column indexes, perform a memory load of 4 elements and store the same via `C(x,y) = A(x,y)`

The OpenCL runtime would have partition the data among the work groups, together with the IDs for the work items as well as work groups; hence there would not be a situation where the thread IDs being duplicated and hence waging mayhem on the computation (the OpenCL vendor has that responsibility of ensuring it doesn't occur). OpenCL knows how to do this because the dimensions of the input data, together with the number of work groups and number of executing work items are passed via the parameters `work_dim`, `global_work_size`, and `local_work_size` in the `clEnqueueNDRangeKernel` API.

An example should clarify this: Assume that the imaginary input data has two-dimensions and the `global_work_size` is 8192 and `local_work_size` is 16*16, then we will have 8192/(16*16) = 32 work groups; to be able to reference any element in a two-dimensional data block, you will write some code similar to this to generate the global thread ID in (this is not the only way to do this, but it is the generally preferred method):

```
int x = get_local_id(0);//x would range from 0 to 15
int y = get_local_id(1);//y would range from 0 to 15
int blockIdX = get_group_id(0);
int blockIdY = get_group_id(1);
int blockSizeX = get_local_size(0); // would return 16
int blockSizeY = get_local_size(1); // would return 16
uint globalThreadId = (blockIdx * blockSizeX + x) +
                      (blockIdY * blockSizeY + y);
```

The OpenCL kernel will complete its computation eventually because of an invocation to `clWaitForEvents` (we'll talk about this in the next chapter), and then the output buffer is stored with data from the device memory via `clEnqueueReadBuffer` and the data is sanity checked.

www.EngineeringBooksPdf.com

www.EngineeringBooksPdf.com

3
Understanding OpenCL Data Types

In this chapter, we are going to cover the following recipes:

- Initializing the OpenCL scalar data types
- Initializing the OpenCL vector data types
- Using OpenCL scalar types
- Understanding OpenCL vector types
- Vector and scalar address spaces
- Configuring your OpenCL projects to enable the double data type

Introduction

OpenCL supports a wide range of data types derived from the C programming language. They are widely classified into two groups called scalars and vectors. Scalars are basically elemental values, whereas vectors are a collection of elemental values and a good thing about vectors is that many OpenCL SDK vendors have provided automated vectorization which allows the values to be loaded into wide, that is, 128-bit, 256-bit, or 512-bit registers for consumption.

OpenCL scalar integral data types consists of the signed and unsigned types of `bool`, `char`, `short`, `int`, `long`, `uchar`, `ushort`, `uint`, and `ulong` respectively; for floating-point values there are `float`, `half`, and `double`. To represent those types in your host program, you have to just prepend the letters `cl_` to each type, which the OpenCL compiler will understand.

www.EngineeringBooksPdf.com

OpenCL vector data types consists of a multiple of scalar data integral and floating-point data types and they are `char<N>`, `short<N>`, `int<N>`, `long<N>`, `uchar<N>`, `ushort<N>`, `uint<N>`, `ulong<N>`, and `float<N>` where `<N>` represents a value of 2, 3, 4, 8, or 16. Similarly, you will represent those types in your host program by prepending the letters `cl_` to the data types.

In both the cases, if you prefer the explicit form of an `unsigned` type, then you can replace the letter `u` in the data types with the keyword `unsigned`.

Initializing the OpenCL scalar data types

In this recipe, we are going to demonstrate various ways to initialize scalar types, and most of the techniques will make a lot of sense if you already have programmed using the C programming language.

Getting ready

In addition to the regular data types defined in C which works in OpenCL, the standard have added a few more data types in addition to the ones we have mentioned in the previous section, and the following table illustrates them:

Type	Description
`half`	It is a 16-bit floating-point. The `half` data type must conform to the IEEE 754-2008 `half precision` storage format.
`bool`	It is a conditional data type that evaluates to true or false. The value true expands to an integer 1 while false expands to 0.
`size_t`	It is the unsigned integer type of the result of the sizeof operator. This can be a 32-bit or 64-bit unsigned integer.
`ptrdiff_t`	It is a 32-bit or 64-bit signed integer and usually it is used to represent the result of subtracting two points
`intptr_t`	It is a 32-bit or 64-bit signed integer with the property that any valid point to avoid can be converted to this type, and then converted back to point to void and the result will compare equal to the original pointer.
`uintptr_t`	It is a 32-bit or 64-bit unsigned integer that has got the same property as `intptr_t`.

www.EngineeringBooksPdf.com

OpenCL allows the following data types to be used interchangeably in your source codes:

Type in OpenCL	Type in application
bool	n/a
char	cl_char
unsigned char, uchar	cl_uchar
short	cl_short
unsigned short, ushort	cl_ushort
int	cl_int
unsigned int, uint	cl_uint
long	cl_long
unsigned long, ulong	cl_ulong
float	cl_float
double	cl_double
half	cl_half
size_t	n/a
ptrdiff_t	n/a
intptr_t	n/a
uintptr_t	n/a
void	void

So following are a few examples on how you can possibly declare and define scalar data types in your source code in the kernels and host:

```
float f = 1.0f;                  // In the OpenCL kernel
char c = 'a';                    // In the OpenCL kernel
const char* cs = "hello world\n";
cl_char c1 = 'b';                // in the host program
cl_float f1 = 1.0f;              // in the host program
const cl_char* css = "hello world\n";
```

In the previous chapter, *Understanding OpenCL Data Transfer and Partitioning*, we spent some time discussing about data types and how alignment works or in other words, how data misalignment can affect the performance. Scalar data types are always aligned to the size of the data type in bytes. Built-in data types whose sizes are not a power of two must be aligned to the next larger power of two. That is, a `char` variable will be aligned a 1-byte boundary, a `float` variable will be aligned to a 4-byte boundary.

www.EngineeringBooksPdf.com

How to do it...

If your application needs user-defined data types, then you need to place `__attribute__ ((aligned))` to those types; refer to the *Chapter 2*, *Understanding OpenCL Data Transfer and Partitioning* for more details.

In OpenCL, several operators convert operand values from one type to another, and this is commonly referred to as implicit conversions; another way is to apply a cast operation on operands or on the result of a binary operation. Implicit conversions between scalar built-in types are supported with the exception of `void` and `half` data types. What it means is shown in the following code:

```
cl_int x = 9;
cl_float y = x; // y will get the value 9.0
```

Or

```
int x = 9;
float y = x;  // y will get the value 9.0
```

You can use both forms in your application code. You can coerce a data type to another data type in OpenCL too, just as you can do in the C programming language. Refer to the following example:

```
float f = 1.0f;
int i = (int) f; // i would receive the value of 1
```

You can also coerce a scalar data type to a vector data type in OpenCL with the following code:

```
float f = 1.0f;
float4 vf = (float4)f; // vf is a vector with elements (1.0, 1.0, 1.0,
1.0)
uchar4 vtrue = (uchar4)true; // vtrue is a vector with elements(true,
true, true, true)
             // which is actually (0xff, 0xff, 0xff, 0xff)
```

Initializing the OpenCL vector data types

Vectors are extremely powerful to an OpenCL programmer because it allows the hardware to bulk load/store data to/from memory; such computations typically take advantage of the algorithms spatial and temporal locality properties. In this recipe, we are going to familiarize ourselves with creating various types of vectors.

www.EngineeringBooksPdf.com

Getting ready

You can initialize a vector in two primary manners and they are as follows:

- Vector literal
- Vector composition

Creating a vector literal simply means that you can construct your vector of whatever type you wish as shown in the following code:

```
float a = 1.0f;
float b = 2.0f;
float c = 3.0f;
float d = 4.0f;
float4 vf = (float4) (a, b, c, d);
//vf will store (1.0f, 2.0f, 3.0f, 4.0f)
```

Another way to initialize a vector is to do it via a scalar value as shown in the following code:

```
uint4 ui4 = (uint4)(9); // ui4 will store (9, 9, 9, 9)
```

You can also create vectors in the following fashion:

```
float4 f = (float4) ((float2) (1.1f, 2.2f),
                     (float2) (3.3f, 4.4f));
float4 f2 = (float4) (1.1f, (float2) (2.2f, 3.3f), 4.4f);
```

The data type on the left-hand-side and right-hand-side must be same or the OpenCL compiler will issue a complaint.

How to do it...

Vectors have another remarkable property, and that is, you can access the individual components through indexes, that is to say if you wish to access each component of a `float4` vector, `v`, then you would do so via `v.x`, `v.y`, `v.z`, `v.w` respectively, and for larger vectors of 8 or 16 elements we would access those individual elements via `v.s0`, `v.s1` through to `v.s7`, and `v.s0`, `v.s1`, `v.sa` through to `v.sf` respectively. Hence, vectors of type `char2`, `uchar2`, `short2`, `ushort2`, `int2`, `uint2`, `long2`, `ulong2`, and `float2` can access their `.xy` elements.

Following is another way of creating vectors and that is through composition:

```
float4 c;
c.xyzw = (float4) (1.0f, 2.0f, 3.0f, 4.0f);
float4 d;
d.x = c.x;
d.y = c.y;
d.z = c.z;
d.w = c.w; // d stores (1.0f, 2.0f, 3.0f, 4.0f)
```


www.EngineeringBooksPdf.com

How it works...

On a similar note, you can use numerical indexes to reference the components in your vector and create vectors in turn. The following table shows a list of indexes for the various vector data types:

Vector components	Numeric indexes that can be used
2-component	`0, 1`
3-component	`0, 1, 2`
4-component	`0, 1, 2, 3`
8-component	`0, 1, 2, 3, 4, 5, 6, 7`
16-component	`0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F`

To use these numerical indexes, you have to precede by the letter s or S, and following are a few quick examples on how to create vectors:

```
float4 pos = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
float4 swiz= pos.wzyx; // swiz = (4.0f, 3.0f, 2.0f, 1.0f)
float4 dup = pos.xxyy; // dup = (1.0f, 1.0f, 2.0f, 2.0f)
float4 f, a, b;
f.xyzw = a.s0123 + b.s0123;
```

There's more...

Lastly, vector data types can use the `.lo` (or `.even`) and `.hi` (or `.odd`) suffixes to compose new vector types, or to combine smaller vector types to a larger vector type. Multiple levels of `.lo` (or `.even`) and `.hi` (or `.odd`) suffixes can be used until they refer to a scalar term. The `.lo` and `.hi` suffix refers to the lower and upper halves of a vector. The `.even` and `.odd` suffixes of a vector refer to the even and odd elements of a vector. Following are the examples of vector creation via composition:

```
float4 vf = (float4) (1.0f, 2.0f, 3.0f, 4.0f);
float2 low = vf.lo; // returns vf.xy
float2 high = vf.hi; // returns vf.zw
float4 vf4 = (float4) (low, high);// returns (1.0f, 2.0f, 3.0f, 4.0f)
```

Vectors are disallowed from implicit conversions so you cannot perform the following operation:

```
float4 vf4, wf4;
int4 if4;
wf4 = vf4; // illegal
if4 = wf4; // illegal
```


www.EngineeringBooksPdf.com

Explicit casts between vector types are also disallowed, and in fact the only form of explicit cast to a vector type is when you're initializing a vector with a scalar:

```
float f = 4.4f;
float4 va = (float4) (f); // va stores ( 4.4f, 4.4f, 4.4f, 4.4f)
```

If you were to extract components of a 3-component vector type via the suffixes `.lo` (or `.even`), `.hi` (or `.odd`), then the 3-component vector type would behave as if it is a 4-component vector type with the exception that the `w` component would be undefined.

Using OpenCL scalar types

The scalar data types are quite similar to what you would expect if you were programming in the C language. However, two topics deserve more attention and we'll touch on that in this recipe; we'll look at the `half` data type and examine how OpenCL devices might order their data.

Getting ready

Many of the OpenCL compliant devices are actually little-endian architectures, and developers need to ensure that their kernels are tested on both big-endian and little-endian devices to ensure source compatibility with current and future devices. Let's use a simple example to illustrate endian-ness.

How to do it...

Consider a variable `x` that holds the value `0x01234567` and the address of `x` starts at `0x100`. In computer architecture terminology, the value `0x01` is the **most significant byte** (**MSB**) and `0x67` is the **least significant byte** (**LSB**). Big-endian storage scheme stores the MSB first till it meets the LSB and little-endian storage schemes stores the LSB first till it meets the MSB.

Big-endian

Address	0x100	0x101	0x102	0x103	
Values	0x01	0x23	0x45	0x67	...

Little-endian

Address	0x100	0x101	0x102	0x103	
Values	0x67	0x45	0x23	0x01	...

www.EngineeringBooksPdf.com

Review the full code listed in `Ch3/byte_ordering/show_bytes.c`, compile the code by running the commands `cmake` and `make` in that order; that will generate a binary named `ShowBytes`, and then run that program to see its output. This code will print out a series of output, and depending on the endian-ness of the architecture, you will notice different byte orderings.

Refer to the following code:

```
#include <stdio.h>
typedef unsigned char* byte_pointer;
void show_bytes(byte_pointer start, int len) {
  int i;
  for(i = 0; i < len; i++)
    printf(" %.2x", start[i]);
  printf("\n");
}
void show_int(int x) {
  show_bytes((byte_pointer) &x, sizeof(int));
}
void show_float(float x) {
  show_bytes((byte_pointer) &x, sizeof(float));
}
void show_pointer(void* x) {
  show_bytes((byte_pointer) &x, sizeof(void*));
}
void test_show_bytes(int val ) {
  int ival = val;
  float fval = (float) ival;
  int* pval = &ival;
  show_int(ival);
  show_float(fval);
  show_pointer(pval);
}
```

Since you've understood how byte ordering affects the way data (scalar) is being read and written; let's take a look at how the ordering affects vector data types in OpenCL. With vector data types, both, the order of the bytes within each value and the order of the values with respect to one another are reversed. Using an example of a `uint4` vector which contains the values `0x000102030405060708090A0B0C0D0E0F`, at address `0x100`, following table shows how a little-endian storage scheme would look:

0x100	0x104	0x108	0x1b0
0x0F0E0D0C	0x0B0A0908	0x07060504	0x3020100

www.EngineeringBooksPdf.com

Awareness of this fact is important if you are working with data compression and computer-imaging algorithms since these two classes of algorithms have a significant amount of byte-level operations and you don't want to be bitten by these issues.

How it works...

The `half-precision` data type, conveniently called `half` actually has half the storage and precision of a regular `float` type. The `half` type is IEEE754-2008 compliant and was first introduced by NVIDIA, and Industrial Light and Magic. The only thing you can do with this type is to declare a pointer to a buffer that contains `half` values; those values must be finite and normal numbers, de-normalized numbers, infinities, and NaN.

You can choose to use the vector load and store functions such as `vload_half`, `vload_halfn`, `vstore_half`, and so on. However, bear in mind that the load/store operation will create an intermediate floating -point value.

The `load` function read the `half` values from memory and converts it to a regular `float` value. The `store` functions take a `float` as an input, convert it to a `half` value and store that value into memory.

To determine if your device supports this, you can run the program in `Ch2/device_extension/device_extensions`, and the output should contain `cl_khr_fp16`; alternatively you can query the device by passing the parameter `CL_DEVICE_EXTENSIONS` to `clGetDeviceInfo`. Following is the code snippet from `Ch2/device_extensions/device_extensions.c`:

```
/* --- file: device_extensions.c --- */
displayDeviceDetails( devices[i], CL_DEVICE_EXTENSIONS, "CL_DEVICE_
EXTENSIONS");
void displayDeviceDetails(cl_device_id id,
                          cl_device_info param_name,
                          const char* paramNameAsStr) {
  cl_int error = 0;
  size_t paramSize = 0;
  error = clGetDeviceInfo( id, param_name, 0, NULL, &paramSize );
  if (error != CL_SUCCESS ) {
    perror("Unable to obtain device info for param\n");
    return;
  }
  /* the cl_device_info are preprocessor directives defined in cl.h */
  switch (param_name) {
// code omitted
    case CL_DEVICE_EXTENSIONS : {
```


www.EngineeringBooksPdf.com

```
// beware of buffer overflow; alternatively use the OpenCL C++ //
bindings
      char* extension_info[4096];
      error = clGetDeviceInfo( id, CL_DEVICE_EXTENSIONS,
sizeof(extension_info), extension_info, NULL);
      printf("\tSupported extensions: %s\n", extension_info);
    }break;
  } //end of switch
```

Understanding OpenCL vector types

When you start working through your OpenCL project you are inevitably going to use both the scalar and vector data types to model the algorithm. Scalars work like any variable declaration/definition you may have come across in most of the programming languages, and you should think of vectors as a wide container that can deliver all items in that container in parallel, and the one thing that differentiates scalars and vectors is the fact that when an operation is applied to a scalar, it affects just a single value while the same operation applied to a vector affects all items in it in parallel.

In the modern processors, there exist a specialized hardware unit that processes more data per cycle and they are often termed as **Single Instruction Multiple Data** (**SIMD**) or known as **Streaming SIMD Extensions** (**SSE**) which is intel's implementation of SIMD. The advantage that SIMD instructions provide is that they allow multiple values to be operated upon in a large register in a cycle; quite often there are many such units, thus increasing performance of the program. We should be clear that SIMD describes a mechanism that allows parallelism to occur gleaned from Flynn's taxonomy, while SSE describes how two CPU processor manufacturers namely, Intel and AMD implemented SIMD.

The first part of the story is to tell you how OpenCL kernels run on the CPUs before we reveal how it would work on the GPU, and for now we place our attention on the Intel CPU architecture. On these architectures, OpenCL sees a single device with multiple compute units and if you are guessing each core is a compute unit then you're correct and hence, your kernels run on all compute units unless you are using the device fission extension, which is new in OpenCL 1.2.

Device fission (`cl_khr_device_fission`) which is new in OpenCL 1.2 is currently supported by multicore CPUs by Intel, AMD, and IBM Cell Broadband. GPUs are currently not supported.

The next part of the story is to describe, how OpenCL kernels would run on GPUs manufactured by AMD, and we focus on the AMD GPU we used for this book which is based on AMD's Southern Island Architecture which includes their Radeon HD 7900, 7800, and 7700 GPUs; on a side note, you might wish to consult NVIDIA's website for more product details pertaining to their GPUs at `www.nvidia.com`.

www.EngineeringBooksPdf.com

Kernels basically execute instructions that are either scalar-based or vector-based, and work is assigned to a compute unit in blocks of 64 work items, which is termed as wavefront. A wavefront has a single program counter, and is considered as a small unit of work and what that means is that they execute in lock-step.

When your application passes workloads to the GPU, it must first compile the kernel and load it into memory. It must also bind buffers for the source and result data, and finally it would decide how to execute the given workload on the GPU. When the workload is to be executed, the GPU divides the input domain into blocks of 64 threads aka wavefronts and dispatches them to the **compute unit** (**CU**). The kernel is next fetched into the instruction cache and the compute unit begins dispatching instructions to the execution units; each compute unit can work on multiple wavefronts in parallel, simultaneously processing vector and scalar ALU computations, as well as memory accesses. The wavefront continues executing until the end of the kernel is reached, when the wavefront is terminated and a new one can take its place on the GPU.

Taking into account the fact that memory accesses by wavefronts happens in parallel, you will expect some sort of latency to occur and the processor is pretty clever in dealing with that situation, and what it does is executing many wavefronts in parallel and it works such that if one wavefront is waiting for results from the memory, other wavefronts can issue memory requests, and they can execute ALU operations in parallel with outstanding memory requests if and only if they are independent calculations. Factors that increase the amount of parallelism that can be extracted from the program varies, but one of them would be the actual number of hardware units available for parallel computation and in OpenCL terminology, it is known as the CU and in both CPUs and GPUs they are basically the processor.

A compute unit is the basis of parallel computation, and in the Southern Island Architecture which hosts other products, the number of compute units varies and each compute unit basically contains the following:

- Scalar ALU and scalar **GPRs** (**General-Purpose Registers**) aka **SGPRs**
- Four SIMDs, each consisting of a vector ALU and vector GPRs, aka VGPRs
- Local memory
- Read/write access to vector memory through a Level-1 cache
- Instruction cache, which is shared by four CUs, that is, compute units
- Constant cache, which is shared by four CUs, that is, compute units

www.EngineeringBooksPdf.com

Now we will focus on the vector operations on GPUs, which include ALU and memory operations. Each of the four SIMDs contains a vector-ALU that operates on wavefronts over four cycles; each SIMD also can host ten wavefronts in flight, that is, one CU can have forty wavefronts executing in parallel. In the AMD GPU based on the Southern Island Architecture used for this book which is the AMD HD 7870, we have 20 compute units and we know now that each CU holds four SIMDs and each SIMD would execute a wavefront means that we can have 20 x 4 x 10 x 64 = 51,200 work items at any one time, and if you were to imagine that each work item is at the stage of executing vector operations then the parallelism offered by GPUs is considerably larger than that of the CPU; the specific CPU we are referring to is the Intel Xeon Phi which has 60 cores and each core hosts 4 work items which provides 60 x 4 = 240 work items; be aware that we're not stating that GPUs are superior to CPUs since each device has its niche but we illustrate these numbers to demonstrate a simple fact that GPU has a higher throughput than the CPU.

Having said all that, we are going to see an example soon but first recall that vector operations are component-wise and that vectors can be accessed via numeric indexes, and each index can be combined into larger group of indices to perform a store/load to/from memory. Refer to the following code:

```
float4 v, u;
float f;
v = u + f;
// equivalent to
// v.x = u.x + f
// v.y = u.y + f
// v.z = u.z + f
// v.w = u.w + f
float4 a, b, c;
c = a + b
// equivalent to
// c.x = a.x + b.x
// c.y = a.y + b.y
// c.z = a.z + b.z
// c.w = a.w + b.w
```

The component-wise manner in which vectors can be aggregated to perform an operation without code verbosity actually helps the programmer in their daily work and increases productivity. Next, we can a dive into how vector types are translated to utilize your hardware.

Getting ready

The demonstration we are going to describe has two parts in it. First, we are going to use the Intel OpenCL compiler on Windows to demonstrate the implicit vectorization of the kernel code; secondly, we are going to demonstrate how to enable native vector type notation in your code to express the desire to generate vectorized code using the AMD APP SDK v2.7 or v2.8 on Linux.

www.EngineeringBooksPdf.com

We combined these two approaches with the intention to solve the problem of transferring a large input array from one part of the device memory to another part of the device memory, and finally we extract and compare them for equality. As before, we would prepare the data structures for transfers on the host code and write a suitable OpenCL kernel to actually transfer the memory contents across. The source can be found in `Ch3/vectorization`, and we build the program using the AMD APP SDK.

Readers who are interested in the OpenCL code generation for AMD CPU and GPU platforms should consult the `AMD CodeXL` product as the AMD APP Kernel Analyzer has been retired. You may wish to consult the AMD Intermediate Language Manual in conjunction when you study the intermediate language output.

Implicit vectorization is a required feature that is supported by all the compliant OpenCL compiler implementations, and the reason we chose to demonstrate this feature with the Intel OpenCL compiler is the fact that the generated SIMD instructions are more likely to be recognized by the reader than would the intermediate code generated by other compiler implementations such as AMD or NVIDIA's. The kernel code we have for you can be found in `Ch3/vectorization/vectorization.cl`, and we reveal it as in the following code:

```
__kernel void copyNPaste(__global float* in, __global float8* out) {
  size_t id = get_global_id(0);
  size_t index = id*sizeof(float8);
  float8 t = vload8(index, in);
  out[index].s0 = t.s0;
  out[index].s1 = t.s1;
  out[index].s2 = t.s2;
  out[index].s3 = t.s3;
  out[index].s4 = t.s4;
  out[index].s5 = t.s5;
  out[index].s6 = t.s6;
  out[index].s7 = t.s7;
}
```

This kernel's main action is to transfer the contents from one place to another, and it does this by transporting it in parallel using two vectors of eight floats each and you will notice that we use the vector component notation to state these memory transfers explicitly.

In the next demonstration, we swing from the kernel code back to the host code assuming that the developer has a desire to control the code generation in a more explicit manner; and this can be done through the native vector type notation.

www.EngineeringBooksPdf.com

We ask the reader to refer to the section *There's more...* for details, but the demonstration here rests on the assumption that the developer would like to hand tune the procedure that handles data validation once the memory transfers have been completed in the device, and this function can be found in `Ch3/vectorization/vectorization.c` named `valuesOK` and the following code is how it is implemented:

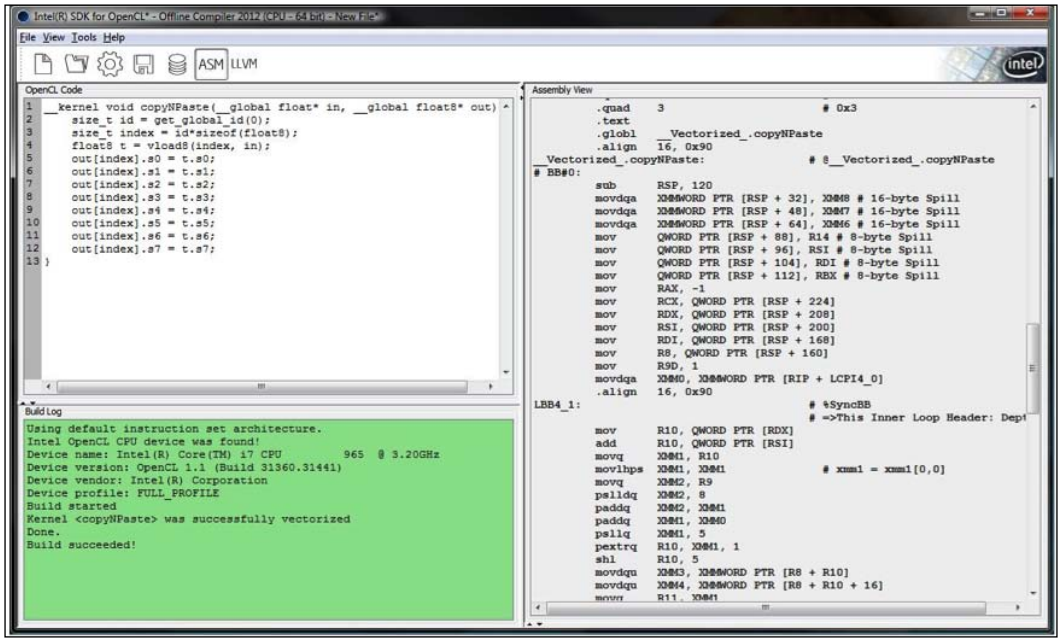
```
#ifdef __CL_FLOAT4__
int valuesOK(cl_float8* to, cl_float8* from, size_t length) {
#ifdef DEBUG
  printf("Checking data of size: %lu\n", length);
#endif
  for(int i = 0; i < length; ++i) {
#ifdef __SSE__
    __cl_float4 __toFirstValue = to->v4[0];
    __cl_float4 __toSecondValue = to->v4[1];
    __cl_float4 __fromFirstValue = from->v4[0];
    __cl_float4 __fromSecondValue = from->v4[1];
    __m128i vcmp = (__m128i) _mm_cmpneq_ps(__toFirstValue, __
fromFirstValue);
    uint16_t test = _mm_movemask_epi8(vcmp);
    __m128i vcmp_2 = (__m128i) _mm_cmpneq_ps(__toSecondValue, __
fromSecondValue);
    uint16_t test_2 = _mm_movemask_epi8(vcmp_2);
    if( (test|test_2) != 0 ) return 0; // indicative that the result
failed
#else
    #error "SSE not supported, which is required for example code to
work!"
#endif
  }
return 1;
}
#endif
```

How to do it...

Implicit vectorization through the Intel OpenCL compiler is relatively easy and the purpose of this simple example, we have chosen to install it on the Windows operating system. You can download the compiler from `http://software.intel.com/en-us/vcsource/tools/opencl`.

www.EngineeringBooksPdf.com

To witness how the implicit vectorization can be achieved through this compiler, you would copy and paste the kernel code (the previous code) into the editor pane of the GUI and start the compilation. Once compiled, you would be able to view the generated code by clicking on the **ASM** or **LLVM** buttons on the GUI. An example of this is shown in the following screenshot:



The next item is to hand-tune our data validation code, `valuesOK`, to exhibit vectorization. This example is only meant to illustrate how one would go about accomplishing something similar to this and you don't have to do anything besides invoking `make` in the directory `Ch3/vectorization`, and an executable vectorization will be dropped into the filesystem to which we'll next dissect it.

If you are running OpenCL 1.1 on Mac OSX 10.7, then passing the flag `-cl-auto-vectorizer-enable` to `clBuildProgram` as a build option will vectorize the kernels that will execute on the CPU. The SIMD instructions will be similar to the ones you see in this recipe.

www.EngineeringBooksPdf.com

Hand-tuning your code in such a manner basically turns implicit vectorization off, and you will need to judge for your scenario whether the effort justifies with respects to the complexity of the issue. To view the generated SIMD code, it would be best to put the program under a debugger, and on Linux the best debugger will be the GNU GDB. You basically load the program into the debugger and issue the command `disassemble /m valuesOK` to verify that the SIMD instructions were indeed generated. Following is a sample `gdb` session where the disassembly is interleaved with the source code:

```
$ gdb ./Vectorization
GNU gdb (GDB) 7.5-ubuntu
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later sa<http://gnu.org/licenses/
gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show
copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/tayboonl/PACKT_OpenCL_Book/src/Ch3/
vectorization/Vectorization...done.
(gdb) disassemble /m valuesOK
Dump of assembler code for function valuesOK:
warning: Source file is more recent than executable.
31 int valuesOK(cl_float8* to, cl_float8* from, size_t length) {
  0x000000000040117c <+0>: push  %rbp
  0x000000000040117d <+1>: mov  %rsp,%rbp
  0x0000000000401180 <+4>: sub  $0xf0,%rsp
  0x0000000000401187 <+11>: mov  %rdi,-0xd8(%rbp)
  0x000000000040118e <+18>: mov  %rsi,-0xe0(%rbp)
  0x0000000000401195 <+25>: mov  %rdx,-0xe8(%rbp)
32 #ifdef DEBUGf
33 printf("Checking data of size: %lu\n", length);
  0x000000000040119c <+32>: mov  -0xe8(%rbp),%rax
  0x00000000004011a3 <+39>: mov  %rax,%rsi
  0x00000000004011a6 <+42>: mov  $0x4020a8,%edi
  0x00000000004011ab <+47>: mov  $0x0,%eax
  0x00000000004011b0 <+52>: callq  0x400f20 <printf@plt>
34 #endif
35 for(int i = 0; i < length; ++i) {
  0x00000000004011b5 <+57>: movl  $0x0,-0xc4(%rbp)
  0x00000000004011bf <+67>: jmpq  0x4012a9 <valuesOK+301>
  0x00000000004012a2 <+294>: addl  $0x1,-0xc4(%rbp)
  0x00000000004012a9 <+301>: mov  -0xc4(%rbp),%eax
  0x00000000004012af <+307>: cltq
```


www.EngineeringBooksPdf.com

```
  0x00000000004012b1 <+309>: cmp  -0xe8(%rbp),%rax
  0x00000000004012b8 <+316>: jb  0x4011c4 <valuesOK+72>
36 #ifdef __SSE__
37    __cl_float4 __toFirstValue = to->v4[0];
  0x00000000004011c4 <+72>: mov  -0xd8(%rbp),%rax
  0x00000000004011cb <+79>: movaps (%rax),%xmm0
  0x00000000004011ce <+82>: movaps %xmm0,-0xc0(%rbp)
38    __cl_float4 __toSecondValue = to->v4[1];
  0x00000000004011d5 <+89>: mov  -0xd8(%rbp),%rax
  0x00000000004011dc <+96>: movaps 0x10(%rax),%xmm0
  0x00000000004011e0 <+100>: movaps %xmm0,-0xb0(%rbp)

39    __cl_float4 __fromFirstValue = from->v4[0];
  0x00000000004011e7 <+107>: mov  -0xe0(%rbp),%rax
  0x00000000004011ee <+114>: movaps (%rax),%xmm0
  x00000000004011f1 <+117>: movaps %xmm0,-0xa0(%rbp)
40    __cl_float4 __fromSecondValue = from->v4[1];
  0x00000000004011f8 <+124>: mov  -0xe0(%rbp),%rax
  0x00000000004011ff <+131>: movaps 0x10(%rax),%xmm0
  0x0000000000401203 <+135>: movaps %xmm0,-0x90(%rbp)
  0x000000000040120a <+142>: movaps -0xc0(%rbp),%xmm0
  0x0000000000401211 <+149>: movaps %xmm0,-0x60(%rbp)
---Type <return> to continue, or q <return> to quit---
  0x0000000000401215 <+153>: movaps -0xa0(%rbp),%xmm0
  0x000000000040121c <+160>: movaps %xmm0,-0x50(%rbp)
41    __m128i vcmp = (__m128i) _mm_cmpneq_ps(__toFirstValue, __
fromFirstValue);
  0x0000000000401229 <+173>: movdqa %xmm0,-0x80(%rbp)
  0x000000000040122e <+178>: movdqa -0x80(%rbp),%xmm0
  0x0000000000401233 <+183>: movdqa %xmm0,-0x40(%rbp)
42    uint16_t test = _mm_movemask_epi8(vcmp);
  0x0000000000401241 <+197>: mov  %ax,-0xc8(%rbp)
  0x0000000000401248 <+204>: movaps -0xb0(%rbp),%xmm0
  0x000000000040124f <+211>: movaps %xmm0,-0x30(%rbp)
  0x0000000000401253 <+215>: movaps -0x90(%rbp),%xmm0
  0x000000000040125a <+222>: movaps %xmm0,-0x20(%rbp)
43    __m128i vcmp_2 = (__m128i) _mm_cmpneq_ps(__toSecondValue, __
fromSecondValue);
  0x0000000000401267 <+235>: movdqa %xmm0,-0x70(%rbp)
  0x000000000040126c <+240>: movdqa -0x70(%rbp),%xmm0
  0x0000000000401271 <+245>: movdqa %xmm0,-0x10(%rbp)
44    uint16_t test_2 = _mm_movemask_epi8(vcmp_2);
  0x000000000040127f <+259>: mov  %ax,-0xc6(%rbp)
45    if( (test|test_2) != 0 ) return 0; // indicative that the result
```


www.EngineeringBooksPdf.com

```
failed
  0x0000000000401286 <+266>: movzwl -0xc6(%rbp),%eax
  0x000000000040128d <+273>: movzwl -0xc8(%rbp),%edx
  0x0000000000401294 <+280>: or  %edx,%eax
  0x0000000000401296 <+282>: test  %ax,%ax
  0x0000000000401299 <+285>: je  0x4012a2 <valuesOK+294>
  0x000000000040129b <+287>: mov  $0x0,%eax
  0x00000000004012a0 <+292>: jmp  0x4012c3 <valuesOK+327>
46 #else
47    #error "SSE not supported, which is required for example code to
work!"
48 #endif
49  }
50 return 1;
  0x00000000004012be <+322>: mov  $0x1,%eax
51  }
  0x00000000004012c3 <+327>: leaveq
  0x00000000004012c4 <+328>: retq
End of assembler dump
(gdb)
```

How it works...

Implicit vectorization is a piece of complicated software written into the compiler provided by the implementation, and is definitely hardware dependent and often represented by an **intermediate language** (**IL**) that's proprietary to the processor manufacturer and to our disappointment, not very well documented so we like to focus on how native vector type notation works in more detail.

The interested reader is however invited to explore the ILs developed by AMD and NVIDIA, which are known as AMD IL, and NVIDIA's PTX respectively.

This method of hand-tuning allows the developer to reference the built-in vector data type of the platform they're working on instead of relying on the OpenCL compiler to auto-vectorize the code, and may bring about performance benefits. The manner in which it is being done in OpenCL so far is to abstract these differences into platform dependent macros in the file `cl_platform.h`. Let's work out how this would work in our example.

The example, we saw previously, was tested on the Ubuntu Linux 12.04 operating system with an Intel Core i7 CPU and an AMD Radeon HD 7870 GPU, but since our example focuses on explicit vectorization on the host code, it implies that we need to know the width of SIMD vectors based on the Intel instruction set. We know this to be 128-bits and what this means is as follows:

www.EngineeringBooksPdf.com

```
float4 a,b;
float4 c = a + b;
```

The preceding code gets translated to the following C code snippet:

```
__m128 a, b;
__m128 c = __mm_add_ps(a, b);
```

The function `__mm_add_ps` is the SIMD function for adding two vectors by adding their single precision floating-point values component-wise in this manner and at first hand, it will look like syntax sugar but this is one of the many ways in which OpenCL provides cross platform compatibility and removes the pain of delivering customized vectorized code for various processor architectures so in this way a façade is actually a good thing.

Coming back to the problem we are trying to solve, which is to vectorize the procedure for performing data validation for the input and output arrays. In our example, we chose arrays or rather vectors that can contain 8 floats and what we will like to do is to examine them and compare them for equality. Using the native vector type notation in OpenCL, we know that the vector-of-8 can be decomposed into vector-of-4 elements because, OpenCL stipulates that if a platform can support a native vector type then the macro is identified in the `cl_platform.h` file by the name `__CL_<TYPEN>`, where `<TYPEN>` can be `UCHAR16`, `CHAR16`, `INT4`, `FLOAT4`, that is, the vectorized primitive types and in general, you can access the native components using the `.v<N>` subvector notation where `<N>` is the number of elements in the subvector.

Using this newly found information, we can dissect the program we saw previously with the fact that the memory content of the original host memory is represented by the `cl_float8 *` to while the copied memory contents from host to device are held by the `cl_float8*` from:

```
int valuesOK(cl_float8* to, cl_float8* from, size_t length) {
// code omitted
for(int i = 0; i < length; ++i) {
```

We need to iterate through the vectors in both input and output arrays and proceed to extract the first and second vector-of-4s from the host pointer as follows:

```
__cl_float4 __hostFirstValue = to->v4[0];
__cl_float4 __hostSecondValue = to->v4[1];
```

Then we extract the first and second vector-of-4s from the device pointer as follows:

```
__cl_float4 __deviceFirstValue = from->v4[0];
__cl_float4 __deviceSecondValue = from->v4[1];
```


www.EngineeringBooksPdf.com

Now, we compare each of the halves by using the SSE API `__mm_cmp_neq_ps`, and keep the result of each test into the variables test and test2 as shown in the following code:

```
    __m128i vcmp = (__m128i) _mm_cmpneq_ps(__hostFirstValue, __
deviceFirstValue);
    uint16_t test = _mm_movemask_epi8(vcmp);
    __m128i vcmp_2 = (__m128i) _mm_cmpneq_ps(__hostSecondValue, __
deviceSecondValue);
    uint16_t test_2 = _mm_movemask_epi8(vcmp_2);
```

Finally, we compare those results as follows:

```
    if( (test|test_2) != 0 ) return 0; // indicative that the result
failed
#else
```

There's more...

Another part of the vectorization story that we wanted to tell you is that you, the developer, has the option of controlling the auto-vectorization by providing an explicit compiler hint to the kernel code. This can be useful if you want to hand-tune the vectorization of your code.

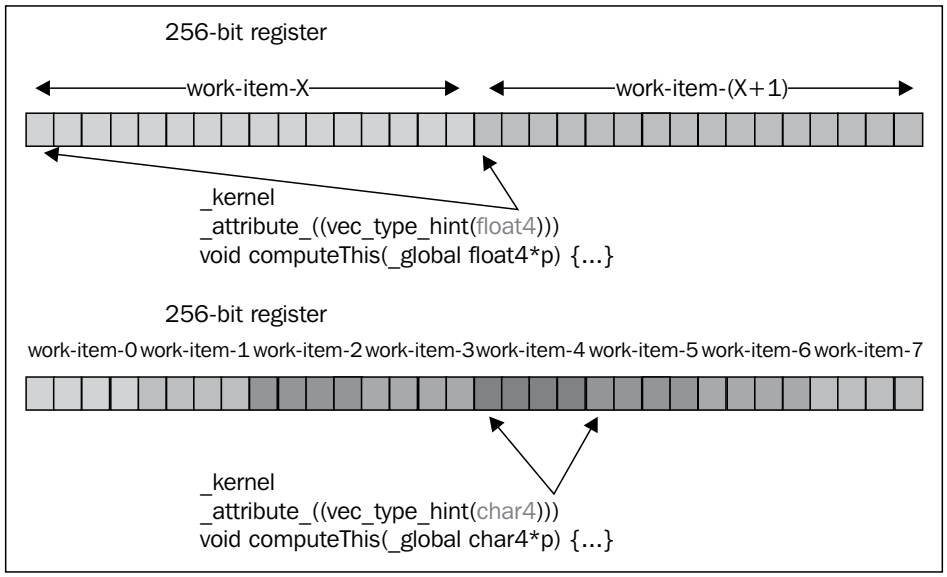
The compiler hint we are referring to is the `vec_type_hint(<type>)` where `<type>` is any of the built-in scalar or vector data types we mentioned previously. The attribute `vec_type_hint(<type>)` represents the computation width of the kernel and if it's not specified, the kernel is assumed to have the `vec_type_hint(int)` qualifier applied to the kernel, that is, 4-bytes wide. The following code snippets illustrate how the computation width of the kernel changes from 16-bytes to 8-bytes and finally to 4-bytes which happens to be the default:

```
// autovectoize assuming float4 as computation width
__kernel __attribute__((vec_type_hint(float4)))
void computeThis(__global float4*p ) {…}
// autovectorize assuming double as computation width
__kernel __attribute__((vec_type_hint(double)))
void computeThis(__global float4*p ) {…}
// autovectorize assuming int (default) as computation width
__kernel __attribute__((vec_type_hint(int)))
void computeThis(__global float4*p ) {…}
```

For you, the developer, to be able to use this, you will need to know the width of the vector units in your platform which could be running on a CPU or GPU. In the next diagram, we have two scenarios where we assume that both the `__kernel` functions are declared with `__attribute__((vec_type_hint(float4)))` and `__attribute__((vec_type_hint(char4)))` respectively. Furthermore, we assumed that the kernel is running on 256-bit wide registers and how the auto-vectorizer might choose to run one or more work items so that the register's usage is maximized; this is of course dependent on

www.EngineeringBooksPdf.com

the compiler's implementation. The following figure is a conceptual view of how the OpenCL compiler might generate work items to consume the data in the wide registers:



In the native vector type notation method for explicit vectorization, we mentioned that native vector types are identified in `cl_platform.h` by the `__CL_<TYPEN>__` preprocessor symbols aka C macros but, we haven't told you how we came to use the SSE instructions in the code example. Let's now find out why, and we need to reference the `cl_platform.h` defined by the OpenCL 1.2 standard (which you can download from `http://www.khronos.org/registry/cl/api/1.2/cl_platform.h`)

The code example was tested on the Ubuntu Linux 12.04 64-bit operating system with an Intel Core i7 CPU and a AMD Radeon HD 7870 GPU, and we should ignore the presence of the GPU as it has no relevance other than to inform you the machine setup.

What this setup tells us is that we have a SSE-capable instruction set and as a convention adopted by the UNIX and GCC community in general, is to look for the `__SSE__` preprocessor symbol and we indeed do that as follows:

```
#if defined(__SSE__)
#if defined(__MINGW64__)
#include <intrin.h>
#else
#include <xmmintrin.h>
#endif
#if defined(__GNUC__)
typedef float __cl_float4 __attribute__((vector_size(16)));
```


www.EngineeringBooksPdf.com

```
#else
typedef __m128 __cl_float4;// statement 1
#endif
#define __CL_FLOAT4__ 1// statement 2
#endif
```

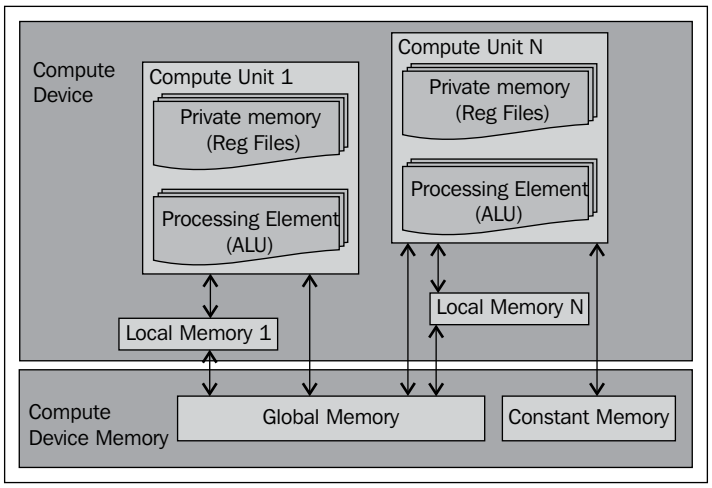
From the preceding code snippet, we know we should be focusing on the statement 1 as it has provided us the indicative width of the SIMD vectors, and we also know that by convention `__m128` indicates that its vector's width is 128-bits; other values includes 64-bits and 256-bits. We should also be careful to contain the explicit vectorization within the preprocessor guard, as a best practice, that is, `#ifdef __CL_FLOAT4__`.Using this understanding, we can proceed to search for the appropriate SSE APIs that allows us to manipulate data values of the desired width. The interested reader is invited to check the Intel Developer Manuals and AMD Developer Manuals, and explore how these ISAs compare and most importantly where they differ.

Vector and scalar address spaces

Now that we have understood how to use scalars and vectors in OpenCL, it's time to examine the OpenCL's defined four address spaces: `__global`, `__local`, `__constant`, and `__private` in which vectors and scalars can exist in. These spaces are mapped to the memory units and hence, limited by the actual resource on the device and define how work items can access memory.

Getting ready

Following is a conceptual diagram of the various memory domains:



www.EngineeringBooksPdf.com

The **Global Memory** and **Constant Memory** found in the lower-half of the preceding diagram corresponds to the `__global` and `__constant` domain. The **Local Memory** associated with each compute unit in OpenCL (that executes the kernel code) will have a memory space that's shared by all work items in the block which corresponds to the `__local` memory space while each processing element will have its own namespace to store data and, it is represented by the `__private` memory space. Be aware that there is no way in which a work item can access the (`__private`) memory space of another work item regardless of whether they're in the same work group or not, the same can be said of shared memory, that is, `__local` memory where no two work groups can inspect the other's memory.

Each compute unit in the device has a certain number of processing elements which executes work items and the compute unit as a whole would access the local, constant, or global memory space as determined by the computation. Each processing element (work group or work item), stores its own private variables in its private memory space.

How to do it...

The `__global` address space name is used to refer to memory objects allocated from the global memory pool. To determine the actual amount of resources available on the device, you need to pass the parameter `CL_DEVICE_GLOBAL_MEM_SIZE` to `clGetDeviceInfo`. The following snippet is drawn from `Ch2/device_details/device_details.c`:

```
displayDeviceDetails( devices[i], CL_DEVICE_GLOBAL_MEM_SIZE, "CL_
DEVICE_GLOBAL_MEM_SIZE");
void displayDeviceDetails(cl_device_id id,
                          cl_device_info param_name,
                          const char* paramNameAsStr) {
  cl_int error = 0;
  size_t paramSize = 0;
  error = clGetDeviceInfo( id, param_name, 0, NULL, &paramSize );
  if (error != CL_SUCCESS ) {
    perror("Unable to obtain device info for param\n");
    return;
  }
  /* the cl_device_info are preprocessor directives defined in cl.h */
  switch (param_name) {
    case CL_DEVICE_GLOBAL_MEM_SIZE:
    case CL_DEVICE_MAX_MEM_ALLOC_SIZE: {
      cl_ulong* size = (cl_ulong*) alloca(sizeof(cl_ulong) *
paramSize);
      error = clGetDeviceInfo( id, param_name, paramSize, size, NULL
);
      if (error != CL_SUCCESS ) {
        perror("Unable to obtain device name/vendor info for
param\n");
        return;
      }
```


www.EngineeringBooksPdf.com

The `__local` address space name is used to describe variables that need to be allocated in the local memory and shared by all work items of a work group. You can determine the maximum size of this space by passing the parameter `CL_DEVICE_MAX_LOCAL_MEM_SIZE` to `clGetDeviceInfo`.

The `__constant` address space name is used to describe non-mutable variables that need to be allocated as read-only in global memory, and can be read by all work items during the kernel's execution. You can determine the maximum size of this space by passing the parameter `CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE` to `clGetDeviceInfo`. This address space is useful if there is a specific value that does not change and is needed by the kernel functions.

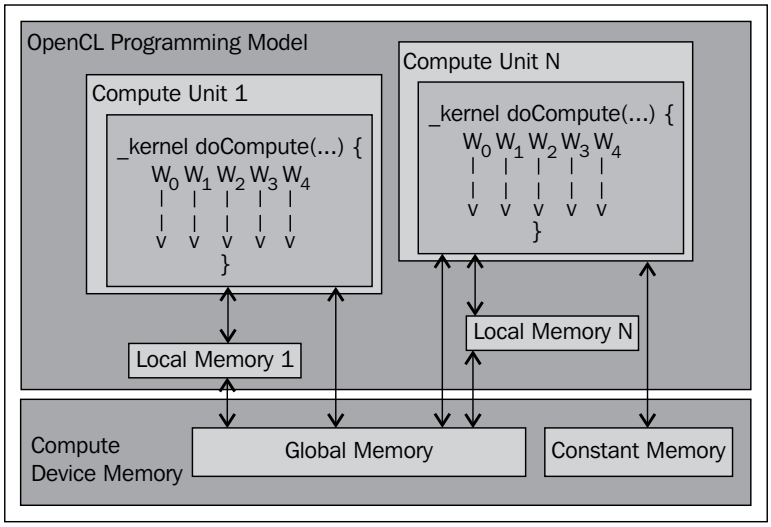
The `__private` address space is used to describe objects private-only distinct work items; hence work items cannot inspect one another's variables if they were marked by `__private`. By default, variables inside a kernel function not declared with any address space qualifiers such as: `__global`, `__local`, or `__constant` are marked `__private`; this includes all variables in the non-kernel functions and function arguments. The following kernel code from `Ch3/vectorization/vectorization.cl` will illustrate the global and private memory spaces whereby the variables `id`, `index`, and `t` are in the private memory space and hence not visible across other work items, therefore, free from interference, whereas the variables `in` and `out` exist in the global memory space and are visible by all work items:

```
__kernel void copyNPaste(__global float* in, __global float8* out) {
    size_t id = get_global_id(0);
    size_t index = id*sizeof(float8);
    float8 t = vload8(index, in);
    out[index].s0 = t.s0;
  //code omitted
  out[index].s7 = t.s7;
}
```

How it works...

The following diagram illustrates the OpenCL programming model:

www.EngineeringBooksPdf.com



Let's use the preceding diagram to understand how your kernel will function in OpenCL. Imagine you have a kernel named `doCompute` that takes several arguments that reference the global, constant, local, or private memory spaces. Work and data is divided among the kernels across the compute units represented by the $W_{0...4}$; they would represent either work groups (collection of work items) or work items.

Typically, computing in OpenCL often either involves individual work items performing the computation independently via the global, private, or constant spaces, or collecting these work items to form a work group so that they can load and store data more efficiently via utilizing the local memory space since that space allows sharing of data across all work items in the work group hence, preventing multiple memory loads from device memory.

Configuring your OpenCL projects to enable the double data type

Today's modern processors from Intel, AMD, and ARM have their floating-point units (FPUs) IEEE 754 compliant; however, ARM has both hardware and software support for half-precision numbers in addition to single-precision and double-precision numbers. Hence this implies that your OpenCL programs can actually utilize half-precision on ARM-based processors and this raise a question on how can one determine what sort of floating-point support does the device have.

www.EngineeringBooksPdf.com

The answer to that question is to query the device via the `clGetDeviceInfo` API and passing in any of the following parameters: `CL_DEVICE_SINGLE_FP_CONFIG`, `CL_DEVICE_DOUBLE_FP_CONFIG`, and `CL_DEVICE_HALF_FP_CONFIG` which identifies whether the device supports single-precision, double-precision, or half-precision number operations.

`CL_DEVICE_HALF_FP_CONFIG` and `CL_DEVICE_DOUBLE_FP_CONFIG` are not supported on Mac OSX 10.6 for OpenCL 1.0.

The result of API invocation returns an object of `cl_device_fp_config` type.

At the time of this writing, `CL_FP_SOFT_FLOAT` was not available on Mac OSX 10.6, but available on AMD APP SDK v2.7 and Intel OpenCL SDK.

In the case of the double-precision floating-point values, the OpenCL device extension, `cl_khr_fp64`, needs to be present before you can utilize the `double` data type in the kernel. As of OpenCL 1.2, the developer no longer has to query the device's extensions to verify the existence of the double-precision floating-point support, and we'll explain what you'll need to do in this case in the later part of this recipe.

As of OpenCL 1.1, the working committee does not mandate the support of the `double` data type except through the OpenCL 1.1 device extension `cl_khr_fp64`. If you are using AMD devices, you should know that AMD provides an extension that implements a subset of `cl_khr_fp64` and is known as `cl_amd_fp64`.

Let's understand this with a simple example.

Getting ready

In the upcoming example, the goal of the example is to illustrate the use of a `double` data type to hold the intermediate result of adding two `floats`, after which we send this `double` to be stored as a `float` in a result array. Take note that you cannot use the `double` type in the kernel code if the extension `cl_khr_fp64` or `cl_amd_fp64` (for AMD devices) is enabled.

The two test machines involved have `cl_khr_fp64` supported on the Intel Core i7 processor and a NVIDIA GPU but the ATI 6870x2 GPU doesn't support `cl_khr_fp64` or `cl_amd_fp64`.

www.EngineeringBooksPdf.com

How to do it...

Following is the code excerpt from `Ch3/double_support/double_support.cl`, which illustrates the kernel code:

```
#ifdef fp64
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
#endif
__kernel void add3(__global float* a, __global float* b, __global
float* out) {
  int id = get_global_id(0);
#ifdef fp64
  double d = (double)a[id] + (double)b[id];
  out[id] = d;
#else
  out[id] = a[id] + b[id];
#endif
}
```

Next, is the code snippet from `Ch3/double_support/double_support.c`, where it shows how to set the kernel arguments to the function `add3`:

```
// memobj1 & memobj2 refers to float arrays for consumption
// outObj refers to the output float array
error = clSetKernelArg(kernels[j], 0, sizeof(cl_mem), &memobj1);
error = clSetKernelArg(kernels[j], 1, sizeof(cl_mem), &memobj2);
error = clSetKernelArg(kernels[j], 2, sizeof(cl_mem), &outObj);
if (error != CL_SUCCESS) {
  perror("Unable to set buffer object in kernel arguments");
  exit(1);
}
/* Enqueue the kernel to the command queue */
size_t local[1] = {1};
size_t global[1] = {64};
error = clEnqueueNDRangeKernel(cQ, kernels[j], 1, NULL, global, local,
0, NULL, NULL);
if (error != CL_SUCCESS) {
  perror("Unable to enqueue task to command-queue");
  exit(1);}
```


www.EngineeringBooksPdf.com

To build the program with `CMake`, navigate to the directory `Ch3/double_support`, and enter `make`. It should drop a nice binary named `DoubleSupport` upon which you can execute it to observe the results. On both the test machines, the results for a small run, that is, 64-floating-point values are good with the runs on CPU and GPU.

```
Number of OpenCL platforms found: 1
Number of detected OpenCL devices: 2
Kernel name: add3 with arity: 3
About to create command queue and enqueue this kernel...
Task has been enqueued successfully!
Checking data of size: 64
Check passed!
Kernel name: add3 with arity: 3
About to create command queue and enqueue this kernel...
Task has been enqueued successfully!
Checking data of size: 64
Check passed!
```

The code in this example was constructed in such a manner that even if `double` wasn't supported the program will run. Upon inspecting the code, you will realize that its use case was to hold the result of adding two `float` values (which by intention will not overflow) but in other situations, you might want to use `double`s, and the conditional-directives, that is, `#ifdef`, `#else`, and `#endif` used to check for the presence of double floating-point support for the device and it is a standard technique.

How it works...

The type, `cl_device_fp_config` is actually composed of several values (shown in the following table) and you can determine whether a particular feature is supported or not by performing a bitwise-AND operation and for example, if we wish to determine which rounding modes are supported in double-precision operations then, we will have the following code:

```
cl_device_fp_config config;
clGetDeviceInfo( deviceId, CL_DEVICE_DOUBLE_FP_CONFIG, sizeof(config),
&config, NULL);
if (config & CL_FP_ROUND_TO_NEAREST) printf("Round to nearest is
supported on the device!");
```


www.EngineeringBooksPdf.com

Parameter	float	double	half
CL_FP_DENORM	Optional	Supported	Optional
CL_FP_INF_NAN	Supported	Supported	Supported
CL_FP_ROUND_TO_NEAREST	Supported	Supported	Optional
CL_FP_ROUND_TO_ZERO	Optional	Supported	Supported
CL_FP_ROUND_TO_INF	Optional	Supported	Supported
CL_FP_FMA	Optional	Supported	Optional
CL_FP_SOFT_FLOAT	Optional	Optional	Optional

For those who are inclined to use OpenCL 1.2, the specification has made double-precision an optional feature instead of an extension, and this means that instead of checking for the existence of the extensions `cl_khr_fp64` or `cl_amd_fp64` in the device, you will simply check that the returned value of the call to `clGetDeviceInfo` when passed the parameter `CL_DEVICE_PREFERRED_VECTOR_WIDTH_DOUBLE` and `CL_DEVICE_NATIVE_VECTOR_WIDTH` must be equal to `1` if the device were to support double-precision. The following code snippet illustrates how to check for the preferred native vector width size for built-in scalar types that can be put into vectors:

```
cl_uint vectorWidth;
size_t returned_size;
clGetDeviceInfo( deviceId, CL_DEVICE_PREFERRED_VECTOR_WIDTH_
DOUBLE,sizeof(cl_uint), &vectorWidth, &returned_size);
if(vectorWidth > 0) printf("Vectors of size %d for 'double' are:",
vectorWidth);
```


www.EngineeringBooksPdf.com

www.EngineeringBooksPdf.com

4
Using OpenCL Functions

In this chapter, we'll cover the following recipes:

- Storing vectors to an array
- Loading vectors from an array
- Using geometric functions
- Using integer functions
- Using floating-point functions
- Using trigonometric functions
- Arithmetic and rounding in OpenCL
- Using the shuffle function in OpenCL
- Using the select function in OpenCL

Introduction

In this chapter, we are going to explore how to utilize the common functions provided by OpenCL in your code. The functions we are examining would be mostly mathematical operations applied to the elements, and in particular applied to a vector of elements. Recall that the vectors are OpenCL's primary way to allow multiple elements to be processed on your hardware. As the OpenCL vendor can often produce vectorized hardware instructions to efficiently load and store such elements, try to use them as much as possible.

www.EngineeringBooksPdf.com

In detail, we are going to take a dive into how the following works:

- Data load and store functions for vectors
- Geometric functions
- Integer functions
- Floating-point functions
- Trigonometric functions

Finally, we will present two sections on how the OpenCL's `shuffle` and `select` functions would work if you choose to use them in your applications.

Storing vectors to an array

In the previous chapters, you caught glimpses of how we use vectors in various ways from a tool to transport data in an efficient manner to the device and from the device. We have also learned that OpenCL provides a substantial amount of functions that actually work on vectors. In this section, we will explore how we can store vectors to an array (when we use arrays in this context with a vector, we mean an array that contains scalar values).

The `vstore<N>` functions, where `<N>` is `2`, `3`, `4`, `8`, and `16`, are the primary functions you will use to actually signal the OpenCL that you wish to store the elements in your vector that has to be transported in a parallel fashion to a destination; this is often a scalar array or another vector.

We should be clear that `gentypeN` is not a C-like type alias for a data type, but rather a logical placeholder for the types such as `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, and `double`. The `N` stands for whether it is a data structure that aggregates `2`, `3`, `4`, `8`, or `16` elements. Remember that if you wish to store vectors of the type `double`, then you need to ensure that the directive `#pragma OPENCL EXTENSION cl_khr_fp64 : enable` is in your code before any `double` precision data type is declared in the kernel code.

Hence, the `vstoreN` API will write `sizeof(gentypeN)` bytes given by the data to the address `(p + (offset *N))`. The address computed as `(p + (offset * N))` must be 8-bit aligned if `gentype` is `char` or `uchar`; 16-bit aligned if `gentype` is `short` or `ushort`; 32-bit aligned if `gentype` is `int` or `uint`; 64-bit aligned if `gentype` is `long`, `ulong` or `double`.

You should notice that the memory writes can span from the global memory space (`__global`) to local (`__local`), or even to a work item private memory space (`__private`) but never to a constant memory space (`__constant` is read-only). Depending on your algorithm, you may need to coordinate the writes to another memory space with memory barriers otherwise known as fences.

www.EngineeringBooksPdf.com

The reason why you will need memory barriers or fences is that the memory reads and writes, in general, can be out of order, and the main reason for this is that the compiler optimization of the source code re-orders the instructions so that it can take advantage of the hardware.

To expand on that idea a little, you might be aware that C++ has a keyword, `volatile`, which is used to mark a variable so that the compiler optimizations generally do not apply optimized load-stores to any use of that variable; and basically any use of such variable typically involves a load-use-store cycle at every use-site also known as sequence points.

Loop unrolling is an optimization technique where the compiler attempts to remove branching in the code and hence, emitting any branch predication instructions so that the code executes efficiently. In the loops that you are accustomed to, you often find an expression as follows:

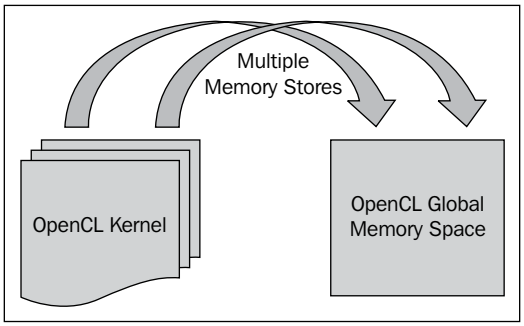
```
for(int i = 0; i  < n; ++i ) { ... }
```

What happens here is that when this code is compiled, you will notice that the ISA will issue an instruction to compare the value of `i` against that of `n`, and based on the result of that comparison, perform certain actions. Branching occurs when the executing thread takes a path if the condition is true or another path if the condition is false. Typically, a CPU executes both paths concurrently until it knows with a 100 percent certainty that it should take one of these paths, and the CPU can either dump the other unused path or it needs to backtrack its execution. In either case, you will lose several CPU cycles when this happens. Therefore, the developer can help the compiler and in our case, give a hint to the compiler what the value of `n` should be so that the compiler doesn't have to generate code to check for `i < n`. Unfortunately, OpenCL 1.2 doesn't support loop unrolling as an extension, but rather the AMD APP SDK and CUDA toolkits provide the following C directives:

```
#pragma unroll <unroll-factor>
#pragma unroll 10
for(int i = 0; i < n; ++i) { ... }
```


www.EngineeringBooksPdf.com

Without these functions, the OpenCL kernel would potentially issue a memory load-store for each processed element as illustrated by the following diagram:



Let's build a simple example of how we can use these `vstoreN` functions in a simple example.

Getting ready

This recipe will show you a code snippet from `Ch4/simple_vector_store/simple_vector_store.cl` where a vector of 16 elements is loaded in and subsequently copied by using `vstore16(...)`. This API isn't exactly sugar syntax for a loop unrolling of 16 elements, and the reason is the compiler generates instructions that loads a vector of 16 elements from memory; also loop unrolling doesn't exist in OpenCL 1.1 as we know it but, it doesn't hurt to think in terms of that if it helps in understanding the concept behind the `vstoreN` APIs.

How to do it...

The following is the kernel code where we will demonstrate the data transfers:

```
//
// This kernel loads 64-elements using a single thread/work-item
// into its __private memory space and writes it back out
__kernel void wideDataTransfer(__global float* in,
  __global float* out) {
    size_t id = get_global_id(0);
    size_t offsetA = id ;
    size_t offsetB = (id+1);
    size_t offsetC = (id+2);
    size_t offsetD = (id+3);

    // each work-item loads 64-elements
    float16 A = vload16(offsetA, in);
    float16 B = vload16(offsetB, in);
    float16 C = vload16(offsetC, in);
```


www.EngineeringBooksPdf.com

```
        float16 D = vload16(offsetD, in);

        vstore16(A, offsetA, out);
        vstore16(B, offsetB, out);
        vstore16(C, offsetC, out);
        vstore16(D, offsetD, out);
    }
```

To compile it on the OS X platform, you will have to run a compile command similar to this:

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

Alternatively, you can type `make` in the source directory `Ch4/simple_vector_store/`. When that happens, you will have a binary executable named `VectorStore`.

To run the program on OS X, simply execute the program `VectorStore` and you should either see the output: `Check passed!` or `Check failed!` as follows:

```
Check passed!
```

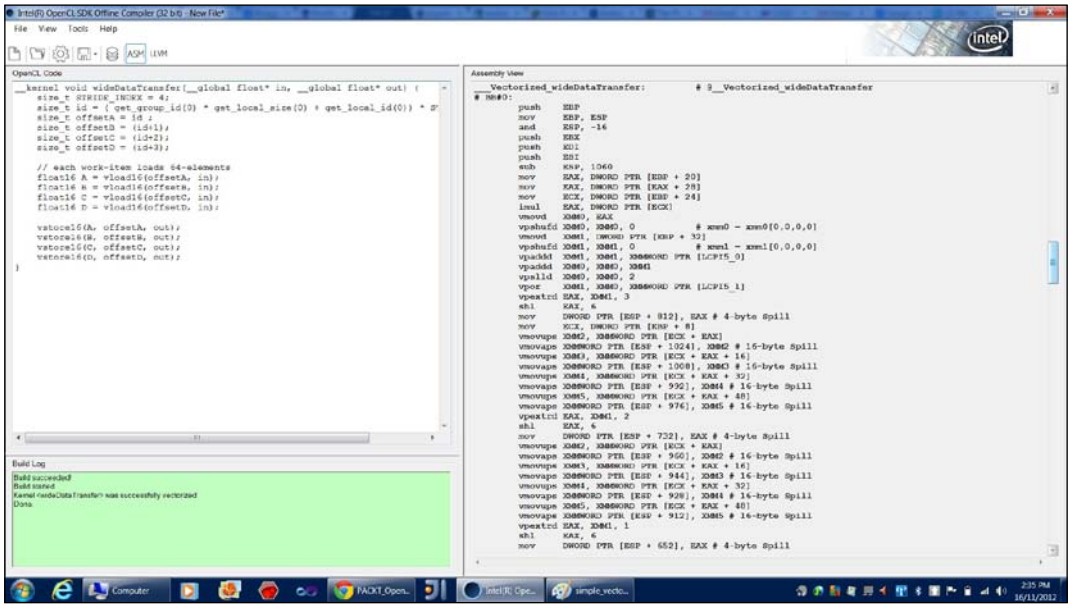
How it works...

This code can be understood from the perspective that a large vector exists in the global memory space, and our attempt is to load the vector into a variable in the private memory, that is, each work item has a unique variable named `t`; do nothing to it and store it back out to another in-memory array that is present in the global memory space.

In case you are curious about how this works, the memory writes are actually coalesced so that the writes are issued in bursts of bytes. The size of this burst is dependent on the hardware's internal architecture. As a concrete example in AMD's ATI GPUs, these memory writes are issued once every 16 writes are known to occur and it is related to the implementation of work items in the GPU. You see that it's very inefficient for the GPU to issue a read or write for every work item. When you combine this with the fact that there could be potentially hundreds of thousands of computing threads active in a clustered GPU solution, you can imagine the complexity is unfathomable if the manufacturers were to implement a logic that allows the developer to manage the programs on a work item/per-thread granularity. Hence graphic card manufacturers have decided that it is more efficient to implement the graphical cards to execute a group of threads in lock-step. ATI calls this group of executing threads a wave-front and NVIDIA calls it a warp. This understanding is critical when you start to develop nontrivial algorithms on your OpenCL device.

www.EngineeringBooksPdf.com

When you build the sample application and run it, it doesn't do anything in particularly special from what we have seen but it is useful to see how the underlying code is generated, and in this example the Intel OpenCL SDK is illustrative.



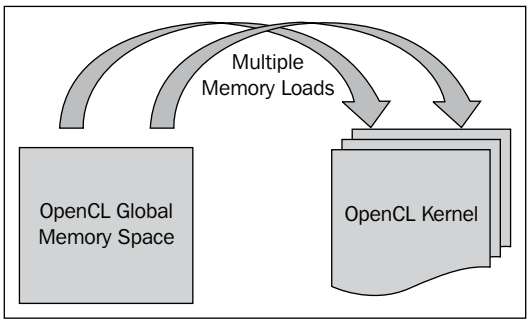
The assembly code snippet in particular is that of the resultant translation to **SSE2/3/4** or **Intel AVX** (**Advanced Vector Extensions**) code.

Loading vectors from an array

The `vloadN` functions are typically used to load multiple elements from an in-memory array to a destination in-memory data structure and are often a vector. Similar to the `vstoreN` functions, the `vloadN` functions also load elements from the global (`__global`), local (`__local`), work item private (`__private`), and finally constant memory spaces (`__constant`).

We should be clear that `gentypeN` is not a C-like type alias for a data type but rather a logical placeholder for the types: `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, or `double` and the `N` stands for whether it's a data structure that aggregates `2`, `3`, `4`, `8`, or `16` elements. Without this function, the kernel needs to issue potentially multiple memory loads as illustrated by the following diagram:

www.EngineeringBooksPdf.com



Getting ready

The following is an excerpt from `Ch4/simple_vector_load/simple_vector_load.cl`. We focus our attention to understand how to load vectors of elements from the device memory space for computation within the device, that is, CPU/GPU. But this time round, we use an optimization technique called **prefetching** (its warming up the cache when your code is going to make use of the data soon and you want it to be near also known as spatial and temporal locality), and is typically used to assign to local memory space so that all work items can read the data off the cache without flooding requests onto the bus.

How to do it...

The following is the kernel code from which we shall draw our inspiration:

```
__kernel void wideDataTransfer(__global float* in,
  __global float* out) {
  size_t id = get_group_id(0) * get_local_size(0) +
    get_local_id(0);
  size_t STRIDE = 16;
  size_t offsetA = id;
  prefetch(in + (id*64), 64);
  barrier(CLK_LOCAL_MEM_FENCE);

  float16 A = vload16(offsetA, in);
  float a[16];
  a[0] = A.s0;
  a[1] = A.s1;
  a[2] = A.s2;
  a[3] = A.s3;
  a[4] = A.s4;
  a[5] = A.s5;
  a[6] = A.s6;
  a[7] = A.s7;
```


www.EngineeringBooksPdf.com

```
    a[8] = A.s8;
    a[9] = A.s9;
    a[10] = A.sa;
    a[11] = A.sb;
    a[12] = A.sc;
    a[13] = A.sd;
    a[14] = A.se;
    a[15] = A.sf;
    for( int i = 0; i < 16; ++i ) {
      out[offsetA*STRIDE+i] = a[i];
    }
  }
```

To compile it on the OS X platform, you will have to run a compile command similar to this:

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

Alternatively, you can type `make` in the source directory `Ch4/simple_vector_load/`. When that happens, you will have a binary executable named `VectorLoad`.

To run the program on OS X, simply execute the program `VectorLoad` and you should either see the output: `Check passed!` or `Check failed!` as follows:

```
Check passed!
```

How it works...

The kernel would proceed to prefetch the `16` values of type `float` from the `__global` memory space to the global cache via the first work item in the work group, which would ultimately arrive in the work item's `__private` memory space via the `vload16` API. Once that value is loaded, we can assign individual floats to the array and finally output them to the destination via an explicit write to the `__global` memory space of `out`. This is one method in which you can conduct memory load from a scalar array that resides in the global memory space.

```
prefetch(in +(id*64), 64);
```

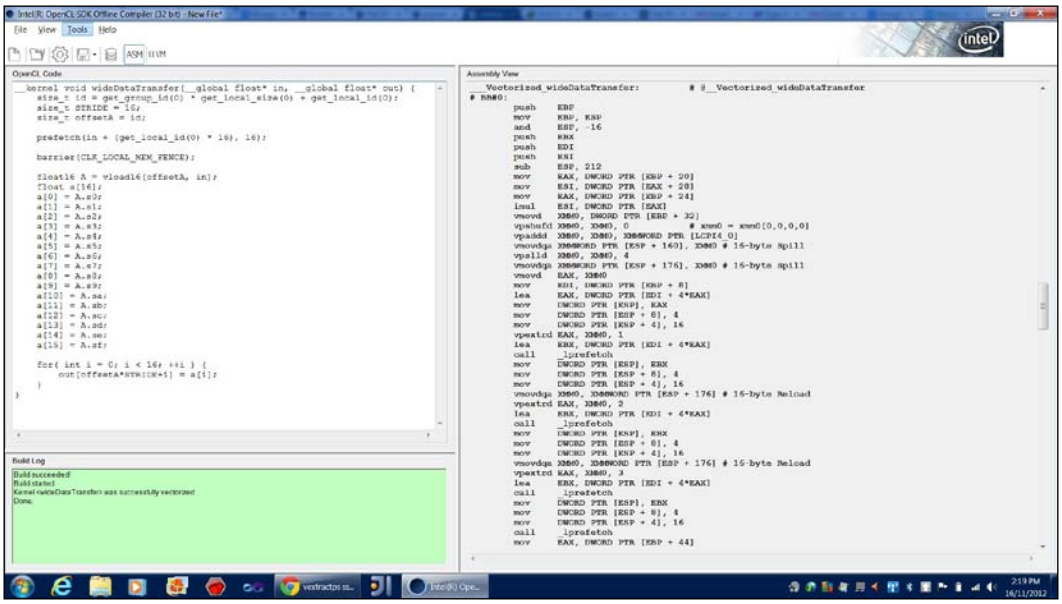
The preceding line is an optimization technique used to improve data reuse by making it available before it is required; this prefetch instruction is applied to a work item in a work group and we've chosen the first work item in each work group to carry this out. In algorithms where there is heavy data reuse, the benefits would be more significant than the following example:

Another thing you may have noticed is that we didn't write the following code:

```
out[offset*STRIDE + i] = A; // 'A' is a vector of 16 floats
```


www.EngineeringBooksPdf.com

The reason why we did not do this is because OpenCL forbids the implicit/explicit conversion of a vector type to a scalar.



One interesting thing that is worth pointing out other than the generated SSE instructions is the fact that multiple hardware prefetch instructions are generated, even though the code only mentions one prefetch instruction. This is the sort of façade that allows OpenCL vendors to implement the functionality based on an open standard, while still allowing the vendors to hide the actual implementation details from the developer.

Using geometric functions

The geometric functions are used by the programmers to perform common computation on vectors, for example, cross or dot products, normalizing a vector, and length of a vector. To recap a little about vector cross and dot products, remember that a vector in the mathematical sense represents a quantity that has both direction and magnitude, and these vectors are used extensively in computer graphics.

Quite often, we need to compute the distance (in degrees or radians) between two vectors and to do this, we need to compute the dot product, which is defined as:

$$a.b = \|a\|\|b\|\cos\theta$$

www.EngineeringBooksPdf.com

It follows that if *a* is perpendicular to *b* then it must be that *a . b = 0*. The dot product is also used to compute the matrix-vector multiplication which solves a class of problems known as **linear systems**. Cross products of two 3D vectors will produce a vector that is perpendicular to both of them and can be defined as:

$$\|u \times v\| = \|u\|\|v\| \sin\theta$$

The difference between these products is the fact that the dot product produces a scalar value while the cross product produces a vector value.

The following is a list of OpenCL's geometric functions:

Function	Description
`float4 cross(float4 m, float4 n)` `float3 cross(float3 m, float3 n)`	Returns the cross product of `m.xyz` and `n.xyz` and the `w` component in the result vector is always zero
`float dot(floatn m, floatn n)`	Returns the dot product of two vectors
`float distance(floatn m, floatn n)`	Returns the distance between `m` and `n`. This is computed as:`length(m - n)`
`float length(floatn p)`	Return the length of the vector `p`
`floatn normalize(floatn p)`	Returns a vector in the same direction as `p` but with a length of `1`
`float fast_distance(floatn p0, floatn p1)`	Returns `fast_length(p0 - p1)`
`float fast_length(floatn p)`	Returns the length of vector `p` computed as: `half_sqrt()`
`floatn fast_normalize(floatn p)`	Returns a vector in the same direction as `p` but with a length of `1`. `fast_normalize` is computed as: `p * half_sqrt()`

You should be aware that these functions are implemented in OpenCL using the *round to nearest even* rounding mode also known as **rte-mode**.

Next, let's take a look at an example that utilizes some of these functions.

www.EngineeringBooksPdf.com

Getting ready

The code snippet in `Ch4/simple_dot_product/matvecmult.cl` illustrates how to compute the dot product between a 2D vector and a matrix and write back the result of that computation to the output array. When you are starting out with OpenCL, there might be two probable ways in which you will write this functionality, and I think it is instructive to discover what the differences are; however we only show the relevant code snippet that demonstrates the dot API.

How to do it...

The following is the simplest implementation of the matrix dot product operation:

```
__kernel void MatVecMultUsingDotFn(__global float4* matrix,
  __global float4* vector, __global float* result) {
    int i = get_global_id(0);
    result[i] = dot(matrix[i], vector[0]);
}
```

To compile this on the OS X platform, you will have to run a compile command similar to this:

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

Alternatively, you can type `make` in the source directory `Ch4/simple_dot_product/`. When that happens, you will have a binary executable named `MatVecMult`.

To run the program on OS X, simply execute the program `MatVecMult` and you should either see the output: `Check passed!` or `Check failed!` as follows:

```
Check passed!
```

How it works...

The previous code snippet is probably the simplest you will want to write to implement the matrix dot product operation. The kernel actually reads a vector of `4` floats from the `__global` memory spaces of both inputs, computes the dot product between them, and writes it back out to `__global` memory space of the destination. Previously, we mentioned that there might be another way to write this. Yes, there is and the relevant code is shown as follows:

```
__kernel void MatVecMult(const __global float* M,
  const __global float* V, uint width, uint height,
    __global float* W) {
    // Row index
    uint y = get_global_id(0);
    if (y < height) {
       // Row pointer
```


www.EngineeringBooksPdf.com

```
        const __global float* row = M + y * width;
        // Compute dot product
        float dotProduct = 0;
       for (int x = 0; x < width; ++x)
         dotProduct += row[x] * V[x];
     // Write result to global memory
     W[y] = dotProduct;
     }
}
```

When you compare this implementation without using the dot API, you will discover that you not only need to type more but also you will have increased the number of work item variables which happens to be in the `__private` memory space; often you don't want to do this because it hinders the code readability, and also quite importantly scalability because too many registers are consumed.

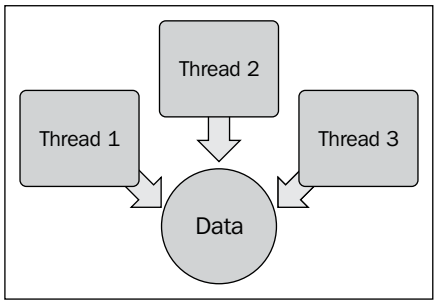
In OpenCL implementations, they would need to manage the available resources on the device, which could be available memory or available compute units. One such resource is the register file that contains a fixed number of general-purpose registers that the device has for executing one or many kernels. During the compilation of the OpenCL kernel, it will be determined how many registers are needed by each kernel for execution. An example would be where we assume that a kernel is developed that uses 10 variables in the `__private` memory space and the register file is `65536`, and that would imply that we can launch 65536 / 10 = 6553 work items to run our kernel. If you rewrite your kernel in such a way that uses more data sharing through the `__local` memory spaces, then you can free more registers and you can scale your kernel better.

Using integer functions

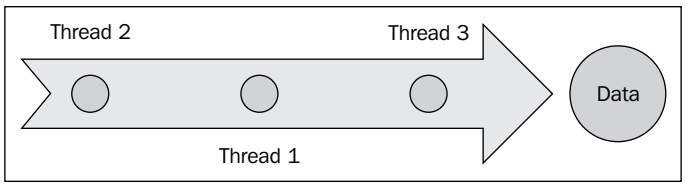
The integer functions in OpenCL primarily provides useful ways in which you can use them to perform the usual mathematical calculations such as obtaining an absolute value, halving a value, locating the minimum or maximum of three values, cyclic shift of a number, and specialized form of multiplication which is designed to work for a certain class of problems. Many of the functions that we have mentioned such as `min` and `max` do not perform the comparisons in an atomic fashion, but if you do like to ensure that, then a class of atomic functions can be used instead and we'll examine them later.

A class of integer functions is the atomic functions, which allows the developer to swap values (single-precision floating-point values too) in an atomic fashion, and some of these functions implements **CAS** (**Compare-And-Swap**) semantics. Typically, you may want to ensure some sort of atomicity to certain operations because without that, you will encounter race conditions.

www.EngineeringBooksPdf.com



The atomic functions typically take in two inputs (they have to be of integral types, only `atomic_xchg` supports single-precision floating-point types), where the first argument is a pointer to a memory location in the global (`__global`) and local (`__local`) memory spaces ,and they are typically annotated with the `volatile` keyword, which prevents the compiler from optimizing the instructions related to the use of the variable; this is important as the reads and writes could be out of order and could affect the correctness of the program. The following is an illustration of a mental model of how atomic operations serialize the access to a piece of shared data:



The following example, `atomic_add`, has two versions which work on signed or unsigned values:

```
int atomic_add(volatile __global int*p, int val)
unsigned int atomic_add(volatile __global uint*p, uint val)
```

Another observation you need to be aware of is the fact that just because you can apply atomicity to assert the correctness of certain values, it does not necessarily imply program correctness.

The reason why this is the case is due to the manner in which work items are implemented as we mentioned earlier in this chapter, that NVIDIA and ATI executes work items in groups known as work groups and each work group would contain multiple chunks of executing threads, otherwise, known as **warp** (32 threads) and **wavefront** (64 threads) respectively. Hence when a work group executes on a kernel, all the work items in that group are executing in lock-step and normally this isn't a problem. The problem arises when the work group is large enough to contain more than one warp/wavefront; then you have a situation where one warp/wavefront executes slower than another and this can be a big issue.

www.EngineeringBooksPdf.com

The real issue is that the memory ordering cannot be enforced across all compliant OpenCL devices; so the only way to tell the kernel that we like the loads and stores to be coordinated is by putting a memory barrier at certain points in your program. When such a barrier is present, the compiler will generate the instructions that will make sure all the loads-stores to the global/local memory space prior to the barrier is done for all the executing work items before executing any instructions that come after the barrier, which will guarantee that the updated data is seen; or in compiler lingo: memory loads and stores will be committed to the memory before any loads and stores follows the barrier/fence.

These APIs provide the developer with a much better level of control when it comes to ordering both reads and writes, reads only, or writes only. The argument flags, can take a combination of `CLK_LOCAL_MEM_FENCE` and/or `CLK_GLOBAL_MEM_FENCE`.

Getting ready

The recipe will show you the code snippet in `Ch4/par_min/par_min.cl` for finding the minimum value in a large array in the device, that is, GPU or CPU memory space. This example combines a few concepts such as using the OpenCL's atomic directives to enable atomic functions and memory barriers to coordinate memory loads and stores.

How to do it...

The following code demonstrates how you might want to find the minimum number in a large container of integers:

```
#pragma OPENCL EXTENSION cl_khr_local_int32_extended_atomics :
  enable
#pragma OPENCL EXTENSION cl_khr_global_int32_extended_atomics :
  enable
__kernel void par_min(__global uint4* src,
  __global uint * globalMin, __local  uint * localMin,
    int numOfItems) {
    uint count = ( numOfItems / 4) / get_global_size(0);
    uint index = get_global_id(0) * count;
    uint stride = 1;
    uint partialMin = (uint) -1;
    for(int i = 0; i < count; ++i,index += stride) {
      partialMin = min(partialMin, src[index].x);
      partialMin = min(partialMin, src[index].y);
      partialMin = min(partialMin, src[index].z);
      partialMin = min(partialMin, src[index].w);
    }
    if(get_local_id(0) == 0) localMin[0] = (uint) -1;
      barrier(CLK_LOCAL_MEM_FENCE);
    atomic_min(localMin, partialMin);
    barrier(CLK_LOCAL_MEM_FENCE);
```


www.EngineeringBooksPdf.com

```
        if (get_local_id(0) == 0)
          globalMin[ get_group_id[0] ] = localMin[0];
    }
    __kernel void reduce(__global uint4* src,
      __global uint * globalMin) {
        atom_min(globalMin, globalMin[get_global_id(0)]);
    }
```

To compile it on the OS X platform, you will have to run a compile command similar to this:

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

Alternatively, you can type `make` in the source directory `Ch4/par_min/`. When that happens, you will have a binary executable named `ParallelMin`.

To run the program on OS X, simply execute the program `ParallelMin` and you should either see the output: `Check passed!` or `Check failed!` as follows:

```
Check passed!
```

How it works...

The way this works is that a work item walks through the source buffer and attempts to locate the minimum value in parallel, and when the kernel is running on the CPU or GPU, the source buffer is chopped evenly between those threads and each thread would walk through the buffer that's assigned to them in `__global memory` and reduces all values into a minimum value in the `__private` memory.

Subsequently, all threads will reduce the minimum values in their `__private` memories to `__local` memory via an atomic operation and this reduced value is flushed to the `__global` memory.

Once the work groups have completed the execution, the second kernel, that is, `reduce` will reduce all the work group values into a single value in the `__global` memory using an atomic operation.

Using floating-point functions

So far, you have seen a couple of functions that takes argument as input or output single-precision or double-precision floating-point values. Given a floating-point value *x*, the OpenCL floating-point functions provide you with the capability to extract the mantissa and exponent from *x* via `frexp()`, decompose *x* via `modf()`, compute the next largest/smallest single-precision floating-point value via `nextafter()`, and others. Considering that there are so many useful floating-point functions, there are two functions which are important to understand because it's very common in OpenCL code. They are the `mad()` and `fma()` functions which is Multiply-Add and Fused Multiply-Add instruction respectively.

www.EngineeringBooksPdf.com

The **Multiply-Add** (**MAD**) instruction performs a floating-point multiplication followed by a floating-point addition, but whether the product and its intermediary products are rounded is undefined. The **Fused Multiply-Add** (**FMA**) instruction only rounds the product and none of its intermediary products. The implementations typically trade off the precision against the speed of the operations.

We probably shouldn't dive into academic studies of this nature; however in times like this, we thought it might be helpful to point out how academia in many situations can help us to make an informed decision. Having said that, a particular study by Delft University of Technology entitled *A Comprehensive Performance Comparison of CUDA* and OpenCL link `http://www.pds.ewi.tudelft.nl/pubs/papers/icpp2011a.pdf`, suggests that FMA has a higher instruction count as compared to MAD implementations, which might lead us to the conclusion that MAD should run faster than FMA. We can guess approximately how much faster by taking a simple ratio between both instruction counts, which we should point out is a really simplistic view since we should not dispense away the fact that compiler vendors play a big role with their optimizing compilers, and to highlight that NVIDIA conducted a study entitled *Precision & Performance: Floating Point and IEEE 754 compliance for NVIDIA GPUs*, which can be read at: `http://developer.download.nvidia.com/assets/cuda/files/NVIDIA-CUDA-Floating-Point.pdf`. The study suggests that FMA can offer performance in addition to precision, and NVIDIA is at least one company that we are aware of who is replacing MAD with FMA in their GPU chips.

Following the subject of multiplication, you should be aware that there are instructions for the multiplication of integers instead of floats; examples of those are `mad_hi`, `mad_sat`, and `mad24`, and these functions provide the developer with the fine grain control of effecting a more efficient computation and how it can be realized using these optimized versions. For example, `mad24` only operates on the lower 24-bits of a 32-bit integer because the expected value is in the range of [-223, 223 -1] when operating signed integers or [0, 224 -1] for unsigned integers.

Getting ready

The code snippet in `Ch4/simple_fma_vs_mad/fma_mad_cmp.cl` demonstrates how we can test the performance between the MAD and FMA instructions, if you so wish, to accomplish the computation. However, what we are going to demonstrate is to simply run each one of the kernels in turn, and we can check that the results are the same in both computations.

How to do it...

The following code demonstrates how to use the MAD and FMA functions in OpenCL:

```
__kernel void mad_test(__global float* a, __global float* b,
  __global float* c, __global float* result) {
```


www.EngineeringBooksPdf.com

```
    float temp = mad(a, b, c);
    result[get_global_id(0)] = temp;
  }
  __kernel void fma_test(__global float* a, __global float* b,
    __global float* c, __global float* result) {
    float temp = fma(a, b, c);
    result[get_global_id(0)] = temp;
  }
```

To compile it on the OS X platform, you will have to run a compile command similar to this:

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

Alternatively, you can type `make` in the source directory `Ch4/simple_fma_vs_mad/`. When that happens, you will have a binary executable named `FmaMadCmp`.

To run the program on OS X, simply execute the program `FmaMadCmp` and you should either see the output: `Check passed!` or `Check failed!` as follows:

```
Check passed!
```

How it works...

The driver code uses single-precision floating-point values to compute the value of the equation by running the two kernels in turn on the GPU/CPU. Each kernel would load the values from the `__global` memory space to the work item/thread's `__private` memory space. The difference between both kernels is that one uses the FMA instruction while the other uses the MAD instruction. The method that is used to detect whether FMA instruction support is available on the device of choice is to detect whether `CP_FP_FMA` is returned after a call to `clGetDeviceInfo` passing in any of the following parameters: `CL_DEVICE_SINGLE_FP_CONFIG`, `CL_DEVICE_DOUBLE_FP_CONFIG`, and `CL_DEVICE_HALF_FP_CONFIG`. We use the flag `CP_FP_FMA` and `FP_FAST_FMA` to load the `fma` functions on our platform by including the header file `#include <math.h>`.

The C-macro `FP_FAST_FMA`, if defined is set to the constant of 1 to indicate that the `fma()` generally executes about as fast, or faster than, a multiple and an addition of double operands. If this macro is undefined, then it implies that your hardware doesn't support it.In the GNU GCC compiler suite, the macro you want to detect is `__FP_FAST_FMA`, which links to the `FP_FAST_FMA` if defined or passing `-mfused-madd` to the GCC compiler (on by default, autogenerate the FMA instructions if ISA supports).

www.EngineeringBooksPdf.com

Using trigonometric functions

The trigonometric functions are very useful if you were in the computer graphics industry , or you are writing a simulation program for weather forecasts, continued fractions, and so on. OpenCL provides the usual suspects when it comes to the trigonometry support with `cos`, `acos`, `sin`, `asin`, `tan`, `atan`, `atanh` (hyperbolic arc tangent), `sinh` (hyperbolic sine), and so on.

In this section, we will take a look at the popular trigonometric identity function:

sin2 + cos2 = 1

From the Pythagoras's theorem, we understood that a right-angled triangle with sides *a*,*b*,*c* and angle *t* at the vertex where *a* and *c* meet, *cos(t)* is by definition *a*/*c*, *sin(t)* is by definition *b*/*c*, and so *cos2(t) + sin2(t) = (a/c)2 + (b/c)2* when combined with the fact that *a2 + b2 = c2* hence *cos2(t) + sin2(t) = 1*.

Having armed ourselves with this knowledge, there are many interesting problems you can solve with this identity but for the sake of illustration let's suppose that we want to find the number of unit circles.

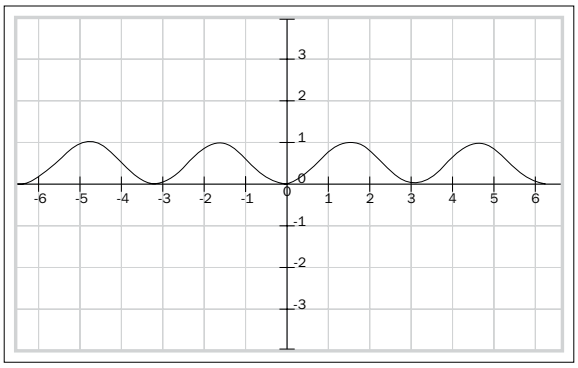
Unit circles are another way of looking at the identity we just talked about. A contrived example of this would be to determine which values would be valid unit circles from the given two arrays of supposedly values in degrees.

Getting ready

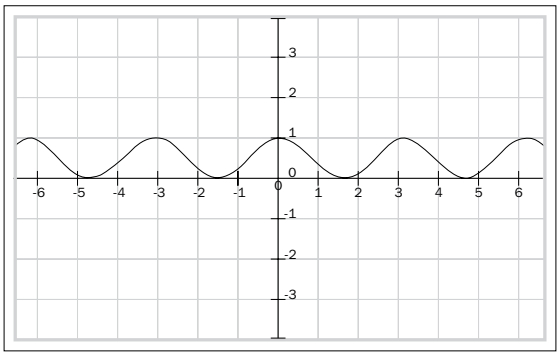
The code snippet in `Ch4/simple_trigonometry/simple_trigo.cl` demonstrates the OpenCL kernel that is used to compute which values from the two data sources can correctly form a unit circle.

If you recall from basic trigonometry lessons you took, when you add the result of *sin(x) + cos(x)* where *x* is drawn from either positive or negative numbers, it will produce two distinct straight line functions *y = 1* and *y = -1* and when you square the results of *sin(x)* and *cos(x)*, the result of *cos2(t) + sin2(t) = 1* is obvious. See the following diagrams for illustration:

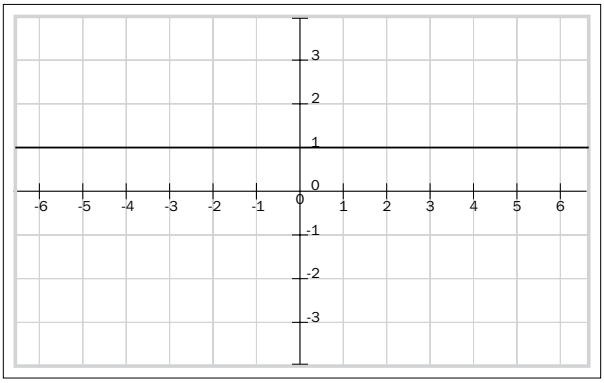
www.EngineeringBooksPdf.com



The preceding diagram and the following diagram reflect the graphs of *sin(x)* and *cos(x)* respectively:



The following diagram illustrates how superimposing the previous two graphs would give a straight line that is represented by the equation:



www.EngineeringBooksPdf.com

How to do it...

The following code snippet shows you the kernel code that will determine unit circles:

```
__kernel void find_unit_circles(__global float16* a,
  __global float16* b, __global float16* result) {
    uint id = get_global_id(0);
    float16 x = a[id];
    float16 y = b[id];
    float16 tresult = sin(x) * sin(x) + cos(y) * cos(y);
    result[id] = tresult;
}
```

To compile it on the OS X platform, you will have to run a compile command similar to this:

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

Alternatively, you can type `make` in the source directory `Ch4/simple_trigonometry/`. When that happens, you will have a binary executable named `SimpleTrigo`.

To run the program on OS X, simply execute the program `SimpleTrigo` and you should either see the output shown as follows:

```
Find Unit Circle:
Unit circle with x=1, y=1
```

How it works...

The driver program conducts its usual operations of loading the two data sources by filling it up with values. Then the data sources is registered on the device command queue along with the kernel program objects that are ready for execution.

During the execution of the kernel, the data sources are loaded into the device via a single-precision floating-point 16-element vector. As highlighted in previous chapters, this takes advantage of the device's vectorized hardware. The in-memory vectors are passed into the sine and cosine functions which comes in two versions where one takes a scalar value and second takes a vector value, and we flush the result out to global memory once we are done; and you will notice that the multiplication/addition operator actually does component-wise multiplication and addition.

www.EngineeringBooksPdf.com

Arithmetic and rounding in OpenCL

Rounding is an important topic in OpenCL and we have not really dived into it yet but that's about to change. OpenCL 1.1 supports four rounding modes: round to nearest (even number), round to zero, round to positive infinity, and round to negative infinity. The only round mode required by OpenCL 1.1 compliant devices is the round to nearest even.

If the result is intermediate between two representable values, the even representation is chosen. Even, here, means that the lowest bit is zero.

You should be aware that these are applicable to single-precision floating-point values supported in OpenCL 1.1; we have to check with the vendors who provide functions that operate on double-precision floating-point values, though the author suspects that they should comply at least to support the round to nearest even mode.

Another point is that, you cannot programmatically configure your kernels to inherit/change the rounding mode used by your calling environment, which most likely is where your program executes on the CPU. In GCC at least, you can actually use the inline assembly directives, for example, `asm("assembly code inside quotes")` to change the rounding mode in your program by inserting appropriate hardware instructions to your program. The next section attempts to demonstrate how this can be done by using the regular C programming with a little help from GCC.

In the Intel 64 and IA-32 architectures, the rounding mode is controlled by a 2-bit **rounding control** (**RC**) field, and the implementation is hidden in two hardware registers: **x87 FPU** control register and **MXCSR** register. These two registers have the RC field and the RC in the x87 FPU control register is used by the CPU when computations are performed in the x87 FPU, while the RC field in the MXCSR is used to control rounding for **SIMD** floating-point computations performed with the **SSE/SSE2** instructions.

Getting ready

In the code snippet found in `Ch4/simple_rounding/simple_rounding.cl`, we demonstrate how *round to nearest even* mode is the default mode in the built-in functions provided by OpenCL 1.1. The example proceeds to demonstrate how a particular built-in function and remainder, will use the default rounding mode to store the result of a floating-point computation. The next couple of operations is to demonstrate the usage of the following OpenCL built-in functions such as `rint`, `round`, `ceil`, `floor`, and `trunc`.

www.EngineeringBooksPdf.com

How to do it...

The following code snippet examines the various rounding modes:

```
__kernel void rounding_demo(__global float *mod_input,
  __global float *mod_output, __global float4 *round_input,
    __global float4 *round_output) {
    mod_output[1] = remainder(mod_input[0], mod_input[1]);
    round_output[0] = rint(*round_input);
    round_output[1] = round(*round_input);
    round_output[2] = ceil(*round_input);
    round_output[3] = floor(*round_input);
    round_output[4] = trunc(*round_input);
}
```

To compile it on the OS X platform, you will have to run a compile command similar to this:

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

Alternatively, you can type `make` in the source directory `Ch4/simple_rounding/`. When that happens, you will have a binary executable named `SimpleRounding`.

To run the program on OS X, simply execute the program `SimpleRounding` and you should either see the output shown as follows:

```
Input: -4.5f, -1.5f, 1.5f, 4.5f
Rint:
Round:
Ceil:
Floor:
Trunc:
```

How it works...

As before, the in-memory data structures on the host are initialized with values and they are issued to the device once the device's command queue is created; once that's done the kernel is sent off to the command queue for execution. The results is subsequently read back from the device and displayed on the console.

In order to understand how these functions work, is important that we study their behavior by first probing their method signatures, and subsequently analyzing the results of executing the program to gain insights into how the results came to be.

www.EngineeringBooksPdf.com

There's more...

OpenCL 1.2 brings a wealth of mathematical functions to arm the developer and four of the common ones are computing the floor and ceiling, round-to-integral, truncation, and rounding floating-point values. The floor's method signature is:

```
gentype floor(gentype x);
// gentype can be float,float2,float3,float4,float8,float16
```

This function rounds to the integral value using the *round to negative infinity* rounding mode. First of all, your OpenCL device needs to support this mode of rounding, and you can determine this by checking the existence of the value `CL_FP_ROUND_TO_INF` when you pass in `CL_DEVICE_DOUBLE_FP_CONFIG` to `clGetDeviceInfo(device_id, ...)`.

The next method, ceil's signature is:

```
gentype ceil(gentype x);
// gentype can be float,float2,float3,float4,float8,float16
```

This function rounds to the integral value using the *round to positive infinity* rounding mode.

Be aware that when a value between `-1` and `0` is passed to `ceil`, then the result is automatically `-0`.

The method for rounding to the integral value has a signature like this:

```
gentype rint(gentype x);
// gentype can be float,float2,float3,float4,float8,float16
```

This function rounds to the integral value using the *round to nearest even* rounding mode.

Be aware that when a value between `-0.5` and `0` is passed to `rint`, then the result is automatically `-0`.

The truncation function is very useful when precision is not high on your priority list and its method signature is:

```
gentype trunc(gentype x);
// gentype can be float,float2,float3,float4,float8,float16
```

This function rounds to the integral value using the *round to zero* rounding mode.

The rounding method signature is:

```
gentype round(gentype x);
// gentype can be float,float2,float3,float4,float8,float16
```

This function returns the integral value nearest to *x* rounding halfway cases away from zero, regardless of the current rounding direction. The full list of available functions can be found in the *Section 6.12.2* of the OpenCL 1.2 specification.

www.EngineeringBooksPdf.com

When you run the program, you should get the following result:

```
Input: -4.5, 1.5, 1.5, 4.5
Rint:  -4.0, -2.0, 2.0, 4.0
Round: -5.0, -2.0, 2.0, 5.0
Ceil:  -4.0, -1.0, 2.0, 5.0
Floor: -5.0, -2.0, 1.0, 4.0
Trunc: -4.0, -1.0, 1.0, 4.0
```

Using the shuffle function in OpenCL

The `shuffle` and `shuffle2` functions were introduced in OpenCL 1.1 to construct a permutation of elements from their inputs (which are either one vector or two vectors), and returns a vector of the same type as its input; the number of elements in the returned vector is determined by the argument, `mask`, that is passed to it. Let's take a look at its method signature:

```
gentypeN shuffle(gentypeM x, ugentypeN mask);
gentypeN shuffle(gentypeM x, gentypeM y, ugentypeN mask);
```

The `N` and `M` used in the signatures represents the length of the returned and input vectors and can take values from {`2`,`3`,`4`,`8`,`16`}. The `ugentype` represents an unsigned type, `gentype` represents the integral types in OpenCL, and floating-point types (that is, half, single, or double-precision) too; and if you choose to use the floating-point types then recall the extensions `cl_khr_fp16` or `cl_khr_fp64`.

Here's an example of how it works:

```
uint4 mask = {0,2,4,6};
uint4 elements = {0,1,2,3,4,5,6};
uint4 result = shuffle(elements, mask);
// result = {0,2,4,6};
```

Let's take a look at a simple implementation where we draw our inspiration from the popular **Fisher-Yates Shuffle**(**FYS**) algorithm. This FYS algorithm generates a random permutation of a finite set and the basic process is similar to randomly picking a numbered ticket from a container, or cards from a deck, one after another until none is left in the container/deck. One of the nicest properties of this algorithm is that it is guaranteed to produce an unbiased result. Our example would focus on how shuffling would work, since what it essentially does is to select a particular element based on a mask that's supposed to be randomly generated.

www.EngineeringBooksPdf.com

Getting ready

The code snippet in `Ch4/simple_shuffle/simple_shuffle.cl` pretty much captured most of the ideas we are trying to illustrate. The idea is simple, we want to generate a mask and use the mask to generate permutations of the output array. We are not going to use a pseudo random number generator like the Mersenne twister, but rather rely on C's `stdlib.h` function, a random function with a valid seed from which we generate a bunch of random numbers where each number cannot exceed the maximum size of the array of the output array, that is, `15`.

The `rand()` function in `stdlib.h` is not really favored because it generates a less random sequence than `random()`, because the lower dozen bits generated by `rand()` go through a cyclic pattern.

How to do it...

Before we begin the shuffling, we need to seed the RNG prior, and we can do that via a simple API call to `srandom()` passing the seed. The next step is to run our kernel a number of times and we achieve this by enclosing the kernel execution in a loop. The following code snippet from the host code in `Ch4/simple_shuffle/simple_shuffle.c` shows this:

```
#define ITERATIONS 6
#define DATA_SIZE 1024
srandom(41L);
  for(int iter = 0; iter < ITERATIONS; ++iter) {
    for(int i = 0; i < DATA_SIZE; ++i) {
      mask[i] = random() % DATA_SIZE;
      // kernel is invoked
    }// end of inner-for-loop
   }//end of out-for-loop
```

The following kernel code transports the inputs via *a* and *b* and their combined element size is `16`, the mask is being transported on the constant memory space (that is, read-only).

```
__kernel void simple_shuffle(__global float8* a,
  __global float8* b, __constant uint16 mask,
    __global float16* result) {
    uint id = get_global_id(0);
    float8 in1 = a[id];
    float8 in2 = b[id];
    result[id] = shuffle2(in1, in2, mask);
}
```


www.EngineeringBooksPdf.com

To compile it on the OS X platform, you will have to run a compile command similar to this:

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

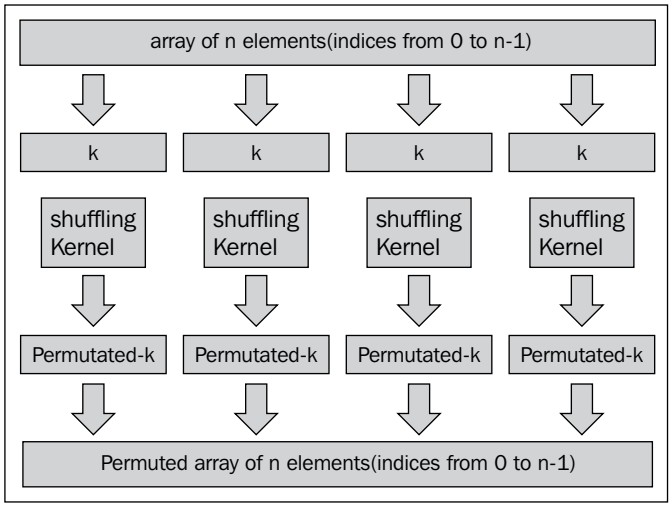
Alternatively, you can type `make` in the source directory `Ch4/simple_shuffle/`. When that happens, you will have a binary executable named `SimpleShuffle`.

To run the program on OS X, simply execute the program `SimpleShuffle` and you should see the output shown as follows:

```
Shuffle: -4.5f, -1.5f, 1.5f, 4.5f
```

How it works...

The following diagram suggests that each executing kernel works through a portion of the source array, which contains of *k* elements by fetching the data from the `__global` memory space to the `__private` memory space. The next operation is to run the shuffling using a vector of random numbers, which we have pregenerated on the host and for each partitioned data block, the kernel will produce a resultant array; and once that's done the kernel flushes out the data to the `__global` memory space. The following diagram illustrates the idea where the resultant array consists of a permutated array made from its individual constituents which are themselves permutations:



www.EngineeringBooksPdf.com

Using the select function in OpenCL

The `select` function is first of all similar to the `shuffle` and `shuffle2` functions we have seen in the previous section and is also known as the **ternary selection**, and it is a member of the relational functions in OpenCL, which is commonly found in the C++ and Java programming languages; but there is a significant difference and that is the `select` function and its variant `bitselect` works not only with single-precision or double-precision floating types, but also vectors of single-precision or double-precision floating-point values. Here's what it looks like:

```
(predicate_is_true? eval_expr_if_true : eval_expr_if_false)
```

Hence, when the predicate is evaluated to be true the expression on the left-hand side of the colon will be evaluated; otherwise the expression on the right-hand side of the colon is evaluated and in both evaluations, a result is returned.

Using an example in OpenCL, the conditional statement as follows:

```
if (x == 1) r = 0.5;
if (x == 2) r = 1.0;
```

can be rewritten using the `select()` function as:

```
r = select(r, 0.5, isequal(x, 1));
r = select(r, 1.0, isequal(x, 2));
```

And for such a transformation to be correct, the original `if` statement cannot contain any code that calls to I/O.

The main advantage `select/bitselect` offers is that vendors can choose to eradicate branching and branch predication from its implementation, which means that the resultant program is likely to be more efficient. What this means is that these two functions act as a façade so that vendors such as AMD could implement the actual functionality using the ISA of SSE2 `__mm_cmpeq_pd`, and `__mm_cmpneq_pd` ; similarly, Intel could choose from the ISA of Intel AVX such as `__mm_cmp_pd`, `__mm256_cmp_pd`, or from SSE2 to implement the functionality of `select` or `bitselect`.

Getting ready

The following example demonstrates how we can use the function, `select`. The function demonstrates the convenience that it offers since it operates on the abstraction of applying a function to several data values, which happens to be in a vector. The code snippet in `Ch4/simple_select_filter/select_filter.cl` attempts to conduct a selection by picking the elements from each list in turn to establish the result, which in this example happens to be a vector.

www.EngineeringBooksPdf.com

How to do it...

The following snippet demonstrates how to do use the `select` function in OpenCL:

```
__kernel void filter_by_selection(__global float8* a,
  __global float8* b, __global float8* result) {
    uint8 mask = (uint8)(0,-1,0,-1,0,-1,0,-1);
    uint id = get_global_id(0);
    float8 in1 = a[id];
    float8 in2 = b[id];
    result[id] = select(in1, in2, mask);
}
```

To compile it on the OS X platform, you will have to run a compile command similar to this:

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

Alternatively, you can type `make` in the source directory `Ch4/simple_select/`. When that happens, you will have a binary executable named `SelectFilter`.

To run the program on OS X, simply execute the program `SelectFilter` and you should either see the output shown as follows:

```
select: -4.5f, -1.5f, 1.5f, 4.5f
```

How it works...

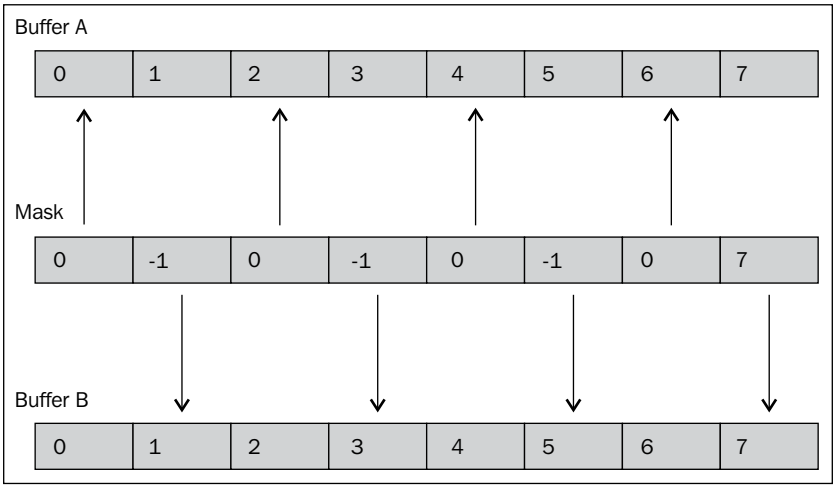
The program proceeds to establish a context to the OpenCL compliant device through the APIs `clGetPlatformIDs` and `clGetDeviceIDs`. Once that is established, we go about creating our in-memory data structures and prepare it for submission to the device's command queue.

The in-memory data structures on the host are small arrays, which we can submit to the device for consumption by sending it across the system bus to hydrate the structures in the device memory. They stay in the device memory as local variables represented by variables `in1` and `in2`.

Once the data is inflated in the device's memory, the algorithm in `select_filter.cl` will proceed to select each element in turn by conducting a bit comparison where the most significant bit is checked; if the MSB is equal to `1` the corresponding value from **Buffer B** is returned; otherwise the corresponding position from **Buffer A** is returned. Recall from computer science that -1, that is, unary minus 1, works out to be `0xffff` in 2's complement notation and hence the MSB of that value would most definitely be equal to `1`.

www.EngineeringBooksPdf.com

The following diagram illustrates this selection process. As before, once the selection process is completed, it is flushed out to the results vector, result.



www.EngineeringBooksPdf.com

www.EngineeringBooksPdf.com

5
Developing a Histogram OpenCL program

In this chapter, we'll cover the following recipes:

- Implementing a histogram in C/C++
- OpenCL implementation of the histogram
- Work-item synchronization

Introduction

Anyone who has taken elementary math in school would know what a histogram is. It's one of the myriad of ways by which one can visualize the relationship between two sets of data. These two sets of data are arranged on two axes such that one axis would represent the distinct values in the dataset and the other axis would represent the frequency at which each value occurred.

The histogram is an interesting topic to study because its practical applications are found in computational image processing, quantitative/qualitative finance, computational fluid dynamics, and so on. It is one of the earliest examples of OpenCL usage when running on CPUs or GPUs, where several implementations have been made and each implementation has its pros and cons.

Implementing a Histogram in C/C++

Before we look at how we can implement this in OpenCL and run the application on the desktop GPU, let's take a look at how we can implement it using a single thread of execution.

www.EngineeringBooksPdf.com

Getting ready

This study of the sequential code is important because we need a way to make sure our sequential code and parallel code produce the same result, which is quite often referred to as the **golden reference** implementation.

In your role as an OpenCL engineer, one of the items on your to-do list would probably be to translate sequential algorithms to parallel algorithms, and it's important for you to be able to understand how to do so. We attempt to impart some of these skills which may not be exhaustive in all sense. One of the foremost important skills to have is the ability to identify **parallelizable routines**.

Examining the code that follows, we can begin to understand how the histogram program works.

How to do it...

Here, we present the sequential code in its entirety, where it uses exactly one executing thread to create the memory structures of a histogram. At this point, you can copy the following code and paste it in a directory of your choice and call this program `Ch5/histogram_cpu/histogram.c`:

```
#define DATA_SIZE 1024
#define BIN_SIZE 256

int main(int argc, char** argv) {
    unsigned int* data = (unsigned int*) malloc( DATA_SIZE *
                         sizeof(unsigned int));
    unsigned int* bin  = (unsigned int*) malloc( BIN_SIZE *
                         sizeof(unsigned int));
    memset(data, 0x0, DATA_SIZE * sizeof(unsigned int));
    memset(bin, 0x0, BIN_SIZE * sizeof(unsigned int));

    for( int i = 0; i < DATA_SIZE; i++) {
        int indx = rand() % BIN_SIZE;
        data[i] = indx;
    }

    for( int i = 0; i < DATA_SIZE; ++i) {
       bin[data[i]]++;
    }

}
```


www.EngineeringBooksPdf.com

To build the program, we are assuming that you have a GNU GCC compiler. Type the following command to into a terminal:

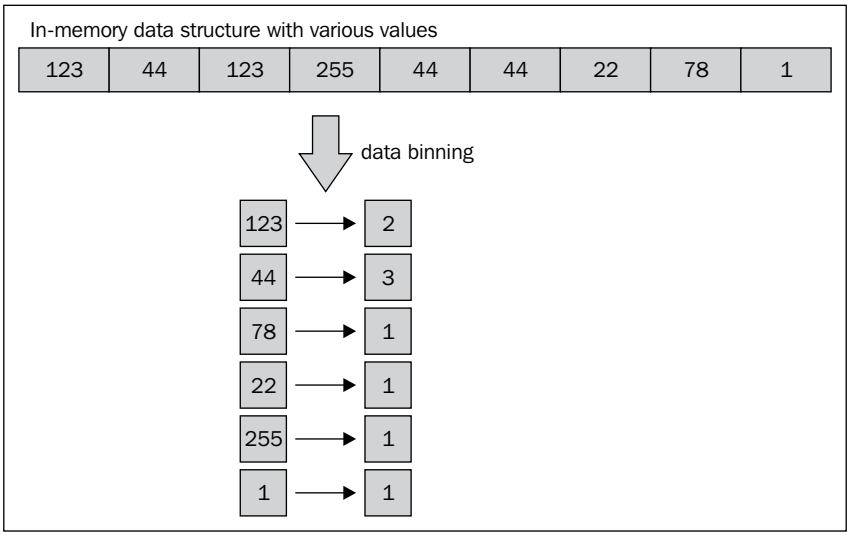
```
/usr/bin/gcc -o histogram Ch5/histogram_c/histogram.c
```

Alternatively, run `make` at the directory `Ch5/histogram_c`, and an executable named `histogram` will be deposited in your directory where you issued that command.

To run the program, simply execute the program `histogram` deposited on the folder `Ch5/histogram_c`, and it should output nothing. However, feel free to inject C's output function `printf`, `sprintf` into the previous code and convince yourself that the histogram is working as it should.

How it works...

To make a histogram, we need to have an initial dataset where it contains values. The values in a histogram are computed by scanning through the dataset and recording how many times a scanned value has appeared in the dataset. Hence, the concept of **data binning**. The following diagram illustrates this concept:



In the following code, we see that the first `for` loop fills up the array `data` with values ranging from `0` to `255`:

```
for( int i = 0; i < DATA_SIZE; i++) {
        int indx = rand() % BIN_SIZE;
        data[i] = indx;
}
```


www.EngineeringBooksPdf.com

The second `for` loop walks the `data` array and records the occurrence of each value, and the final `for` loop serves to print out the occurrence of each value. That is the essence of data binning.

```
for( int i = 0; i < DATA_SIZE; ++i) {
       bin[data[i]]++;
}
```

Finally, you would iterate the binned data and print out what you've found:

```
for( int i = 0; i < BIN_SIZE; i ++) {
        if (bin[i] == 0) continue;
        else printf("bin[%d] = %d\n", i, bin[i]);
}
```

Next, we are going to look at how OpenCL can apply data binning into its implementation.

OpenCL implementation of the Histogram

In this section, we will attempt to develop your intuition to be able to identify possible areas of parallelization and how you can use those techniques to parallelize sequential algorithms.

Not wanting to delve into too much theory about parallelization, one of the key insights about whether a routine/algorithm can be parallelized is to examine whether the algorithm allows work to be split among different processing elements. Processing elements from the OpenCL's perspective would be the processors, that is, CPU/GPU.

Recall that OpenCL's work items are execution elements that act on a set of data and execute on the processing element. They are often found in a work group where all work items can coordinate data reads/writes to a certain degree and they share the same kernel and work-group barriers.

Examining the code, you will notice that the first thing that is probably able to fulfill the description:

> *"...allows work to be split among different processing elements"*

This would be to look for `for` loops. This is because loops mean that the code is executing the same block of instructions to achieve some outcome, and if we play our cards right, we should be able to split the work in the loop and assign several threads to execute a portion of the code along with the data.

www.EngineeringBooksPdf.com

Getting ready

In many algorithms, you will see that splitting the work sometimes does not necessarily imply that the data needs to be cleanly partitioned, and that's because the data is read-only; however, when the algorithm needs to conduct both reads and writes to the data, then you need to figure out a way to partition them cleanly. That last sentence deserves some explanation. Recall in *Chapter 2*, *Understanding OpenCL Data Transfer and Partitioning*, where we discussed work items and data partitioning, and by now you should have understood that OpenCL does not prevent you, the developer, from creating race conditions for your data if you miscalculated the data indexing or even introduced data dependencies.

With great power, comes great responsibility.

In building a data parallel algorithm, it's important to be able to understand a couple of things, and from the perspective of implementing an OpenCL histogram program, here are some suggestions:

- **Understand your data structure**: In the previous chapters, we have seen how we can allow user-defined structures and regular 1D or 2D arrays to be fed into the kernel for execution. You should always search for an appropriate structure to use and make sure you watch for the off-by-one errors (in my experience, they are more common than anything else).
- **Decide how many work items should execute in a work-group**: If the kernel only has one work item executing a large dataset, it's often not efficient to do so because of the way the hardware works. It makes sense to configure a sizeable number of work items to execute in the kernel so that they take advantage of the hardware's resources and this often increases the temporal and spatial locality of data, which means your algorithm runs faster.
- **Decide how to write the eventual result**: In the histogram implementation we've chosen, this is important because each kernel will process a portion of the data and we need to merge them back. We have not seen examples of that before, so here's our chance!

Let's see how those suggestions could apply. The basic idea is to split a large array among several work groups. Each work group will process its own data (with proper indexing) and store/bin that data in the scratchpad memory provided by the hardware, and when the work group has finished its processing, its local memory will be stored back to the global memory.

We have chosen the 1D array to contain the initial set of data and this data can potentially be infinite, but the author's machine configuration doesn't have limitless memory, so there's a real limit. Next, we will split this 1D array into several chunks, and this is where it gets interesting.

www.EngineeringBooksPdf.com

Each chunk of data will be cleanly partitioned and executed by a work group. This work group has chosen to house 128 work items and each work item will produce a bin of size 256 elements or a 256 bin.

Each work group will store these into the local memory also known as s**cratchpad memory** because we don't want to keep going back and forth global and device memory. This is a real performance hit.

In the code presented in the following section, one of the techniques you will learn is to use the scratchpad memory or local memory in aiding your algorithm to execute faster.

Local memory is a software controlled scratchpad memory, and hence its name. The scratchpad allows the kernel to explicitly load items into that memory space, and they exist in local memory until the kernel replaces them, or until the work group ends its execution. To declare a block of local memory, the `__local` keyword is used and you can declare them in the parameters to the kernel call or in the body of the kernel. This memory allocation is shared by all work items in the work group.

The host code cannot read from or write to local memory. Only the kernel can access local memory.

So far you have seen how to obtain memory allocation from the OpenCL device and fire the kernel to consume the input data and reading from that processed data subsequently for verification. What you are going to experience in the following paragraphs might hurt your head a little, but have faith in yourself, and I'm sure we can get this through.

How to do it...

The complete working kernel is presented as follows from `Ch5/histogram/histogram.cl`, and we have littered comments in the code so as to aid you in understanding the motivation behind the constructs:

```
#define MEMORY_BANKS 5U // 32-memory banks.

__kernel

void histogram256(__global const unsigned int4* data,
                                __local uchar* sharedArray,
                                __global uint* binResult) {

// these 4 statements are meant to obtain the ids for the first
// dimension since our data is a 1-d array
size_t localId = get_local_id(0);
size_t globalId = get_global_id(0);
```


www.EngineeringBooksPdf.com

```
size_t groupId = get_group_id(0);
size_t groupSize = get_local_size(0);

int offSet1 = localId & 31;
int offSet2 = 4 * offSet1;
int bankNumber = localId >> MEMORY_BANKS;

__local uchar4* input = (__local uchar4*) sharedArray;

// In a work-group, each work-item would have an id ranging from
// [0..127]
// since our localThreads in 'main.c' is defined as 128
// Each work-item in the work-group would execute the following
// sequence:
// work-item id = 0, input[128 * [0..63]] = 0
// Not forgetting that input is a vector of 4 unsigned char type,
// that effectively means
// that each work-group would execute this loop 8192 times and each
// time it would set
// 4 bytes to zero => 8192 * 4 bytes = 32-KB and this completes the
// initialization of the
// local shared memory array.

for(int i = 0; i < 64; ++i )
  input[groupSize * i + locald] = 0;

// OpenCL uses a relaxed consistency memory model which means to say
// that the state of
// memory visible to a work-item is not guaranteed to be consistent
// across the collection
// of work-items at all times.
// Within a work-item memory has load/store consistency. Local memory
// is consistent
// across work-items in a single work-group at a work-group barrier.
// The statement below
// is to perform exactly that function.
// However, there are no guarantees of memory consistency between
// different
// work-groups executing a kernel

// This statement means that all work-items in a single work-group
// would have to reach
// this point in execution before ANY of them are allowed to continue
// beyond this point.
```


www.EngineeringBooksPdf.com

```
barrier(CLK_LOCAL_MEM_FENCE);

// The group of statements next fetch the global memory data and
// creates a binned
// content in the local memory.
// Next, the global memory is divided into 4 chunks where the
// row_size = 64 and'
// column_size = 128. The access pattern for all work-items in the
// work-group is
// to sweep across this block by accessing all elements in each
// column 64-bytes at a time.
// Once that data is extracted, we need to fill up the 32-KB local
// shared memory so we
// next extract the vector values from the local variable "value" and
// fill them up. The
// pattern we used to store those values is as follows:
// value.s0 can only range from [0..255] and value.s0 * 128 would
// indicate which row
// and column you like to store the value. Now we land in a
// particular row but we need
// to decide which 4-byte chunk its going to store this value since
// value.s0 is a int and
// sharedArray is a uchar-array so we use offSet2 which produces an
// array [0,4,8...124]
// and now we need which chunk its going to land in. At this point,
// you need to remember
// that value.s0 is a value [0..255] or [0x00..0xFF] so we need to
// decide which element in
// this 4-byte sub-array are we going to store the value.
// Finally, we use the value of bankNumber to decide since its range
// is [0..3]
for(int i = 0; i < 64; ++i) {
  uint4 value = data[groupId * groupSize * BIN_SIZE / 4 + i *
groupSize + localId];
  sharedArray[value.s0 * 128 + offSet2 + bankNumber]++;
  sharedArray[value.s1 * 128 + offSet2 + bankNumber]++;
  sharedArray[value.s2 * 128 + offSet2 + bankNumber]++;
  sharedArray[value.s3 * 128 + offSet2 + bankNumber]++;
}

// At this point, you should have figured it out that the 128 * 256
// resembles a hashtable
// where the row indices are the keys of the 256-bin i.e. [0..255]
```


www.EngineeringBooksPdf.com

```
// and the "list" of values
// following each key is what it looks like
// [0]   -> [1,3,5,6 ...]
// [1]   -> [5,6,2,1... ]
// ...
// [255] -> [0,1,5,..]
// Next, we go through this pseudo-hashtable and aggregate the values
// for each key
// and store this result back to the global memory.
// Apply the barrier again to make sure every work-item has completed
// the population of
// values into the local shared memory.

barrier(CLK_LOCAL_MEM_FENCE);

// Now, we merge the histograms
// The merging process is such that it makes a pass over the local
// shared array
// and aggregates the data into 'binCount' where it will make its way
// to the
// global data referenced by 'binResult'

if(localId == 0) { // each work-group only has 1 work-item executing
this code block
  for(int i = 0; i < BIN_SIZE; ++i) {
    uint result = 0;
    for(int j = 0; j < groupSize; ++j) {
      result += sharedArray[i * groupSize + j];
    }
    binResult[groupId * BIN_SIZE  + i] = result;
  }
}
```

To compile it on the OSX platform, you would run a compile command similar to the following:

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

Alternatively, you can run `make` at the directory `Ch5/histogram`, and you would have a binary executable named `Histogram`.

To run the program, simply execute the program, `Histogram`. A sample output on my machine, which is an OS X, is:

```
Passed!
```


www.EngineeringBooksPdf.com

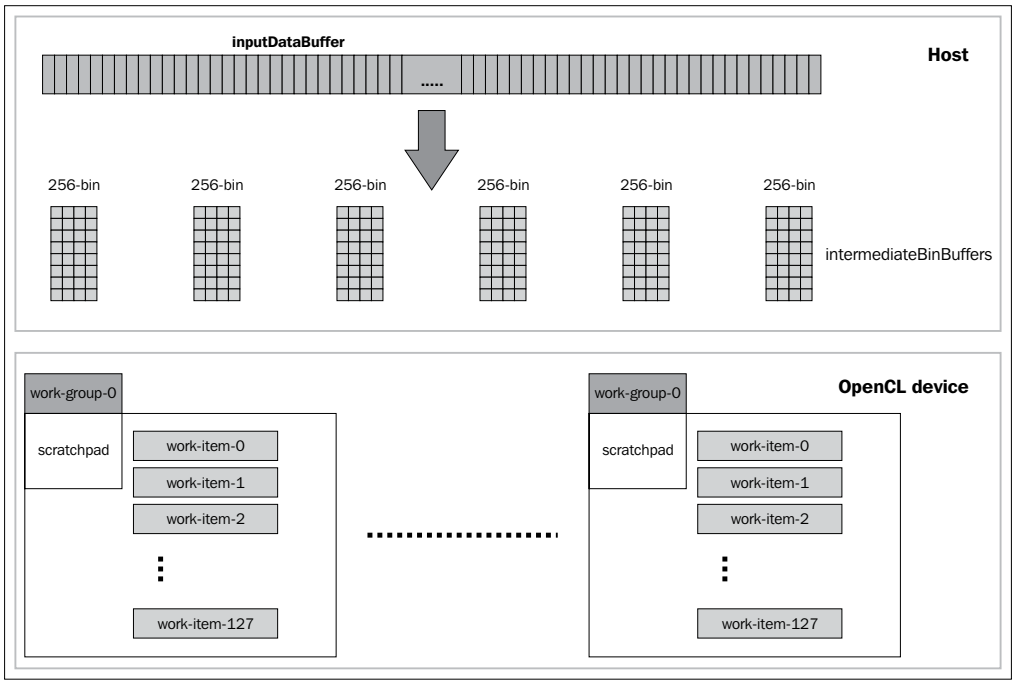
How it works...

In the host code, we first assign the necessary data structures that we need to implement the histogram. An excerpt from the source `Ch5/histogram/main.c` demonstrates the code that creates a single device queue, with the kernel and your usual suspects. The variables `inputBuffer` and `intermediateBinBuffer` refer to the unbinned array and intermediate bins:

```
queue = clCreateCommandQueue(context, device, 0, &error);

cl_kernel kernel = clCreateKernel(program, "histogram256", &error);

inputBuffer = clCreateBuffer(context,
                             CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,
                             width * height * sizeof(cl_uint),
                             data,
                             &error);

intermediateBinBuffer = clCreateBuffer(context,
                                       CL_MEM_WRITE_ONLY,
                                       BIN_SIZE * subHistogramCount *
sizeof(cl_uint),
                                       NULL,
                                       &error);

clSetKernelArg(kernel, 0, sizeof(cl_mem),(void*)& inputBuffer);

// the importance of uchar being that its unsigned char i.e. value //
range[0x00..0xff]
clSetKernelArg(kernel, 1, BIN_SIZE * GROUP_SIZE * sizeof(cl_uchar),
NULL); // bounded by LOCAL MEM SIZE in GPU
clSetKernelArg(kernel, 2, sizeof(cl_mem), (void*)&
intermediateBinBuffer);
```

So conceptually, the code splits the input data into chunks of 256 elements and each such chunk would be loaded into device's local memory, which would be processed by the work items in the work group. The following is an illustration of how it looks like:

www.EngineeringBooksPdf.com



Now, imagine the kernel is going to execute the code and it needs to know how to fetch the data from the global memory, process it, and store it back to some data store. Since we have chosen to use the local memory as a temporary data store, let's take a look at how local memory can be used to help our algorithm, and finally examine how it's processed.

Local memory resembles a lot to any other memory in C, hence you need to initialize it to a proper state before you can use it. After this, you need to make sure that proper array indexing rules are obeyed since those one-off errors can crash your program and might hang your OpenCL device.

The initialization of the local memory is carried out by the following program statements:

```
__local uchar* input = (__local uchar4*) sharedArray;

for(int i = 0; i < 64; ++i)
  input[groupSize * i + localId] = 0;

barrier(CLK_LOCAL_MEM_FENCE);
```

At this point, I should caution you to put on your many-core hat now and imagine that 128 threads are executing this kernel. With this understanding, you will realize that the entire local memory is set to zero by simple arithmetic. The important thing to realize by now, if you haven't, is that each work item should not perform any repeated action.

www.EngineeringBooksPdf.com

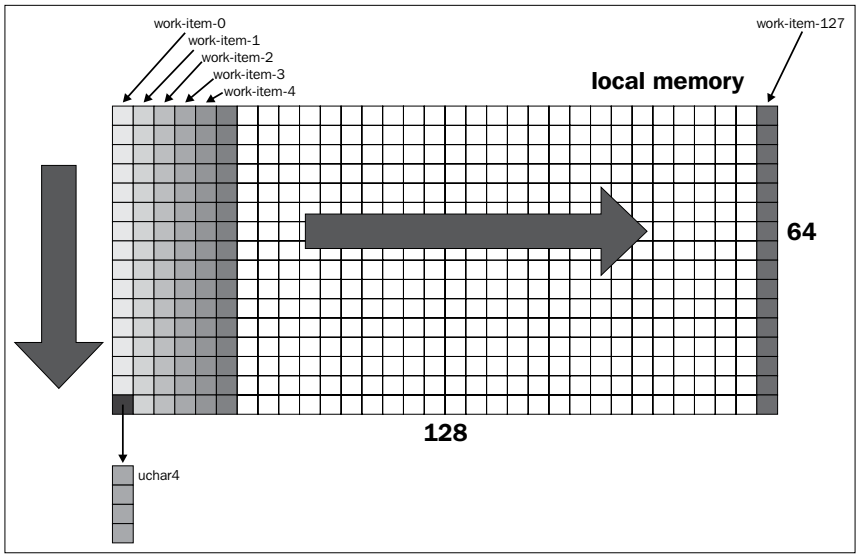
The initialization could have been written in a sequential fashion and it would still work, but it means each work item's initialization would overlap with some other work item's execution. This is, in general, bad since in our case, it would be harmless, but in other cases it means that you could be spending a large amount of time debugging your algorithm. This synchronization applies to all work items in a work group, but doesn't help in synchronizing between work groups.

Next, we see a statement that we probably have not seen before. This is a form of synchronization or memory barrier. The interesting observation about barriers is that all the work items must reach this statement before being allowed to proceed any further. It's like a starting line for runners in a 100 meter race.

Reason for this is that our algorithm's correctness depends on the fact that each element in the local memory must be `0` prior to any work-item wishing to read and write to it.

You should be aware that you cannot set a value for the local memory greater than what is available on the OpenCL device. In order to determine what is the maximum configured scratchpad memory on your device, you need to employ the API `clGetDeviceInfo` passing in the parameter `CL_DEVICE_LOCAL_MEM_SIZE`.

Conceptually, here's what the previous piece of code is doing—each work item sets all elements to zero in a column-wise fashion and sets the elements collectively as a work group with **128** work items executing it, sweeping from left to right. As each item is a `uchar4` data type, you see that the number of rows is **64** instead of **256**:



www.EngineeringBooksPdf.com

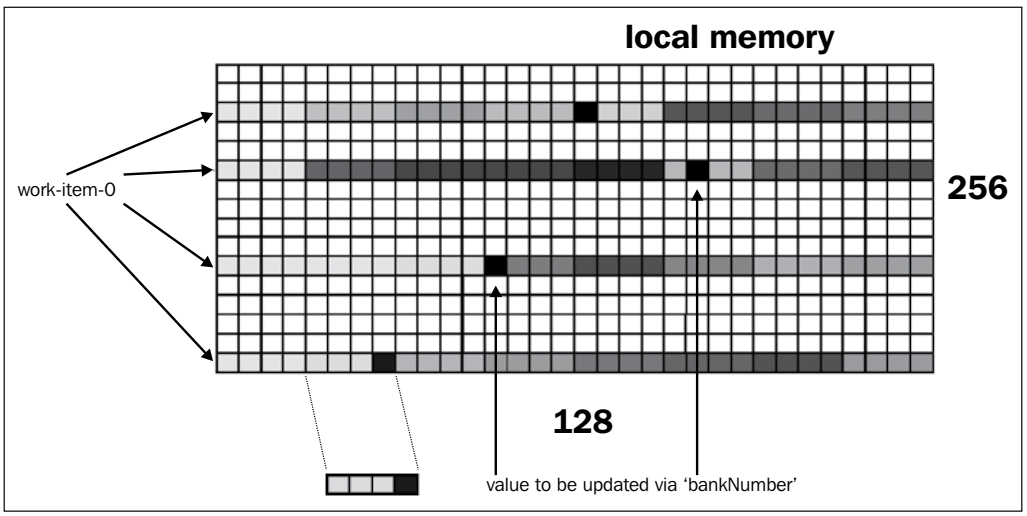
Finally, let's attempt to understand how the values are fetched from global memory and stored in the scratchpad.

When a work group begins executing, it will reach into global memory and fetch the contents of four values and stores them into a local variable and once that's done, the next four statements are executed by each work item to process each retrieved value using the component selection syntax, that is, `value.s0, value.s1, value.s2, value.s3`.

The following illustration, provides how a work item can potentially access four rows of data on the scratchpad and update four elements in those rows by incrementing them. The important point to remember is that all elements in the scratchpad must be written before they can be processed, and hence this is the barrier.

This type of programming technique where we build intermediate data structures so that we can obtain the eventual data structure is often called **thread-based histograms** in some circles. The technique is often employed when we know what the final data structure looks like and we use the same ADT to solve for smaller portions of data so that we can merge them in the end.

```
for(int i = 0; i < 64; i++)
{
       uint4 value =  data[groupId * groupSize * BIN_SIZE/4 + i *
groupSize + localId];
       sharedArray[value.s0 * 128 + offSet2 + bankNumber]++;
       sharedArray[value.s1 * 128 + offSet2 + bankNumber]++;
       sharedArray[value.s2 * 128 + offSet2 + bankNumber]++;
       sharedArray[value.s3 * 128 + offSet2 + bankNumber]++;
}
barrier(CLK_LOCAL_MEM_FENCE);
```



www.EngineeringBooksPdf.com

If you analyze the memory access pattern, you will realize that what we have created an **Abstract Data Type** (**ADT**) known as the **hash table** where each row of data in the local memory represents the list of frequencies of the occurrence of a value between 0 and 255.

With that understanding, we can come to the final part of solving this problem. Again, imagine that the work group has executed to this point, you have basically a hash table, and you want to merge all those other hash tables held in the local memories of the other work groups.

To achieve this, we need to basically walk through the hash table, aggregate all the values for each row, and we would have our answer. However, now we only need one thread to perform all this, otherwise all 128 threads executing the *walk* would mean you're overcounting your values by 128 times! Therefore, to achieve this, we make use of the fact that each work item has a local ID in the work group, and we execute this code by selecting one particular work item only. The following code illustrates this:

```
if(localId == 0) {
    for(int i = 0; i < BIN_SIZE; ++i) {
        uint result = 0;
        for(int j = 0; j < 128; ++j)  {
            result += sharedArray[i * 128 + j];
        }
        binResult[groupId * BIN_SIZE + i] = result;
    }
}
```

There is no particular reason why the first work item is chosen, I guess this is done just by convention, and there's no harm choosing other work items, but the important thing to remember is that there must only be one executing code.

Now we turn our attention back to the host code again, since each intermediate bin has been filled conceptually with its respective value from its respective portions of the large input array.

The (slightly) interesting part of the host code is simply walking through the returned data held in `intermediateBins` and aggregating them to `deviceBin`:

```
for(int i = 0; i < subHistogramCount; ++i)
    for( int j = 0; j < BIN_SIZE; ++j) {
        deviceBin[j] += intermediateBins[i * BIN_SIZE + j];
}
```

And we are done!

www.EngineeringBooksPdf.com

Work item synchronization

This section is to introduce you to the concepts of synchronization in OpenCL. Synchronization in OpenCL can be classified into two groups:

- Command queue barriers
- Memory barriers

Getting ready

The command queue barrier ensures that all previously queued commands to a command queue have finished execution before any following commands queued in the command queue can begin execution.

The work group barrier performs synchronizations between work items in a work group executing the kernel. All work items in a work group must execute the barrier construct before any are allowed to continue execution beyond the barrier.

How to do it...

There are two APIs for the command queue barriers and they are:

```
cl_int clEnqueueBarrierWithWaitList
          (cl_command_queue command_queue,
           cl_uint num_events_in_wait_list,
           const cl_event *event_wait_list,
           cl_event *event)

cl_int clEnqueueMarkerWithWaitList
          (cl_command_queue command_queue,
           cl_uint num_events_in_wait_list,
           const cl_event *event_wait_list,
           cl_event *event)
```

But as of OpenCL 1.2, the following command queue barriers are deprecated:

```
cl_int clEnqueueBarrier(cl_command_queue queue);
cl_int clEnqueueMarker(cl_command_queue queue, cl_event* event);
```

These four/two APIs in OpenCL 1.2/1.1 respectively, allow us to perform synchronization across the various OpenCL commands, but they do not synchronize the work items.

www.EngineeringBooksPdf.com

There is no synchronization facility available to synchronize between work groups.

We have not seen any example codes on how to use this, but it is still good to know they exist, if we ever need them.

Next, you can place barriers to work items in a work group that performs reads and writes to/from local memory or global memory. Previously, you read that all work items executing the kernel must execute this function before any are allowed to continue execution beyond the barrier. This type of barrier must be encountered by all work items in a work group.

How it works...

The OpenCL API is as follows:

```
void barrier(cl_mem_fence flags);
```

where flags can be `CLK_LOCAL_MEM_FENCE` or `CLK_GLOBAL_MEM_FENCE`. Be careful where you place the barrier in the kernel code. If the barrier is needed in a conditional statement that is like an `if-then-else` statement, then you must make sure all execution paths by the work items can reach that point in the program.

The `CLK_LOCAL_MEM_FENCE` barrier will either flush any variables stored in local memory or queue a memory fence to ensure correct ordering of memory operations to local memory.

The `CLK_GLOBAL_MEM_FENCE` barrier function will queue a memory fence to ensure correct ordering of memory operations to global memory.

Another side effect of placing such barriers is that when they're to be placed in loop construct, all work items must execute the barrier for each iteration of the loop before any are allowed to continue execution beyond the barrier. This type of barrier also ensures correct ordering of memory operations to local or global memory.

www.EngineeringBooksPdf.com

6
Developing a Sobel Edge Detection Filter

In this chapter, we'll cover the following recipes:

- Understanding the convolution Theory
- Understanding convolution in 1D
- Understanding convolution in 2D
- OpenCL implementation of the Sobel edge filter
- Understanding profiling in OpenCL

Introduction

In this chapter, we are going to take a look at how to develop a popular image processing algorithm known as edge detection. This problem happens to be a part of solving a more general problem in image segmentation.

Image segmentation is the process of partitioning a digital image into multiple segments (sets of pixels, also known as super pixels). The goal of segmentation is to simplify and/or change the representation of an image into something that is more meaningful and easier to analyze. Image segmentation is typically used to locate objects and boundaries (lines, curves, and so on) in images.

The Sobel operator is a discrete differentiation operator, computing an approximation of the gradient of the image density function. The Sobel operator is based on convolving the image with a small, separable, and an integer-value filter in both horizontal and vertical directions. Thus, it is relatively inexpensive in terms of computations.

www.EngineeringBooksPdf.com

Don't worry if you don't understand these notations right away, we are going to step through enough theory and math, and help you realize the application in OpenCL.

Briefly, the Sobel filtering is a three-step process. Two 3 x 3 filters are applied separately and independently on every pixel and the idea is to use these two filters to approximate the derivatives of x and y, respectively. Using the results of these filters, we can finally approximate the magnitude of the gradient.

The gradient computed by running Sobel's edge detector through each pixel (which also uses its neighboring eight pixels) will inform us whether there are changes in the vertical and horizontal axes (where the neighboring pixels reside).

For those who are already familiar with the convolution theory, in general, may skip to the *How to do it* section of this recipe.

Understanding the convolution theory

In the past, mathematicians developed calculus so that there's a systematic way to reason about how things change, and the convolution theory is really about measuring how these changes affect one another. At that time, the convolution integral was born.

$$f(x) \otimes g(x) = \int_{-\infty}^{\infty} f(x).g(u-x) = \int_{-\infty}^{\infty} g(x).f(u-x)$$

And the $\otimes$ operator is the convolution operator used in conventional math. An astute reader will notice immediately that we have replaced one function with the other, and the reason why this is done is because of the fact that the convolution operator is commutative, that is, the order of computation does not matter. The computation of the integral can be done in discrete form, and without loss of generality, we can replace the integral sign $(\int)$ with the summation sign $(\sum)$, and with that, let's see the mathematical definition of convolution in discrete time domain.

Getting ready

Later we will walk through what the following equation tells us over a discrete time domain:

$$y[n] = x[n] * h[n] = \sum_{k=-\infty}^{\infty} x[k] \bullet h[n-k]$$

where *x[n]* is an input signal, *h[n]* is an impulse response, and *y[n]* is the output.
The asterisk $(*)$ denotes convolution. Notice that we multiply the terms of *x[k]* by the terms of a time-shifted *h[n]* and add them up. The key to understanding convolution lies behind impulse response and impulse decomposition.

www.EngineeringBooksPdf.com

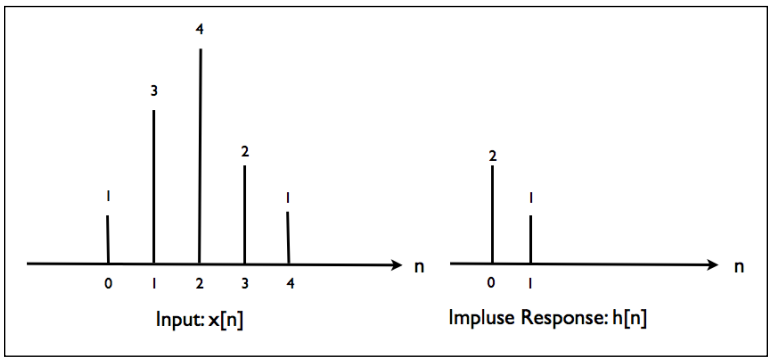
How to do it...

In order to understand the meaning of convolution, we are going to start from the concept of signal decomposition. The input signal can be broken down into additive components, and the system response of the input signal results in by adding the output of these components passed through the system.

The following section will illustrate on how convolution works in 1D, and once you're proficient in that, we will build on that concept and illustrate how in convolution works 2D and we'll see the Sobel edge detector in action!

Understanding convolution in 1D

Let's imagine that a burst of energy (signal) have arrived into our system and it looks similar to the following diagram with *x[n] = {1,3,4,2,1}, for n = 0,1,2,3,4*.



And let's assume that our impulse function has a non-zero value whenever **n** = **0** or **1**, while it'll have a zero value for all other values of **n**.

How to do it...

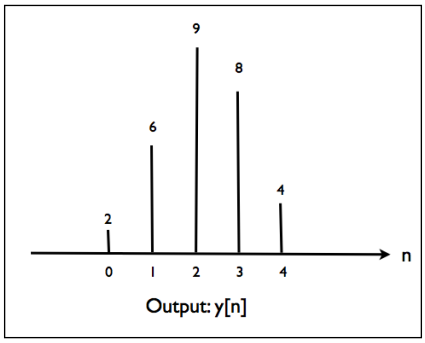
Using the preceding information, let's work out what the output signal would be by quickly recalling the following equation:

$$y[n] = x[n] * h[n] = \sum_{k=-\infty}^{\infty} x[k] \bullet h[n-k]$$

www.EngineeringBooksPdf.com

Following this equation faithfully, we realize that the output signal is amplified initially and quickly tapers off, and after solving this manually (yes, I mean evaluating the equation on a pencil and paper) we would see the following final output signal:

$$y[0] = x[k] * h[0] = \sum_{k=-\infty}^{\infty} x[k] \bullet h[0-k] = x[0] * h[0] = 1 * 2$$
$$y[1] = x[k] * h[1] = \sum_{k=-\infty}^{\infty} x[k] \bullet h[1-k] = x[0] * h[1-0] + x[1] * h[1-1] = x[0] * h[1] + x[1] * h[1-1] = 0 * 1 + 3 * 2 = 6$$
$$y[2] = x[k] * h[2] = \sum_{k=-\infty}^{\infty} x[k] \bullet h[2-k] = x[0] * h[2-0] + x[1] * h[2-1] + x[2] * h[2-2] = 1 * 0 + 3 * 1 + 4 * 2 = 9$$
$$y[3] = x[k] * h[3] = \sum_{k=-\infty}^{\infty} x[k] \bullet h[3-k] = x[0] * h[3-0] + x[1] * h[3-1] + x[2] * h[3-2] + x[3] * h[3-3] = 1 * 0 + 3 * 0 + 4 * 1 + 2 * 2 = 8$$
$$y[4] = x[k] * h[4] = \sum_{k=-\infty}^{\infty} x[k] \bullet h[4-k] = x[0] * h[4-0] + x[1] * h[4-1] + x[2] * h[4-2] + x[3] * h[4-3] + x[4] * h[4-4] = 1 * 0 + 3 * 0 + 4 * 0 + 2 * 1 + 1 * 2 = 4$$



How it works...

Looking at the preceding equation again, this time we rearrange them and remove all terms that evaluate to zero. Let's try to see whether we can discover a pattern:

$$y[0] = x[k] * h[0] = \sum_{k=-\infty}^{\infty} x[k] \bullet h[0-k] = h[0] * x[0]$$
$$y[1] = x[k] * h[1] = \sum_{k=-\infty}^{\infty} x[k] \bullet h[1-k] = h[0] * x[1] + h[1] * x[0]$$
$$y[2] = x[k] * h[2] = \sum_{k=-\infty}^{\infty} x[k] \bullet h[2-k] = h[0] * x[2] + h[1] * x[1]$$
$$y[3] = x[k] * h[3] = \sum_{k=-\infty}^{\infty} x[k] \bullet h[3-k] = h[0] * x[3] + h[1] * x[2]$$
$$y[4] = x[k] * h[4] = \sum_{k=-\infty}^{\infty} x[k] \bullet h[4-k] = h[0] * x[4] + h[1] * x[3]$$

And I believe you can see that each output value is computed from its previous two output values (taking into account the impulse function)! And now we may conclude, quite comfortably, that the general formula for computing the convolution in 1D is in fact the following:

$$y[i] = x[i] * h[0] + x[i-1] * h[1] + x[i-2] * h[2] + \ldots + x[i-(k-1)] * h[k-1]$$

www.EngineeringBooksPdf.com

Finally, you should be aware that (by convention) any value that is not defined for any *x[i-k]* is automatically given the value zero. This seemingly small, subtle fact will play a role in our eventual understanding of the Sobel edge detection filter which we'll describe next.

Finally for this section, let's take a look at how a sequential convolution code in 1D might look like:

```
// dataCount is size of elements in the 1D array
// kernelCount is the pre-defined kernel/filter e.g. h[0]=2,h[1]=1
// h[x]=0 for x ={…,-1,2,3,…}
for(int i = 0; i < dataCount; ++i) {
  y[i] = 0;
  for(int j = 0; j < kernelCount; ++j) {
    y[i] += x[i – j] * h[j]; // statement 1
  }
}
```

Examining the code again, you will probably notice that we are iterating over the 1D array and the most interesting code would be in `statement 1`, as this is where the action really lies. Let's put that new knowledge aside and move on to extending this to a 2D space.

Understanding convolution in 2D

Convolution in 2D is actually an extension of the previously described *Understanding convolution in 1D* section, and we do so by computing the convolution in two dimensions.

Getting ready

The impulse function also exists in a 2D spatial domain, so let's call this function. *b[x,y]* has the value 1, where x and y are zero, and zero where x,y ¹ 0. The impulse function is also referred to as filter or kernel when it's being used in image processing.

How to do it...

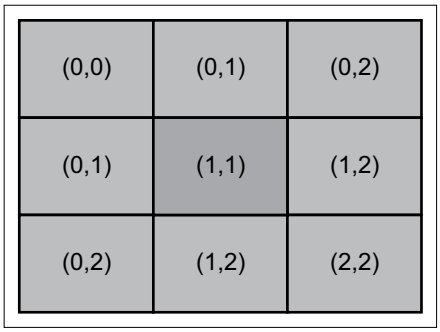
Using the previous example as a guide, let's start thinking from the perspective of a signal which can be decomposed into the sum of its components and impulse functions, and their double summation accounts to the fact that this runs over both vertical and horizontal axes in our 2D space.

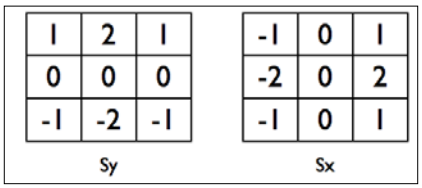
$$y[m,n] = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} x[i,j] \bullet \beta[m-i, n-j]$$

www.EngineeringBooksPdf.com

Next, I think it's very helpful if we use an example to illustrate how it works when we have two convolution kernels to represent the filters we like to apply on the elements in a 2D array. Let's give them names, **Sx** and **Sy**. The next thing is to try out how the equation would develop itself in a 2D setting, where the element we want to convolve is at *x[1,1]* and we make a note of its surrounding eight elements and then see what happens.

If you think about why we are choosing the surrounding eight elements, it's the only way we can measure how big a change is with respect to every other element.





Let's give it a go:

$$y[1,1] = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} x[i,j] \bullet \beta[1-i,1-j]$$

$$y[1,1] = x[0,0]*\beta[1,1] + x[1,0]*\beta[0,1] + x[2,0]*\beta[-1,0] + x[0,1]*\beta[1,0] + x[1,1]*\beta[0,0] + x[2,1]*\beta[-1,0] + x[0,2]*\beta[1,-1] + x[1,2]*\beta[0,1] + x[2,2]*\beta[-1,-1]$$

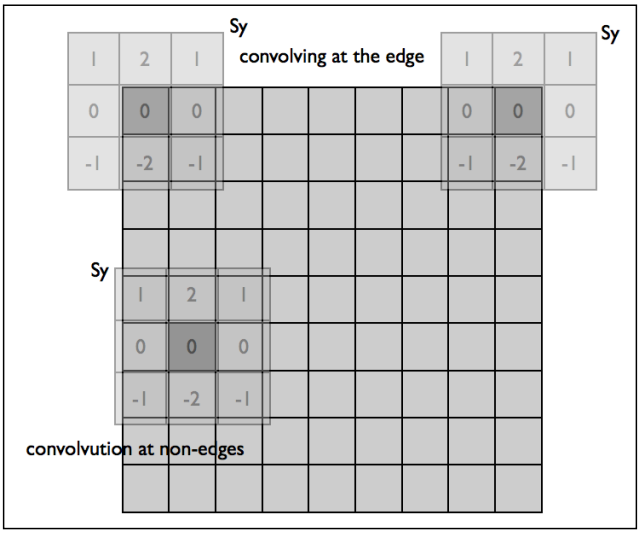
This results in the summation of nine elements (including the element we're interested in), and this process is repeated for all elements in the 2D array. The following diagram illustrates how convolution in 2D works in a 2D space.

You may wish to read Irwin Sobel's 1964 original doctoral thesis since he's the inventor, and this author had a good fortune of meeting the man himself.

www.EngineeringBooksPdf.com

What happens when you attempt to convolve around the elements that border the 2D array or in image processing, are they referred to as edge pixels? If you use this formula for computation, you will notice that the results will be inaccurate, because those elements are undefined and hence they're in general discounted from the final computation. In general, you can imagine a 3 x 3 filtering operation being applied to each element of the 2D array and all such computations will result in a new value for that element in the output data array.

Next, you may wonder what is being done to this output array? Remember that this array now contains values, which basically shows how big is the change detected in a particular element is. And when you obtain a bunch of them in the vicinity, then it usually tells you major color changes, that is, edges.



How it works...

With this understanding, you can probably begin to appreciate why we took this effort to illustrate the theory behind a concept.

When you want to build non-trivial OpenCL applications for your customers, one of the things you have to deal with is learning how to interpret a problem and convert it to a solution. And what that means is mostly about formulating an algorithm (or picking existing algorithms to suit your case) and verifying that it works. Most of the problems you're likely to encounter are going to involve some sort of mathematical understanding and your ability to learn about it. You should treat this as an adventure!

www.EngineeringBooksPdf.com

Now that we've armed ourselves with what convolution is in a 2D space, Let's begin by taking a look at how convolution in 2D would work in regular C/C++ code with the following snippet:

```
// find centre position of kernel (assuming a 2D array of equal
// dimensions)
int centerX = kernelCols/2;
int centerY = kernelRows/2;
for(int i = 0; i < numRows2D; ++i) {
  for(int j = 0; j < numCols2D; ++j) {
    for(m = 0; m < kernelRows; ++m) {
          mm = kernelRows - 1 - m;
    for(n = 0; n < kernelCols; ++n) {
              nn = kernelCols - 1 - n;
        ii = i + (m - centerX);
        jj = j + (n - centerY);
        if (ii >= 0 && ii < rows && jj >= 0 && jj < numCols)
       out[i][j] += in[ii][jj] * kernel[mm][nn]; // statement 1
    }
    }
  }
}
```

This implementation is probably the most direct for the purpose of understanding the concept, although it may not be the fastest (since it's not many-core aware). But it works, as there are conceptually two major loops where the two outer `for` loops are for iterating over the entire 2D array space, while the two inner `for` loops are for iterating the filter/kernel over the element, that is, convoluting and storing the final value into an appropriate output array.

Putting on our parallel algorithm developer hat now, we discover that `statement 1` appears to be a nice target for work items to execute over. Next, let's take a look at how we can take what we've learnt and build the same program in OpenCL.

OpenCL implementation of the Sobel edge filter

Now that you've been armed with how convolution actually works, you should be able to imagine how our algorithm might look like. Briefly, we will read an input image assuming that it's going to be in the Windows BMP format.

Getting ready

Next we'll construct the necessary data structures for transporting this image file in the OpenCL device for convolution, and once that's done we'll read and write the data out to another image file, so that we can compare the two.

www.EngineeringBooksPdf.com

Optionally, you can choose to implement this using the `clCreateImage(...)` APIs provided by OpenCL, and we'll leave it as an exercise for the reader to make the attempt.

In the following sections, you will be shown with an implementation from what is translated, what we have learnt so far. It won't be the most efficient algorithm, and that's really not our intention here. Rather, we want to show you how you can get this done quickly and we'll let you inject those optimizations which include the not withstanding, following data binning, data tiling, shared memory optimization, warp / wavefront-level programming, implementing 2D-convolution using fast fourier transformations, and so many other features.

A possible avenue from where I derived a lot of the latest techniques about solving convolution was by reading academic research papers published by AMD and NVIDIA, and also by visiting `gpgpu.org`, `developer.amd.com`, `developer.nvidia.com`, and `developer.intel.com`. Another good resource I can think of are books on image processing and computer vision from your favorite local bookstores. Also, books on processor and memory structure released by Intel are also good resources if you like.

How to do it...

We only show the code for the kernel found in `Ch6/sobelfilter/sobel_detector.cl`, since this is where our algorithm translation will reach its Xenith. And we've not shown the host code in `Ch6/sobelfilter/SobelFilter.c`, since we believe that you would be confident to know what typically resides in there:

```
__kernel void SobelDetector(__global uchar4* input,
                            __global uchar4* output) {
      uint x = get_global_id(0);
      uint y = get_global_id(1);

  uint width = get_global_size(0);
  uint height = get_global_size(1);

  float4 Gx = (float4)(0);
  float4 Gy = (float4)(0);

    // Given that we know the (x,y) coordinates of the pixel we're
    // looking at, its natural to use (x,y) to look at its
    // neighbouring pixels
    // Convince yourself that the indexing operation below is
    // doing exactly that
```


www.EngineeringBooksPdf.com

```
        // the variables i00 through to i22 seek to identify the pixels
        // following the naming convention in graphics programming e.g.
        // OpenGL where i00 refers
        // to the top-left-hand corner and iterates through to the bottom
        // right-hand corner

      if( x >= 1 && x < (width-1) && y >= 1 && y < height - 1)
      {
        float4 i00 = convert_float4(input[(x - 1) + (y - 1) * width]);
        float4 i10 = convert_float4(input[x + (y - 1) * width]);
        float4 i20 = convert_float4(input[(x + 1) + (y - 1) * width]);
        float4 i01 = convert_float4(input[(x - 1) + y * width]);
        float4 i11 = convert_float4(input[x + y * width]);
        float4 i21 = convert_float4(input[(x + 1) + y * width]);
        float4 i02 = convert_float4(input[(x - 1) + (y + 1) * width]);
        float4 i12 = convert_float4(input[x + (y + 1) * width]);
        float4 i22 = convert_float4(input[(x + 1) + (y + 1) * width]);

            // To understand why the masks are applied this way, look
            // at the mask for Gy and Gx which are respectively equal
            // to the matrices:
            // { {-1, 0, 1}, { {-1,-2,-1},
            //   {-2, 0, 2},   { 0, 0, 0},
            //   {-1, 0, 1}}   { 1, 2, 1}}

    Gx = i00 + (float4)(2) * i10 + i20 - i02  - (float4)(2) * i12 -i22;
    Gy = i00 - i20  + (float4)(2)*i01 - (float4)(2)*i21 + i02  -  i22;

            // The math operation here is applied to each element of
            // the unsigned char vector and the final result is applied
            // back to the output image
      output[x + y *width] = convert_uchar4(hypot(Gx, Gy)/(float4)(2));
      }
    }
```

An astute reader will probably figure out by reading the code, that the derived values for `Gx` and `Gy` should have been as follows:

```
Gx = i00 + (float4)(2) * i10 + i20 - i02  - (float4)(2) * i12 - i22+
0*i01+0*i11+0*i21;
Gy = i00 - i20  + (float4)(2)*i01 - (float4)(2)*i21 + i02 - i22+
0*i10+0*i11+0*i12;
```

But since we know their values will be zero, there is no need for us to include the computation inside it. Although we did, it's really a minor optimization. It shaved off some GPU processing cycles!

www.EngineeringBooksPdf.com

As before, the compilation steps are similar to that in `Ch6/sobelfilter/SobelFilter.c` with the following command:

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

To execute the program, simply execute the executable file (`SobelFilter`) on the `Ch6/sobelfilter` directory, and an output image file named `OutputImage.bmp` would be presented (it's the output of reading in `InputImage.bmp` and conducting the convolution process against it).

The net effect is that the output contains an image that outlines the edges of the original input image, and you can even refer to the picture images in the *How it works...* section of this recipe to see how these two images are different from one another.

How it works...

At first, we create a representation of a pixel to represent each of the channels in the RGBA fashion. That structure is given a simple name, `uchar4`, where it consists of four unsigned char data types which will correctly represent each color's range from [0..255] or [0x00..0xFF], since that's how each color's range is defined by convention.

We omit the description of the mechanism behind pulling the pixel information from the input image to how we construct the final in-memory representation of the image. Interested readers can search on the Internet regarding the Windows BMP format to understand how we parse the image data or read the source code in the `bmp.h` file via the `load` function, and we write out the image using the `write` function.

Skipping the OpenCL device memory allocation, since that by now is standard fare we arrived quickly at the portion where we look at how the kernel processes each pixel of the input data.

Before we do that, let's quickly recall from the kernel launching code how many global work-items have been assigned and whether the work-group composition is like:

```
clEnqueueNDRangeKernel(command, queue, 2, NULL, globalThreads,
localThreads, 0, NULL, NULL);
```

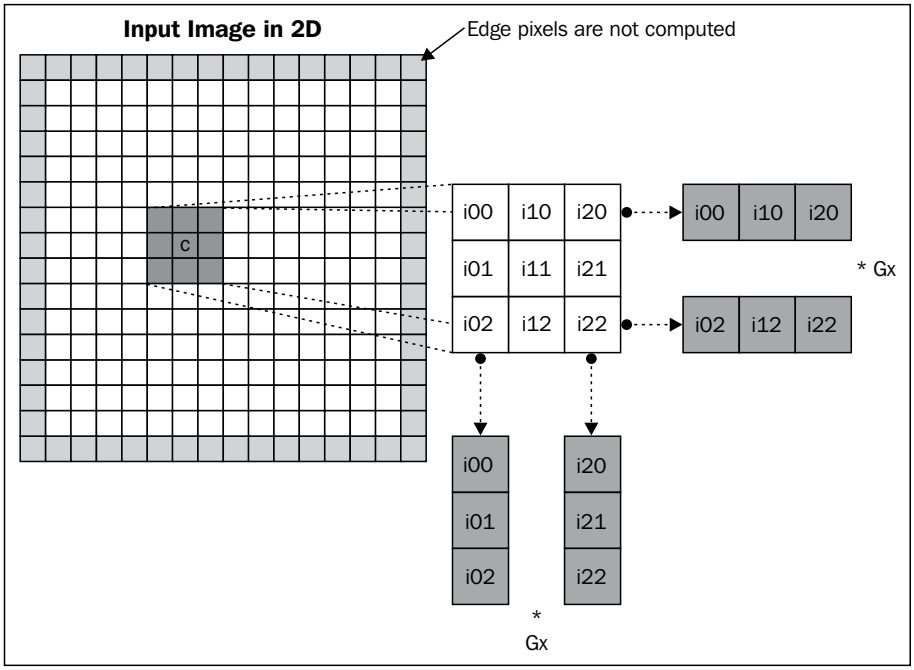
`localThreads` is configured to have work-groups of sizes {256,1}, work-items processing a portion of the input 2D image data array.

When the image is loaded into the device memory, the image is processed in blocks. Each block has a number of work-items or threads if you process the image. Each work-item proceeds the next to perform the convolution process on the center of the pixel and also on its eight neighbors. The resultant value generated by each work-item will be outputed as pixel value into the device memory. Pictorially, the following diagram illustrates what a typical work-item will perform.

www.EngineeringBooksPdf.com

You need to watch out and that is we actually used the data type conversion function, `convert_float4` to apply our unsigned char data values encapsulated within each pixel, which effectively widens the data type so that it doesn't overflow when the Sobel operator is applied on them.

Finally, once we have the masked the values we need to compute the magnitude of this gradient and the standard way of computing that is to apply $\sqrt{Gx^2 + Gy^2}$ where **Gx** = $\begin{matrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{matrix}$ and **Gy** = $\begin{matrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{matrix}$.



Whether this algorithm works, the only way is to check it through an image. The following is the side-by-side comparison, where the first image is before the Sobel operator is applied and the second one is after it's being applied.

www.EngineeringBooksPdf.com

However, there is another nice optimization which we could have done, and it would have helped if we understood that a 3 X 3 convolution kernel (for example, the Sobel operator) is actually equivalent to the product of two vectors. This realization is behind the optimization algorithm also known as separable convolution.

www.EngineeringBooksPdf.com

Technically, a two-dimensional filter is considered to be separable if it can be expressed as an outer product of two vectors. Considering the Sobel operator here, we can actually write

$$\begin{matrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{matrix} = [1 \;\; 2 \;\; 1]^T * [-1 \;\; 0 \;\; 1]$$ and $$\begin{matrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{matrix} = [-1 \;\; 0 \;\; 1]^T * [1 \;\; 2 \;\; 1]$$.

The superscript T is the transpose of a row vector, which is equivalent to the column-vector and vice versa. Note that convolution is itself associative, so it doesn't really matter in which way you multiply the vectors against the input image matrix.

Why is this important? The main reason is because we actually save processing cycles by using this separable convolution kernel. Let's imagine we have a X-by-Y image and a convolution kernel of M-by-N. Using the original method, we would have conducted XYMN multiples and adds while using the separable convolution technique, we would have actually done XY (M + N) multiples and adds. Theoretically speaking, applying this to our 3-by-3 convolution kernel we would have increased our performance to 50 percent or 1.5 times and when we use a 9-by-9 convolution kernel, we would have increased our performance to 81 / 18 = 4.5 or 450 percent.

Next, we are going to talk about how you can profile your algorithms and their runtimes so that you can make your algorithms not only run faster, but also deepen your understanding of how the algorithm works and more often than not, help the developer develop a better intuition on how to make better use of the OpenCL device's capabilities.

Understanding profiling in OpenCL

Profiling is a relatively simple operation from the perspective of an OpenCL developer, since it basically means that he/she wishes to measure how long a particular operation took. This is important because during any software development, users of the system would often specify the latencies which are considered acceptable, and as you develop bigger and more complex systems, profiling the application becomes important in helping you understand the bottlenecks of the application. The profiling we are going to take is a look done programmatically by the developer to explicitly measure the pockets of code. Of course, there is another class of profilers which profiles your OpenCL operations on a deeper level with various breakdowns on the running times measured and displayed, but that is out of the scope of the book. But we encourage readers to download the profilers from AMD and Intel to check them out.

www.EngineeringBooksPdf.com

While writing this book, AMD has made its OpenCL profiler and a generally available debugger named CodeXL found at `http://developer.amd.com/tools-and-sdks/heterogeneous-computing/codexl/`. Intel has a similar package offered separately and you can refer to the following URL for more details:

`http://software.intel.com/en-us/vcsource/tools/opencl-sdk-2013`. As for NVIDIA GPGPUs, you can only use the APIs provided by OpenCL.

Getting ready

The two operations that OpenCL allows the developer to have such insight into their runtimes are data transfer operations and kernel execution operations; the times are all measured in nanoseconds.

Since all devices cannot resolve to a nanosecond, it's important to determine what is the level of resolution, and you can know this by passing the `CL_DEVICE_PROFILING_TIMER_RESOLUTION` flag to `clGetDeviceInfo` for the appropriate device ID.

How to do it...

All you have to do is to pass the `CL_QUEUE_PROFILING_ENABLE` flag as part of the `properties` argument, when you create the command queue via `clCreateCommandQueue`. The API looks like this:

```
cl_command_queue
clCreateCommandQueue(cl_context context,
                     cl_device_id device,
                     cl_command_queue_properties properties, cl_int*
error_ret);
```

Once the profiling is enabled, the next thing you need to do is to inject OpenCL events into areas of the code, where you want to know how the runtimes fare. To achieve this, you need to create a `cl_event` variable for the regions of code you wish to monitor and associate this variable with one of the following APIs:

- Data transfer operations:
  - `clEnqueue{Read|Write|Map}Buffer`
  - `clEnqueue{Read|Write|Map}BufferRect`
  - `clEnqueue{Read|Write|Map}Image`
  - `clEnqueueUnmapMemObject`

www.EngineeringBooksPdf.com

- clEnqueuCopyBuffer
- clEnqueueCopyBufferRect
- clEnqueueCopyImage
- clEnqueueCopyImageToBuffer
- clEnqueueCopyBufferToImage

- Kernel operations:
  - clEnqueueNDRangeKernel
  - clEnqueueTask
  - clEnqueueNativeTask

How it works...

The way to obtain the runtimes for these operations is to invoke the `clGetEventProfilingInfo` API, passing in one of these flags: `CL_PROFILING_COMMAND_QUEUED`, `CL_PROFILING_COMMAND_SUBMIT`, `CL_PROFILING_COMMAND_START`, or `CL_PROFILING_COMMAND_END`. The API looks like this:

```
cl_int
clGetEventProfilingInfo(cl_event event,
                        cl_profiling_info param_name,
                        size_t param_value_size,
                        void* param_value,
                        size_t* param_value_size_ret);
```

To obtain the time spent by the command in the queue, you invoke `clGetEventProfilingInfo` with `CL_PROFILING_COMMAND_SUBMIT` once, and at the end of the code region invoke `clGetEventProfilingInfo` with `CL_PROFILING_COMMAND_QUEUED` again to get the difference in time.

To obtain the duration that the command took to execute, invoke `clGetEventProfilingInfo` once with `CL_PROFILING_COMMAND_START` and invoke the same API with `CL_PROFILING_COMMAND_END`, from the difference in the runtimes you will obtain the value.

www.EngineeringBooksPdf.com

The following is a small code snippet which illustrates the basic mechanism:

```
cl_event readEvt;
cl_ulong startTime;
cl_ulong endTime;
cl_ulong timeToRead;
cl_command_queue queue = clCreateCommandQueue(context, device, CL_
QUEUE_PROFILING_ENABLE, NULL);
clEnqueueReadBuffer(queue, some_buffer, TRUE, 0, sizeof(data), data,0,
NULL, &readEvt);
clGetEventProfilingInfo(readEvt, CL_PROFILING_COMMAND_START,sizeof(sta
rtTime),&startTime, NULL);
clGetEventProfilingInfo(readEvt, CL_PROFILING_COMMAND_
END,sizeof(endTime),&endTime, NULL);
timeToRead = endTime - startTim;
```


www.EngineeringBooksPdf.com

www.EngineeringBooksPdf.com

7

Developing the Matrix Multiplication with OpenCL

In this chapter, we will cover the following recipes:

- Understanding matrix multiplication
- OpenCL implementation of the matrix multiplication
- Faster OpenCL implementation of the matrix multiplication by thread coarsening
- Faster OpenCL implementation of the matrix multiplication through register tiling
- Reducing global memory via shared memory data prefetching in matrix multiplication

Introduction

In this chapter, we are going to take a look at the problem of multiplying two matrices to produce another matrix. This problem is also known as the matrix multiplication and its applications range from mathematics, finance, physics, and it is a popular system for solving linear equations. For illustration purposes, we present a typical use case for solving linear equations:

$$ax + by = c$$
$$dx + ey = f$$

www.EngineeringBooksPdf.com

These equations can be modeled as $\begin{bmatrix} a & b \\ d & e \end{bmatrix} * \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} c \\ f \end{bmatrix}$, where the L.H.S of the equation consists of a 2 x 2 matrix which is multiplied by a 2 x 1 matrix (often called a vector, and they can be row vectors or column vectors) which is equal to the vector on the R.H.S. Considering the fact that matrices can have any order of rows and columns, mathematicians invented the notation, $Ax = b$ where to solve this, we have to determine $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} c \\ f \end{bmatrix} * \begin{bmatrix} a & b \\ d & e \end{bmatrix}^{-1}$.Here, as we can see that the inverse of the matrix needs to be known. At this point, that's all we like to say about the wonderful world of matrices, lest we fall into the rabbit hole!

You should be aware that only square matrices have inverses, and even among such matrices the inverses are not guaranteed to be present. We won't be covering computing inverses in this chapter or book.

Understanding matrix multiplication

The product C of two matrices A and B is defined as $c_{ik} = a_{ij} b_{jk}$, where j is the sum of all possible values of i and k. There is an implied summation over the indices i, j, and k. The dimensions of the matrix C is: $(n \times m)(m \times p) = (n \times p)$, where $(a \times b)$ denotes a matrix with a rows and b columns and when we write out the product explicitly, it looks as follows:

$$c_{11} = a_{11} b_{11} + a_{12} b_{12} + \ldots + a_{1m} b_{m1}$$

$$\begin{bmatrix} c_{11} & c_{12} & \ldots & c_{1p} \\ c_{21} & c_{22} & \ldots & c_{2p} \\ - & - & - & - \\ c_{n1} & c_{n2} & \ldots & c_{np} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1p} \\ a_{21} & a_{22} & \ldots & a_{2p} \\ - & - & - & - \\ a_{n1} & a_{n2} & \ldots & a_{np} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \ldots & b_{1p} \\ b_{21} & b_{22} & \ldots & b_{2p} \\ - & - & - & - \\ b_{n1} & b_{n2} & \ldots & b_{np} \end{bmatrix}$$

$$c_{12} = a_{11} b_{12} + a_{12} b_{22} + \ldots + a_{1m} b_{m2}$$

$$c_{1p} = a_{11} b_{1p} + a_{12} b_{2p} + \ldots + a_{1m} b_{mp}$$

$$c_{21} = a_{21} b_{11} + a_{22} b_{21} + \ldots + a_{2m} b_{m1}$$

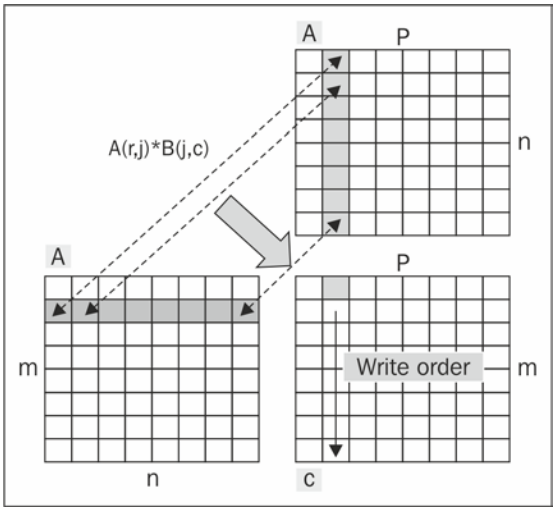
$$c_{np} = a_{n1} b_{1p} + a_{n2} b_{2p} + \ldots + a_{nm} b_{mp}$$

www.EngineeringBooksPdf.com

Another property of matrix multiplication is that multiplication is associative and distributive over addition, but they are however not commutative.

Two matrices A and B are considered commutative if they are diagonal matrices and are of the same dimension.

Knowing these properties will help us in formulating our initial algorithm stemming from this formula: $c_{ik} = a_{ij}b_{jk}$. The commutative property basically informs us that the order of multiplication between matrices A and B matters, while the associative property allows us the flexibility to explore what happens when two matrices A and B are too huge to fit into available memory on the OpenCL device and we need to partition the matrix data across multiple devices. The following diagram illustrates what happens when a row of matrix A and a column of matrix B is read and its aggregated result is written into the appropriate location in the output matrix, C:



Getting ready

At this point, we are in pretty good shape to take a stab at matrix multiplication. As before, we begin with an implementation in C/C++, which is a direct translation of the formula and from there we will develop a better intuition on how to import it to OpenCL and apply suitable optimizations.

www.EngineeringBooksPdf.com

For the rest of this chapter, we are going to craft our algorithm so that it runs on the GPU on your desktop/laptop. The reason for this is because the GPU has more computation units than a CPU, and GPUs are often equipped with other hardware components that allows the OpenCL to take advantage of that hardware (including local data stores, out of order execution units, shared data store, and so on), which often allows an enormous number of threads to execute in. Current CPU processors don't implement OpenCL shared memory, so using GPUs is probably the best option!

Get a GPU that supports OpenCL 1.1 and the preceding information is good enough for these experiments.

How to do it...

By now, you should be familiar with creating the necessary data structures to represent our three matrices in question (let's call them A, B, and C). Coincidentally, they happen to be square matrices, but this does not affect our understanding in any way.

When we examine this problem from the previous section, we understand that we want to basically iterate through both matrices in the following fashion:

1. Pick a row from matrix A.
2. Pick a column from matrix B.
3. Multiply each element from the picked row with the corresponding element from the picked column.

From this description, we can begin to think of various implementation methods and one such method could be as follows:

1. Create two in-memory data structures for A and B, say `TmpA` and `TmpB`.
2. Loop through A and pick a row for which each element to deposit into its corresponding position in `TmpA`, do the same for a picked column and deposit into `TmpB`:

```
loop until i < number_of_rowsA:
  TmpA[i] = A[i]
endloop
loop until i < number_of_colsB:
  TmpB[i] = B[i]
endloop
```

3. Loop through `TmpA` and `TmpB` and perform the matrix multiplication.

www.EngineeringBooksPdf.com

4. In pseudo code, it looks something like this:

```
loop until (i,j) < (rowA * colB):
  loop through A[i][_] deposit values into TmpA
  loop through B[_][j] deposit values into TmpB
  foreach value in TmpA and TmpB:
    C[a] = TmpA[x] * TmpB[y]
endloop
```

Another implementation is very similar to this one with the exception that we use standard C/C++ array indexing techniques to reference the respective row(s) and column(s) and we present an implementation in the following sections.

How it works...

There are various ways of implementing matrix multiplication algorithm in C/C++ as we've discussed previously. And it seems that there isn't a best design to adopt. Personally, I've always favored a readable design versus a convoluted design. However, it's necessary to write high performance code from time to time, so that you can squeeze all the power that the programming language or hardware can provide.

At this point, you may or may not have developed the necessary intuition to design your algorithms, but one way is to continuously practice using different techniques and measure each implementation with some benchmarks, and never clump all the optimizations in one algorithm unless you're confident.

Now that we have some inkling as to what is meant by matrix multiplication, it is definitely time for us to start exploring what the algorithm looks like after being translated into its sequential form. The following is an example of the matrix multiplication program in sequential form (the code is executed by only one thread):

```
Void matrixMul(float *C,
               const float *A,
               const float *B,
               unsigned int hA,
               unsigned int wA,
               unsigned int wB) {
    for (unsigned int i = 0; i < hA; ++i)
        for (unsigned int j = 0; j < wB; ++j){
            float sum = 0;
            for (unsigned int k = 0; k < wA; ++k) {
                double a = A[i * wA + k]; // statement 1
                double b = B[k * wB + j]; // statement 2
                sum += a * b;
```


www.EngineeringBooksPdf.com

```
                }

                C[i * wB + j] = (float)sum; // statement 3
            }
    }
```

When you examine this code, you will notice that there are three loop structures and we use regular C/C++ array indexing techniques to reference each subsequent element from their respective rows and columns. Take some time now to convince that we are actually computing the matrix multiplication.

As before, we put on our parallel developer hat and try to see how we can provide a parallel OpenCL form of the equivalent program. Again, I'm naturally drawn to the loop structures and we have three of them!

We noticed that as we iterate through the matrices A and B, the innermost loop is the code block that is performing all the heavy lifting for `statement 1`, `statement 2`, and `statement 3`. These statements will represent the core of our OpenCL kernel and let's go and take a look at how we can map it to OpenCL.

OpenCL implementation of the matrix multiplication

We have spent a good amount of time understanding how matrix multiplication works and we've looked at how it looks in its sequential form. Now we're going to attempt to map this to OpenCL in the most direct way.

The implementation technique here makes use of the fact that we create 2D thread blocks where each thread/work item in each dimension will access their respective elements in the row/column dimension.

Getting ready

In this recipe, we are going to use two matrices of dimensions 1024 x 1024 (we call A and B), and we'll multiply these two matrices together to produce a third matrix of 1024 x 1024, we call C.

You may wish to refresh your basic matrix theory at this point to convince yourself that this is the case.

www.EngineeringBooksPdf.com

We construct the familiar data structures in our host code and fill them with random values. The host code in `Ch7/matrix_multiplication_01/MatrixMultiplication.c` looks as follows:

```
matrixA = (cl_int*)malloc(widthA * heightA * sizeof(cl_int));
matrixB = (cl_int*)malloc(widthB * heightB * sizeof(cl_int));
matrixC = (cl_int*)malloc(widthB * heightA * sizeof(cl_int));

memset(matrixA, 0, widthA * heightA * sizeof(cl_int));
memset(matrixB, 0, widthB * heightB * sizeof(cl_int));
memset(matrixC, 0, widthB * heightA * sizeof(cl_int));

fillRandom(matrixA, widthA, heightA, 643);
fillRandom(matrixB, widthB, heightB, 991);
```

Next, we set up the OpenCL command queue to enable profiling because we want to keep looking at the effects of the subsequent optimizations that we are going to apply. It's definitely very important to establish a reference point to which your measurements can be compared against.

Recall that OpenCL command queues can be created such that commands are executed out-of-order. In this book, all command queues are created in-order so that they execute in program order also known as program reading order.

How to do it...

We present our first attempt to provide you an OpenCL version of the sequential matrix multiplication algorithm. The kernel can be found in `Ch7/matrix_multiplication_01/simple_mm_mult.cl`:

```
__kernel void mmmult(int widthB,
                     int heightA,
                      __global int* A,
                      __global int* B,
                      __global int* C) {

    int i = get_global_id(0);
    int j = get_global_id(1);
    int tmp = 0;

    if ((i < heightA) && (j < widthB)) {
        tmp = 0;
        for(int k = 0; k < widthB; ++k) {
```


www.EngineeringBooksPdf.com

```
                tmp += A[i*heightA + k] * B[k*widthB + j];
            }
            C[i*heightA + j] = tmp;
        }
    }
```

Given the preceding OpenCL kernel code, we need to build an executable so that it can execute on your platform. As before, the compilation will look familiar to you. On my setup with an Intel Core i7 CPU & AMD HD6870x2 GPU running Ubuntu 12.04 LTS, the compilation looks like this and it'll create an executable called `MatrixMultiplication` into the directory:

```
gcc -std=c99 -Wall -DUNIX -g -DDEBUG -arch i386 -o MatrixMultiplication
-framework OpenCL
```

At this point, you should have an executable deposited in that directory and all you need to do now is to run the program, simply execute the `MatrixMultiplication` program in the directory and you should have noticed an output as follows:

```
Passed!
Execution of matrix-matrix multiplication took X.Xs
```

How it works...

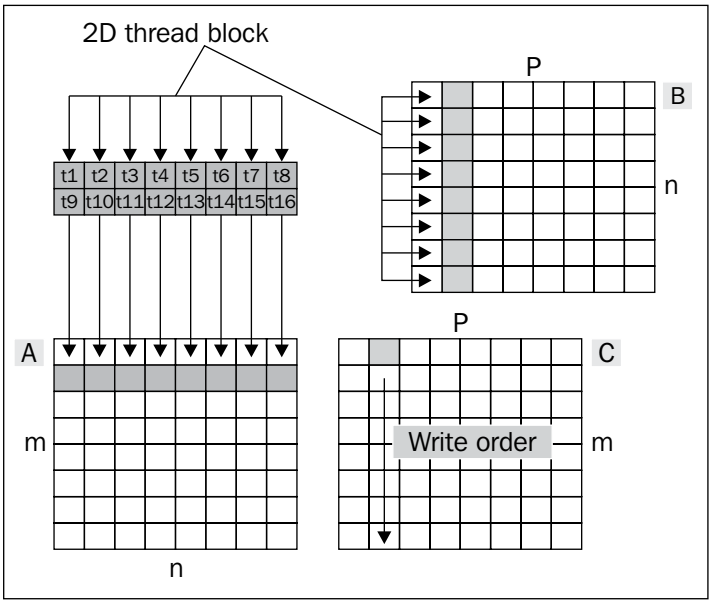
We discussed how the matrices were initialized and the next thing is to realize the execution model where each work item in each dimension would work on each element. And to accomplish this, we have to ensure that the invocation to execute the OpenCL kernel code doesn't dictate the size of the thread block:

```
size_t globalThreads[] = {widthB, heightA};

cl_event exeEvt;
cl_ulong executionStart, executionEnd;
error = clEnqueueNDRangeKernel(queue,
                               kernel,
                               2,
                               NULL,
                               globalThreads,
                               NULL,
                               0,
                               NULL,
                               &exeEvt);
clWaitForEvents(1, &exeEvt);
```


www.EngineeringBooksPdf.com

We achieve this by passing in the `NULL` value to the placeholder meant for dictating work group size in the `clEnqueueNDRangeKernel` API. Next, we set the values of the global work items to be equivalent to that of width of matrix B and height of A represented by the `widthB` and `heightA` variables respectively.

The following diagram serves to illustrate what the execution would have looked like:



An astute reader would probably start guessing that this isn't the best way to conduct this business and you're right! We are going to take a deeper look at how we can make this work better soon.

Faster OpenCL implementation of the matrix multiplication by thread coarsening

In this section, let's try to make this beast run faster by applying a technique in parallel programming: thread coarsening. This is important because when you have a work item accessing an element, and then you have large matrices you could potentially have millions of work items running! In general, that's not a good thing because many devices today cannot support millions of work items in *n* dimensions unless it's a supercomputer. But there are often clever ways to reduce the amount of work items needed.

www.EngineeringBooksPdf.com

Getting ready

The general technique here is to explore ways in which we can merge threads so that each thread now calculates multiple elements. When we reexamine the preceding code, we might wonder if we could do with fewer threads and have them compute more elements, and indeed we can.

The strategy we have adopted will basically have one work item updating an entire row in the matrix C while walking through matrices A and B. At this time, we need not even explore the use of atomic functions in OpenCL, since that's an aspect we should try to delay exploring as long as possible. The main reason for not exploring the use of atomics is simply because their execution time is too long and it isn't mature of utilizing the capabilities of the OpenCL devices.

How to do it...

This OpenCL kernel is revised based on the concept of thread coarsening and can be found in `Ch7/matrix_multiplication_02/mmult.cl`:

```
__kernel void mmmult(int widthB,
                     int heightA,
                      __global int* A,
                      __global int* B,
                      __global int* C) {

    int i = get_global_id(0);
    int tmp = 0;

    if (i < heightA) {
        for(int j = 0; j < widthB; ++j) {
            tmp = 0;
            for(int k = 0; k < widthB; ++k) {
                tmp += A[i*heightA + k] * B[k*widthB + j];
            }
            C[i*heightA + j] = tmp;
        }
    }
}
```

Now that we have taken a good look at the OpenCL kernel, we need to build an executable form. As before, the compilation will look familiar to you. On my setup with an Intel Core i7 CPU & AMD HD6870x2 GPU running Ubuntu 12.04 LTS the compilation looks as follows, and it'll create an executable called `MatrixMultiplication` into the directory:

www.EngineeringBooksPdf.com

```
gcc -std=c99 -Wall -DUNIX -g -DDEBUG -arch i386 -o MatrixMultiplication
-framework OpenCL
```

At this point, an executable should have been deposited in the directory and to execute it, simply execute the program `MatrixMultiplication` in the directory and you should have noticed an output as follows:

```
Passed!
Execution of matrix-matrix multiplication took X.Xs
```

Now if you were to compare the results with the previous one you would notice that it is running faster!

How it works...

The hard part of this is being able to recognize when redundant work is being applied. But in our case, it won't take too much effort to recognize that we are actually using too many threads. How so you may ask? The clue lies in the fact that the original matrix multiplication algorithm ran with one executing thread, so the fact that we are using more than one work item does imply that there's more we can do to improve it.

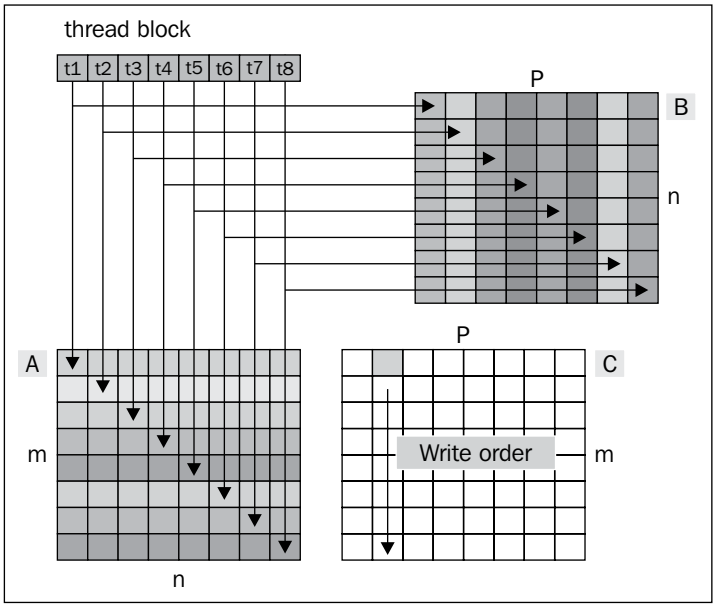
Hence when we look back at the algorithm, we discover a way to make them run faster by getting more creative in the way we obtain those values using one work item. At this point, you should convince yourself that the OpenCL kernel we just looked at is indeed referencing the data values from the matrices A and B as expected.

To achieve what we did, we made some changes to the code in `Ch7/matrix_multiplication_02/MatrixMultiplication.c` as follows:

```
    size_t globalThreads[] = {heightA};
    size_t localThreads[] = {256};
    cl_event exeEvt;
    cl_ulong executionStart, executionEnd;
    error = clEnqueueNDRangeKernel(queue,
                                   kernel,
                                   1,
                                   NULL,
                                   globalThreads,
                                   localThreads,
                                   0,
                                   NULL,
                                   &exeEvt);
    clWaitForEvents(1, &exeEvt);
```


www.EngineeringBooksPdf.com

The problem size is known to us, which is to perform matrix multiplication for matrices of dimensions 1024 x 1024 and the reason why I chose the work group size to be 256 is because my GPU has four compute units and you can discover this by passing `CL_DEVICE_MAX_COMPUTE_UNITS` to `clGetDeviceInfo`. The following diagram illustrates what it is like with thread coarsening:



When you are able to reduce redundant work through thread coarsening, the kernel would now execute faster and scale better because now more processors can execute. It may seem counter intuitive because it defies common sense, since more threads executing the kernel means that it should execute faster. Well, that's the simple picture.

What happens under the hood is more complicated and it starts from the fact that each GPU has a number of processors and each of those processors would execute the kernel. For a GPU to be able to execute at full capacity, naturally its processors must be filled with data in the data cache and instructions should be ready to be fired and execute the OpenCL kernel.

However due to poor data spatial and temporal locality, the data caches perform suboptimal and that causes stalls in the instruction pipeline, which translates to delayed execution. Another problem is also related to the fact that memory access patterns could be erratic or non-coalesced which translates to cache misses and possibly memory ejection. This finally causes more delays.

Coming back to the problem, there is another solution for optimizing the kernel and that's by reusing the hardware registers of the work items.

www.EngineeringBooksPdf.com

Faster OpenCL implementation of the matrix multiplication through register tiling

Register tiling is another technique we can apply to our matrix multiplication algorithm. What it basically means is to explore opportunities to reuse the hardware registers. In our case, what it means is that we need to examine the kernel code and find opportunities to reuse registers.

Now we need to put on our hardcore C developer hat (this person needs to think on the level of the processor core, how data moves on buses, memory loads and stores, and so on). And once your mind is sensitive enough to this level, then things become better.

Recall the kernel code in the previous section and we would notice after careful scrutiny that the `A[i * heightA + k]` statement is always executed in the loop structure, and this causes a lot of memory traffic to transpire because data needs to be loaded from device memory into the registers of the device.

Getting ready

To reduce the global memory traffic caused by the `A[i * heightA + k]` statement, we can pull that statement out of the loop structure and create a thread local memory structure that is visible only to the work item executing thread, and then we can reuse that prefetched data in the subsequent computations.

How to do it

This OpenCL kernel code is found in `Ch7/matrix_multiplication_03/mmult.cl`:

```
__kernel void mmmult(int,
                     int widthB heightA,
                      __global int* A,                      __global
int* B,
                      __global int* C) {

    int i = get_global_id(0);

    int tmp = 0;

    int tmpData[1024];

    if (i < heightA) {
        for(int k = 0; k < widthB; ++k )
            tmpData[k] = A[i*heightA + k];

        for(int j = 0; j < widthB; ++j) {
```


www.EngineeringBooksPdf.com

```
                tmp = 0;
                for(int k = 0; k < widthB; ++k) {
                    tmp += tmpData[k] * B[k*widthB + j];
                }
                C[i*heightA + j] = tmp;
            }
        }
    }
```

Now that we have taken a good look at the OpenCL kernel, we need to build an executable form, where we can execute. As before, the compilation will look familiar to you. On my setup with an Intel Core i7 CPU & AMD HD6870x2 GPU running Ubuntu 12.04 LTS, the compilation looks like this and it'll create an executable called `MatrixMultiplication` into the directory:

```
gcc -std=c99 -Wall -DUNIX -g -DDEBUG -arch i386 -o MatrixMultiplication
-framework OpenCL
```

At this point, the executable should be available to you in the directory. To run the program, simply execute the program in the `MatrixMultiplication` directory and you should notice an output as follows:

```
Passed!
Execution of matrix-matrix multiplication took X.Xs
```

Now if you were to compare the results with the previous one you would notice that it is running faster.

How it works...

The idea originated from a technique found in high performance computing and some folks like to call it scalar replacement. This is the form we have applied in this section. Let's take some time to understand this with a simple algorithm.

Let's say we have the following algorithm:

```
for i1 = 1 to 6
  for i2 = 1 to 6
    A[i1,i2] = A[i1 – 1, i2] + A[i1,i2 -2]
```

Now we unroll the loop so that it looks like this:

```
for i1 = 1 to 6 step-by-2
  for i2 = 1 to 6 step-by-2
    A[i1,i2] = A[i1 –1, i2] + A[i1,i2 -2]    //statement 1
    A[i1 +1,i2] = A[i1,i2] + A[i1+1,i2 -1]    //statement 2
    A[i1,i2 +1] = A[i1 –1, i2+1] + A[i1,i2]   //statement 3
    A[i1+1,i2+1] = A[i1, i2 +1] + A[i1+1,i2]
```


www.EngineeringBooksPdf.com

When we will carefully observe this code, we will notice that the `statement 1`, `statement 2`, and `statement 3` have something in common and that is this code, `A[i1,i2]`. In computer science terms, we noticed that there is one store to memory and two loads from memory to registers. In scalar replacement, we replace `A[i1,i2]` with a variable, which we call `X` for now. The code now looks as follows after scalar replacement:

```
for i1 = 1 to 6 step-by-2
  X = A[i1,0]
  for i2 = 1 to 6 step-by-2
    X          = A[i1 -1, i2] + X
    A[i1 +1,i2] = X + A[i1+1,i2 -1]
    A[i1,i2 +1] = A[i1 -1, i2+1] + X
    A[i1+1,i2+1] = A[i1, i2 +1] + A[i1+1,i2]
     A[i1,i2] = X
```

When the replacements have been done consistently and the algorithm is still working as it should, we are good for now. Have a cup of tea!

Let's have a look at what we did. We have replaced array references (which are in fact memory references) with scalars, and how it helps is that we have actually reduced memory traffic by processing those items in register memory. Considering that memory speed is significantly much slower than register read-write speed, this revised algorithm is in much better form.

Loop unrolling is often used to explode the loop, so that we can identify expressions or statements that can possibly be repeating and allowing scalar replacement to extract those expressions/statements into thread private register memory.

Scalar replacement is actually more complicated in actual practice, but the presentation here serves its purpose in illustrating the general concept.

Another thing we like to share with you is to optimize memory usage for the work items and we've caught several glimpses of it before in previous chapters.

Reducing global memory via shared memory data prefetching in matrix multiplication

Our revised matrix multiplication algorithm appears to be pretty good but it isn't quite there yet. The algorithm is still making a lot of references to matrix B over global memory and we can actually reduce this traffic by prefetching the data. You may not have noticed, but the concept of prefetching, which is to keep the cache "hot" (an idea borrowed from the CPU). A CPU typically has a good size of data and instruction caches (which are really hardware registers), so that the processor can take advantage of the spatial and temporal localities of the data. How does this concept map into other OpenCL devices, for example, the GPU?

www.EngineeringBooksPdf.com

Every GPU that is an OpenCL compliant has a small amount of memory designed for this purpose and their sizes typically are 32 KB to 64 KB. If you wish to determine the exact amount of available high speed memory, simply pass the `CL_DEVICE_LOCAL_MEM_SIZE` variable to `clGetDeviceInfo` for a device.

Getting ready

In order for us to be able to reduce references to global memory, we need to make changes in our code so that we load the data we need. Sieving through the code again, we see that there is indeed one such opportunity and it is the following statement:

```
for(int j = 0; j < widthB; ++j) {
    tmp = 0;
    for(int k = 0; k < widthB; ++k) {
        tmp += tmpData[k] * B[k*widthB + j];
    }
//more code omitted
}
```

Concentrating on this loop, we noticed that matrix B always gets loaded and its values are always reused by all work items executing this kernel. We could of course preload this data into shared memory. That should reduce global memory requests significantly.

How to do it...

The following OpenCL kernel can be found in `Ch7/matrix_multiplicatione_04/mmult.cl`:

```
__kernel void mmmult(int widthB,
                     int heightA,
                      __global int* A,
                      __global int* B,
                      __global int* C,
                      __local  int* shared) {

    int i = get_global_id(0);
    int id = get_local_id(0);
    int size = get_local_size(0);
    int tmp = 0;

    int tmpData[1024];

    if (i < heightA) {
        /*
         Pre-load the data into the work-item's register memory that
```


www.EngineeringBooksPdf.com

```
is
          Visible to the work-item only.
          */
         for(int k = 0; k < widthB; ++k ) {
             tmpData[k] = A[i*heightA + k];
         }

         /*
          Data pre-fetching into shared memory allows all work-items
          To read the data off it instead of loading the data from
global
          Memory for every work-item
         */
         for(int k = id; k < widthB; k += size)
             shared[k] = B[k*widthB +k];
         barrier(CLK_LOCAL_MEM_FENCE);

         for(int j = 0; j < widthB; ++j) {
             tmp = 0;
             for(int k = 0; k < widthB; ++k) {
                 tmp += tmpData[k] * shared[k];
             }
             C[i*heightA + j] = tmp;
         }
     }
}
```

Now that you have taken a look at the OpenCL kernel, you would want to compile the code and run it. As before the compilation will look familiar to you. On my setup with an Intel Core i7 CPU and AMD HD6870x2 GPU running Ubuntu 12.04 LTS, the compilation looks like this and it'll create an executable called `MatrixMultiplication` into the directory.

```
gcc -std=c99 -Wall -DUNIX -g -DDEBUG -arch i386 -o MatrixMultiplication
-framework OpenCL
```

To run the program, simply execute the `MatrixMultiplication` program in the directory and you should get an output that resembles this:

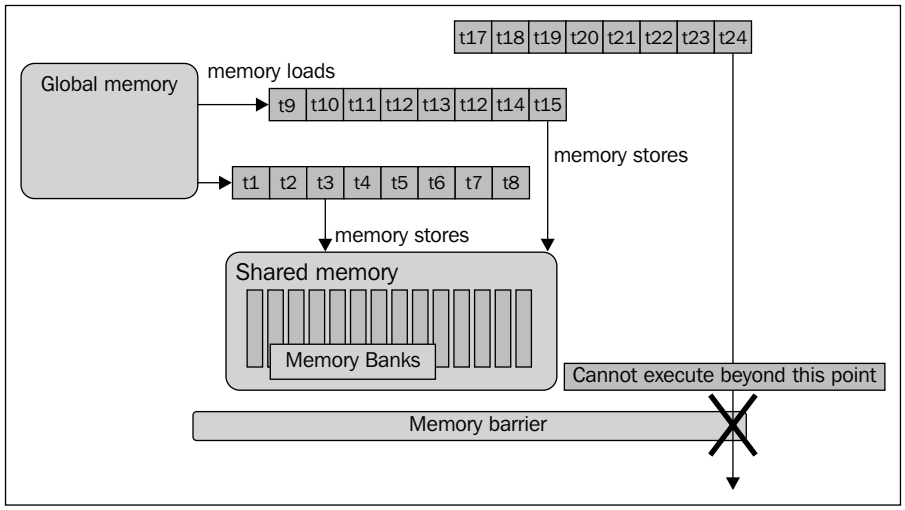
```
Passed!
Execution of matrix-matrix multiplication took X.Xs
```

Now if you were to compare the results with the previous one, you would notice that it is running much faster!

www.EngineeringBooksPdf.com

How it works...

The code that we have introduced might cast some doubts within yourself that because it looks sequential, it is actually executed in parallel during runtime. The parallelism is introduced by the value indicated in the `localThreads` variable, which is passed to `clEnqueueNDRangeKernel`. The memory barrier we placed into the code serves to stop all work items from executing beyond that point, until all functions before that point have been executed and the following diagram serves to illustrate this:



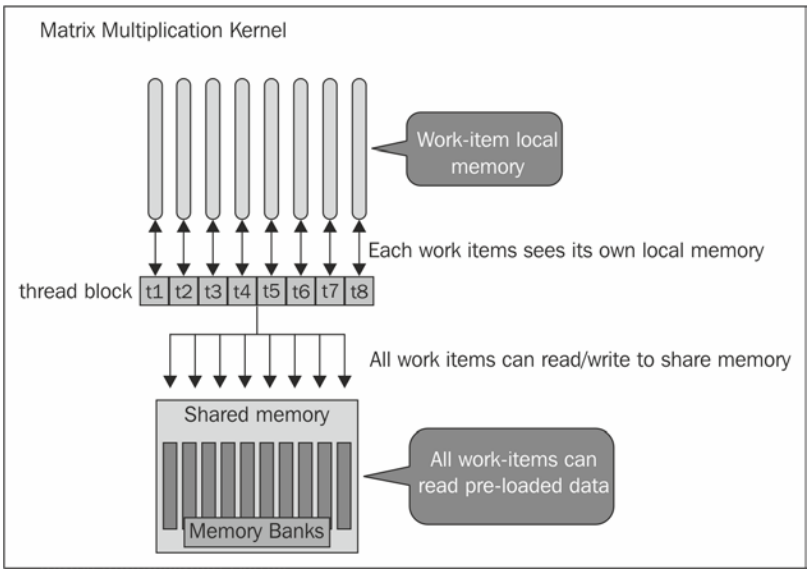
So far you have seen changes made to the OpenCL kernel code, and now we need to make changes to our host code so that we can actually accomplish this. The following code snippet is taken from `Ch7/matrix_multiplication_04/MatrixMultiplication.c`:

```
clSetKernelArg(kernel, 0, sizeof(cl_int),(void*)&widthB);
clSetKernelArg(kernel, 1, sizeof(cl_int),(void*)&heightA);
clSetKernelArg(kernel, 2, sizeof(cl_mem),(void*)&matrixAMemObj);
clSetKernelArg(kernel, 3, sizeof(cl_mem),(void*)&matrixBMemObj);
clSetKernelArg(kernel, 4, sizeof(cl_mem),(void*)&matrixCMemObj);
clSetKernelArg(kernel, 5, sizeof(cl_int)*heightA,NULL);

size_t globalThreads[] = {heightA};
size_t localThreads[] = {256};
cl_event exeEvt;
cl_ulong executionStart, executionEnd;
error = clEnqueueNDRangeKernel(queue,
                               kernel,
                               1,
                               NULL,
```


www.EngineeringBooksPdf.com

```
                                    globalThreads,
                                    localThreads,
                                    0,
                                    NULL,
                                    &exeEvt);
    clWaitForEvents(1, &exeEvt);
```

The schematics of the final algorithm have seen us tailoring the algorithm, so that it achieves an initial reasonable performance and can be conceptually represented by the following diagram:



If you want to know how much shared memory you can possibly create and pass the `CL_DEVICE_LOCAL_MEM_SIZE` parameter to `clGetDeviceInfo` for your device and the value returned will be in bytes. Typical values are between 32 KB to 64 KB.

www.EngineeringBooksPdf.com

www.EngineeringBooksPdf.com

8

Developing the Sparse Matrix Vector Multiplication in OpenCL

In this chapter, we are going to cover the following recipes:

- Solving the **SpMV** (**Sparse Matrix Vector Multiplication**) using the conjugate gradient method
- Understanding the various SpMV data storage formats including ELLPACK, ELLPACK-R, COO, and CSR
- Understanding how to solve SpMV using the ELLPACK-R format
- Understanding how to solve SpMV using the CSR format
- Understanding how to solve SpMV using VexCL

Introduction

In the previous chapter on matrix multiplication, we developed an appreciation of the problem space as well as its domain of application, but what we didn't tell you earlier was that there are dense matrices as well as sparse matrices in addition to their dense and sparse vectors. When we say dense or sparse matrix/vector, we mean that there are a lot of non-zero or zero values, respectively.

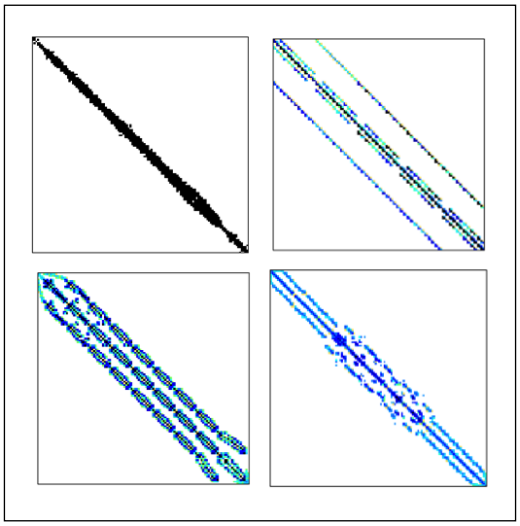
www.EngineeringBooksPdf.com

The fact that a matrix is dense or sparse matters from a computational point of view, since it doesn't really make sense to multiply any value with zero as the result is evidently zero; if you were to apply the naïve method of solving this problem, which is to use the methods you developed during the matrix multiplication to solve the problem where the matrix or vector is sparse, but you would not be taking advantage of that brand new OpenCL CPU/GPU you just bought, you are simply wasting processor cycles and also wasting massive amounts of bandwidth. The question lies in solving this problem in an efficient manner and this requires understanding how to compute this efficiently, which solves one part of the issue. The other part of this issue is to investigate how to store the sparse matrices efficiently, since allocating a $(m \times n)$ matrix to store a matrix that is populated with mostly zeroes is wasteful of memory space.

We are going to take a whirlwind tour of this subject, however it will not be exhaustive. There is a lot of literature already published on this subject. However, we will spend some time to formulate a basic and general idea by recognizing that most of the past and current work focuses on a combination of creating data structures that are efficient and compact to represent the sparse structures. We will also spend some time devising efficient computational methods on those data structures. As far as matrices go, we won't look into the possibilities of dynamic matrices (via insertion or deletion), and instead we will focus on static sparse matrix formats.

Next, we are going to present the theory behind solving SpMV efficiently through building up our knowledge to the conjugate gradient (via steepest descent and Gram-Schmidt), and before applying that algorithm we'll look into some of the common data storage schemes. We'll present an implementation using the VexCL using the **Conjugate Gradient** (**CG**) method which is an OpenCL framework build using C++.

The following are some of the examples of sparse matrices:



www.EngineeringBooksPdf.com

Solving SpMV (Sparse Matrix Vector Multiplication) using the Conjugate Gradient Method

The conjugate gradient method is the most popular iterative method for solving sparse linear systems, and I will attempt to make you understand how it works. Along this journey, we will look into steepest descent, conjugate gradient convergence, and so on.

I wanted to say a big thank you to *Jonathan Richard Shewchuk* (AP of University of California), without whom I might not have understood why conjugate gradients matter You can learn more about him at `http://www.cs.cmu.edu/~jrs/`.

A reason why the CG method is popular in solving sparse systems is that it not only handles really large sparse matrices well but it is also very efficient.

In the previous chapter on matrix multiplication, we have seen what it means to multiply two matrices, and this time round, we are focusing on the problem of $Ax=b$ where *A* is a known square and positive definite matrix, *x* is an unknown vector, and *b* is a known vector.

Getting ready

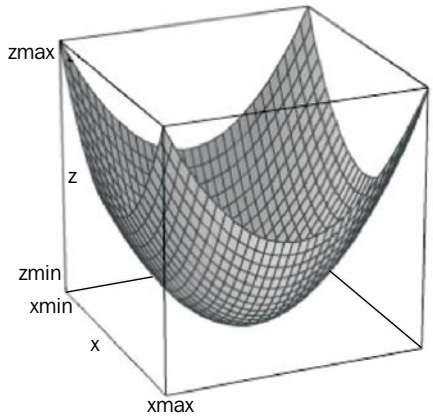
The inner product of two vectors is written as x^Ty, and it represents the scalar sum $\sum_{i=1}^{n} x_i y_i$. *xTy* is equivalent to *yTx*, and if *x* and *y* are orthogonal (at right angles to one another, and this will be important to realize when we study steepest descent), then *xTy = 0*.

A positive-definite matrix *A* is such that for every non-zero vector *x*, *xTAx > 0*.

A quadratic form is actually a scalar and quadratic function of a vector of the form as:

www.EngineeringBooksPdf.com

Just like any linear function, we would know its gradient that can be expressed in this derived form as [illegible] (yep, it's not a typo, and we mean the transpose of matrix *A*), and when we know that matrix *A* is symmetric, that is, $1/2(A^T + A)$ becomes *A* because *AT=A*, then this equation reduces to $f'(x) = Ax - b$. Like any derivate of a linear equation, we know that the mathematical solution to $f'(x)$ can be found when it is equal to *0* and by solving $Ax = b$. The goal is to find a particular value of *x* which minimizes $f(x)$. Diagrammatically, it can be imagined as a parabola like the one in the following diagram, which is what $f(x)$ evaluates to be exactly:



This forms our foundation to study the steepest descent and its cousin method—the conjugate gradient method. In the following sections, let us first explore the concepts behind steepest descent and then head over to conjugate gradient.

In the steepest descent method, we start at an arbitrary point $x_{(0)}$ and slide down to the bottom of the paraboloid. We keep taking steps *x(1)*, *x(2)*, and so on until we are pretty confident in saying that we have come to the solution *x*. That's basically how it works. Generally speaking, we haven't said anything about how to choose the next point to slide to though, as always the devil is in the details. Solder on!

When we take a step, we choose the direction in which $f(x)$ decreases most quickly, and now it's appropriate to introduce two vectors, which we will use to gauge for ourselves whether or not we're dropping in the right direction (that is, if we are moving towards the bottom of the parabola). The error vector $e_{(i)} = x_{(i)} - x$ measures how far we are from the solution from the current step. The residual vector $r_{(i)} = b - Ax_{(i)}$ measures how far we are from the correct value of *b*, and this vector can be thought of as the direction of steepest descent. When we take the next step so that we can be closer to the actual solution, *x*, we are actually choosing a point $x_{(i)} = x_{(i-1)} + \alpha r_{(i-1)}$, and you will notice that another variable has been chosen which is alpha, α.

www.EngineeringBooksPdf.com

This variable α of whichever value will tell us whether we have reached the bottom of the parabola. To put this another way, imagine yourself falling into a salad bowl (closest thing I could think of) and the only way you can stop falling is when you sit at the bottom of the bowl. We know from calculus that the derivative of that point (x, y) where you land is zero, that is, its gradient is also *0*. To determine this value, we have to set the derivative of that point to be equal to zero and we already have seen the equation $f'(x) = Ax - b$, and we know now that $f'(x) = -r_{(i)}$.

How to do it...

Let's now calculate the directional derivative of $\frac{d}{d\alpha} f(x_{(i)})$ when it is equal to zero because α minimizes *f*. Using the chain rule, we know that $\frac{d}{d\alpha} f(x_{(i)}) = f'(x_{(i)})^T r_{(i)} = 0$ and plugging in what we know of $f'(x)$, we have the following sequence of derivations by which we derive the value of α:

$$
\begin{aligned}
& r^T_{(i+1)} r_{(i)} = 0, \\
& (b - Ax_{(i+1)})^T r_{(i)} = 0, \\
& (b - A(x_{(i)} + \alpha r_{(i)}))^T r_{(i)} = 0, \\
& (b - Ax_{(i)})^T r_{(i)} - \alpha (Ar_{(i)})^T r_{(i)} = 0, \\
& (b - Ax_{(i)})^T r_{(i)} = \alpha (A(r_{(i)})^T r_{(i)}, \\
& r^T_{(i)} r_{(i)} = \alpha r^T_{(i)} A r_{(i)}, \\
& \alpha(i) = \frac{r^T_{(i)} r_{(i)}}{r^T_{(i)} A r_{(i)}}.
\end{aligned}
$$

In summary, the steepest descent comprises the following equations:

$$
\begin{aligned}
& r_{(i)} = b - Ax_{(i)}, \\
& \alpha(i) = \frac{r^T_{(i)} r_{(i)}}{r^T_{(i)} A r_{(i)}}, \\
& x_{(i+1)} = x_{(i)} + \alpha_{(i)} r_{(i)}.
\end{aligned}
$$

Using the steepest descent means is that I take a step down the rabbit hole and before I take the next step I'm going to guess what its going to be and take it; if I'm right, hooray!

www.EngineeringBooksPdf.com

The conjugate gradient method builds on steepest descent, and the two share a lot of similarities such that the conjugate gradient makes guesses which will eventually lead to the solution in *x*. Both methods use the residual vector to judge how far the guesses are from the correct answer.

The idea is to pick a set of orthogonal search directions, and in each direction we'll take exactly one step (pretty much the same as what we have seen before) $x_{(i+1)} = x_{(i)} + \alpha_{(i)} d_{(i)}$. It turns out that we need to make the search direction *A-orthogonal* instead of orthogonal. We say that two vectors $d_{(i)}$ and $d_{(j)}$ are A-orthogonal if $d_{(i)}^T A d_{(j)} = 0$. When we use a search direction, one of the things that we want to minimize is the amount of space in which we search, and for this we would need *linear independent vectors* $u_0, u_1, u_2, \ldots$. From there, we can use the Gram-Schmidt process to generate them and we would have the following:

$$d_{(i)} = u_{(i)} + \sum_{k=0}^{i-1} \beta_{ik} d_{(k)}$$

As we did in the steepest descent method, let's use the same trick to determine what β_{ik} is since it looks really familiar like α, and we derive it using the following:

$$d_{(i)} A d_{(j)} = u_i^T A d_{(j)} + \sum_{k=0}^{i-1} \beta_{ik} d_{(k)}^T A d_{(j)},$$
$$0 = u_i^T A d_{(j)} + \beta_{ij} d_{(j)}^T A d_{(j)},$$
$$\beta_{ij} = -\frac{u_i^T A d_{(j)}}{d_{(j)}^T A d_{(j)}}$$

From the previous equation, we plug in the fact that two vectors are A-orthogonal, that is, the left-hand side of the equation is *0*, and we solve for the right-hand side which resulted in β_{ik}. When we compare this value with α, we would discover that they are pretty much the same except for the fact that the CG method uses linear independent vectors instead of the residual vector, as found in steepest descent.

The CG method builds on the Gram-Schimdt process/conjugation and steepest descent, whereby it removes the presence of search vectors. It favors the use of residual vectors instead, and this is important from a computational point of view, otherwise your program would need to store all of the search vectors, and for a large domain space it would probably be a very bad idea. There is a fair bit of math that we skipped, but feel free to download the original paper from *Jonathan Shewchuk* from the following link

`http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf`

www.EngineeringBooksPdf.com

In the method of conjugate gradient, we have the following equations:

$$d_{(i)} = r_{(i)} = b - Ax_{(i)},$$
$$\alpha_{(i)} = \frac{r_{(i)}^T r_{(i)}}{d_{(i)}^T A d_{(i)}},$$
$$x_{(i+1)} = x_{(i)} + \alpha_{(i)} d_{(i)},$$
$$r_{(i+1)} = r_{(i)} - a_{(i)} A d_{(i)},$$
$$\beta_{(i+1)} = \frac{r_{(i+1)}^T r_{(i+1)}}{r_{(i)}^T r_{(i)}},$$
$$d_{(i+1)} = r_{(i+1)} + \beta_{(i+1)} d_{(i)}$$

We're going to see how we can translate this into OpenCL. But first, it's time for a cup of coffee!

Now that we have established a basic idea of what the CG method is like, its time to take a look at how a simple SpMV kernel can be implemented. However, recall that I mentioned that we have to understand how the data in the sparse matrix can be stored. That turns out to be crucial in the implementation, and it's justifiable to spend the next couple of sections illustrating to you the well-known data storage formats.

Understanding the various SpMV data storage formats including ELLPACK, ELLPACK-R, COO, and CSR

There are a wide variety of sparse matrix representations, each with a different storage requirement, even computational characteristics, and with those come the varieties in which you can access and manipulate elements of the matrix. I made a remark earlier that we will be focusing on static sparse matrix formats, and I present here four storage formats that have been proven to be rather popular not only because of the decent performance but also because they were also some of the earliest formats which have been popular among scalar and vector architectures, and quite recently, in GPGPUs.

www.EngineeringBooksPdf.com

In the following paragraphs, we are going to introduce you to the following sparse matrix representations in the following order:

- ELLPACK format
- ELLPACK-R format
- Coordinate format
- Compressed sparse row format

Let's start with the ELLPACK format. This format is also known as ELL. For an *M x N* matrix with a maximum of *K* non-zero values per row, the ELLPACK format stores the non-zero values into a dense *M x K* array which we'll name `data`, where rows with lesser than *K* non-zero values are zero padded. Similarly, the corresponding column indices are stored in another array, which we'll name `indices`. Again, a zero or some sentinel value is used for padding this array. The following representation of matrices illustrates what it looks like:

$$A=\begin{bmatrix}6&9&0&0\\0&2&8&4\\5&0&1&9\\0&0&0&1\end{bmatrix}$$

$$data=\begin{bmatrix}6&9&*\\2&8&4\\5&1&9\\1&*&*\end{bmatrix},indices=\begin{bmatrix}0&1&*\\1&2&3\\0&2&3\\3&*&*\end{bmatrix}$$

A quick analysis on this format means that if the maximum number of non-zero values in each row does not differ too much from the average, the ELL format is rather appealing because it is intuitive, at least to me.

Next, we examine the ELLPACK-R format. This format is a variant of the ELLPACK format, and in addition to the data arrays that you have seen earlier, we have a new array `rl`, which is used to store the actual length of each row. The following representation illustrates what it looks like:

$$A=\begin{bmatrix}6&9&0&0\\0&2&8&4\\5&0&1&9\\0&0&0&1\end{bmatrix}$$

$$data=\begin{bmatrix}6&9&*\\2&8&4\\5&1&9\\1&*&*\end{bmatrix},indices=\begin{bmatrix}0&1&*\\1&2&3\\0&2&3\\3&*&*\end{bmatrix},rl=\begin{bmatrix}2\\3\\3\\1\end{bmatrix}$$

www.EngineeringBooksPdf.com

It's not obvious now how this differs from ELLPACK, but the serial and parallel kernel which we will see later will make use of this new array to make the code and data transfers tighter.

We proceed with the coordinate format. The coordinate format is a simple storage scheme. The arrays `row`, `col`, and `data` store the row indices, column indices, and values, respectively of the non-zero matrix entries. COO is a general sparse matrix representation since the required storage is always proportional to the number of non-zero values. The following is what the COO format looks like:

$$A=\begin{bmatrix}1 & 7 & 0 & 0 & 1\\ 0 & 2 & 8 & 0 & 2\\ 0 & 0 & 6 & 4 & 0\\ 3 & 0 & 3 & 9 & 5\end{bmatrix}$$

row= [0,0,0,1,1,1,2,2,2,2,3,3,4,4,4,4]
col= [0,0,0,1,1,1,2,2,2,2,3,3,4,4,4,4]
data= [1,7,1,2,8,2,5,3,9,9,5,4,3,3,9,5]

In this format, there are three one-dimensional arrays—`row`, `col`, and `data`.

Last one on this list is the **Compressed Sparse Row** (**CSR**) format. The CSR format is a popular, general-purpose sparse matrix representation. Like the COO Format, CSR explicitly stores column indices and non-zero values in the arrays `indices` and `data`. A third array of row pointers, `ptr`, takes the CSR representation. For an *M x N* matrix, `ptr` has length *M + 1*, and stores the offset into the *i*th row in `ptr[i]`. The last entry in `ptr`, which would otherwise correspond to the $M + 1^{th}$ row, stores the number of non-zero values in the matrix. The following representation illustrates what it looks like:

$$A=\begin{bmatrix}1 & 7 & 0 & 0 & 1\\ 0 & 2 & 8 & 0 & 2\\ 5 & 0 & 3 & 9 & 9\\ 3 & 0 & 3 & 9 & 5\end{bmatrix}$$

ptr= [0,3,6,1,0,1,2,1,6]
indices= [0,1,4,1,2,4,0,2,3,4,2,3,0,2,3,4]
data= [1,7,1,2,8,2,5,3,9,9,6,4,3,3,9,5]

At this point, this is all I want to discuss about data representations for sparse matrices.

You should be aware that there are other formats like **DIA**, also known as, **diagonal format**, Hybrid/HYB for ELL/COO, and packet (for processors that resemble vector architectures).

www.EngineeringBooksPdf.com

How to do it...

Now that we have examined three data storage formats, let's go on a little further and check out how we would solve the SpMV problem using the ELLPACK format. As before, we would like to start this section by kicking off with a code presentation on how the SpMV CPU kernel would look:

```
// num_rows  - number of rows in matrix
// data      - the array that stores the non-zero values
// indices   - the array that stores the column indices for zero, non-
//              zero values in the matrix
// num_cols  - the number of columns.
// vec       - the dense vector
// y         - the output
void spmv_ell_cpu(const int num_rows,
                  const int num_cols,
                  const int * indices;
                  const float * data,
                  const float * vec, float * out) {
for( int row = 0; row < num_rows, row++) {
    float temp = 0;
      // row-major order
    for(int n = 0; n < num_cols; n++) {
        int col = indices[num_cols * row + n];
        float value = data[num_cols * row + n];
        if (value != 0 && col != 0)
            temp += value * vec[col];
    }
    out[row] += temp;
}
}
```

Take a few moments to convince yourself that we are indeed using the ELLPACK format to solve SpMV, and the data when stored in the low-level memory, is in row-major order. Putting on your parallel developer hat again, one strategy is to have one thread / work item process one row of the matrix data, and this implies that you can remove the outer loop structure thus giving you this possible SpMV ELL kernel.

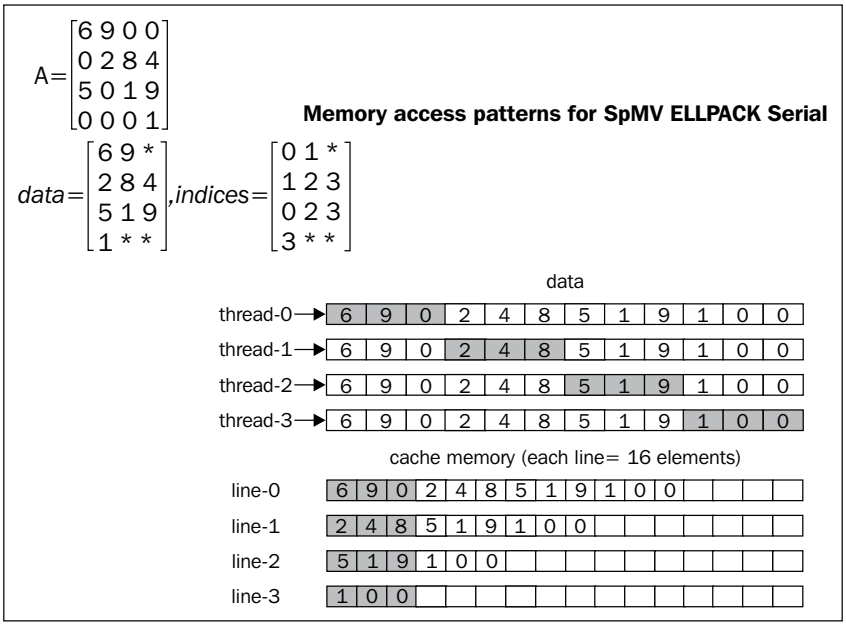
```
// num_rows  - number of rows in matrix
// data      - the array that stores the non-zero values
// indices   - the array that stores the column indices for zero, non-
//              zero values in the matrix
// num_cols  - the number of columns.
// vec       - the dense vector
// y         - the output
__kernel void
```


www.EngineeringBooksPdf.com

```
spmv_ell_gpu(__global const int num_rows,
             __global const int num_cols,
             __global const int * indices;
             __global const float * data,
             __global const float * vec, float * out) {
     int row = get_global_id(0);
     if (row < num_rows) {
    float temp = 0;
       // row-major order
    for(int n = 0; n < num_cols; n++) {
        int col = indices[num_cols * row + n];
        float value = data[num_cols * row + n];
        if (value != 0 && col != 0)
            temp += value * vec[col];
    }
    out[row] += temp;
}
}
```

The first thing you would probably notice is that the outer loop structure has been removed, and that is intuitive when you consider the fact that that structure was present initially so that we can iterate over the inner loop which contains the actual work of the dot product between a row of the matrix and vector.

Now, when we examine its memory access patterns using our strategy of fine-grained parallelism, we would have something like the following representation and it would exhibit similar problems when we look at the SpMV CSR kernel in a later section:



www.EngineeringBooksPdf.com

Understanding how to solve SpMV using the ELLPACK-R format

ELLPACK-R is a variant of the ELLPACK format, and apparently it is rather popular for implementing SpMV on GPUs. ELLPACK-R should be used if no regular substructures such as off-diagonals or dense blocks can be exploited. The basic idea is to compress the rows by shifting all non-zero entries to the left and storing the resulting $M \times N$ matrix column by column consecutively in main host memory, where *N* is the maximum number of non-zero entries per row.

How to do it

The SpMV ELLPACK-R scalar kernel is called scalar because of the fact that we have not taken advantage of a particular aspects unique to GPUs when it comes to parallel program development in OpenCL. This aspect is known as **wavefront-/warp-level programming**. We'll talk more about this in the SpMV CSR kernel presentation in the next section. Hence, in this part we will present our OpenCL kernel, as shown in the following code, that employs the strategy of using one thread to process a row of the matrix data, and this time, we have the help of another array, `rowLengths`, which records the actual length of each row in the matrix where it contains non-zero values:

```
// data - the 1-D array containing non-zero values
// vec - our dense vector
// cols - column indices indicating where non-zero values are
// rowLengths - the maximum length of non-zeros in each row
// dim - dimension of our square matrix
// out - the 1-D array which our output array will be
__kernel void
spmv_ellpackr_kernel(__global const float * restrict data,
                     __global const float * restrict vec
                     __global const int * restrict cols,
                     __global const int * restrict rowLengths,
                     const int dim,
                     __global float * restrict out) {
    int t = get_global_id(0);

    if (t < dim)
    {
        float result = 0.0;
        int max = rowLengths[t];
        for (int i = 0; i < max; i++) {
            int ind = i * dim + t;
            result += data [ind] * vec[cols[ind]];
        }
```


www.EngineeringBooksPdf.com

```
            out[t] = result;
        }
    }
```

Examining the previous code, we noticed that once again we have reduced two `for` loops into one by recognizing the fact that each thread or work item (in OpenCL parlance, if you recall) can perform the work in the inner loop independently.

In the following code we present our kernel that has been "vectorized", we recognized that our SpMV ELLPACK-R kernel could be improved by taking advantage of the hardware's inbuilt feature to run a bunch of threads executing the code and in lock step.

This vectorization will not work if you were to execute it on your OpenCL x86 compliant CPU unless it has the vectorization hardware available to the GPUs.

This is incredibly useful when the occasions call for it, and this situation calls for it. This resulted in our SpMV ELLPACK-R vector kernel shown in the following code. Our strategy is to have a warp processed at each row of the matrix, and we break each row so that data can be processed by the threads in a warp or wavefront:

```
// data – the 1-D array containing non-zero values
// vec – our dense vector
// cols – column indices indicating where non-zero values are
// rowLengths – the maximum length of non-zeros in each row
// dim – dimension of our square matrix
// out – the 1-D array which our output array will be
#define VECTOR_SIZE 32 // NVIDIA = 32, AMD = 64
__kernel void
spmv_ellpackr_vector_kernel(__global const float * restrict val,
                            __global const float * restrict vec,
                            __global const int * restrict cols,
                            __global const int * restrict rowLengths,
                            const int dim,
                            __global float * restrict out) {

    // Thread ID in block
    int t = get_local_id(0);
    // Thread id within warp/wavefront
    int id = t & (VECTOR_SIZE-1);
    // one warp/wavefront per row
    int threadsPerBlock = get_local_size(0) / VECTOR_SIZE;
    int row = (get_group_id(0) * threadsPerBlock) + (t / VECTOR_SIZE);
```


www.EngineeringBooksPdf.com

```
        __local float volatile partialSums[128];

        if (row < dim) {
            float result = 0.0;
            int max = ceil(rowLengths[row]/VECTOR_SIZE);
            // the kernel is vectorized here where simultaneous threads
            // access data in an adjacent fashion, improves memory
            // coalescence and increase device bandwidth
            for (int i = 0; i < max; i ++) {
                int ind = i * (dim * VECTOR_SIZE) + row * VECTOR_SIZE +
    id;
                result += val[ind] * vec[cols[ind]];
            }
            partialSums[t] = sum;
            barrier(CLK_LOCAL_MEM_FENCE);

            // Reduce partial sums
            // Needs to be modified if there is a change in vector length
            if (id < 16) partialSums[t] += partialSums[t +16];
            barrier(CLK_LOCAL_MEM_FENCE);
            if (id <  8) partialSums[t] += partialSums[t + 8];
            barrier(CLK_LOCAL_MEM_FENCE);
            if (id <  4) partialSums[t] += partialSums[t + 4];
            barrier(CLK_LOCAL_MEM_FENCE);
            if (id <  2) partialSums[t] += partialSums[t + 2];
            barrier(CLK_LOCAL_MEM_FENCE);
            if (id <  1) partialSums[t] += partialSums[t + 1];
            barrier(CLK_LOCAL_MEM_FENCE);

            // Write result
            if (tid == 0)
            {
                out[row] = partialSums[tid];
            }

        }
    }
```


www.EngineeringBooksPdf.com

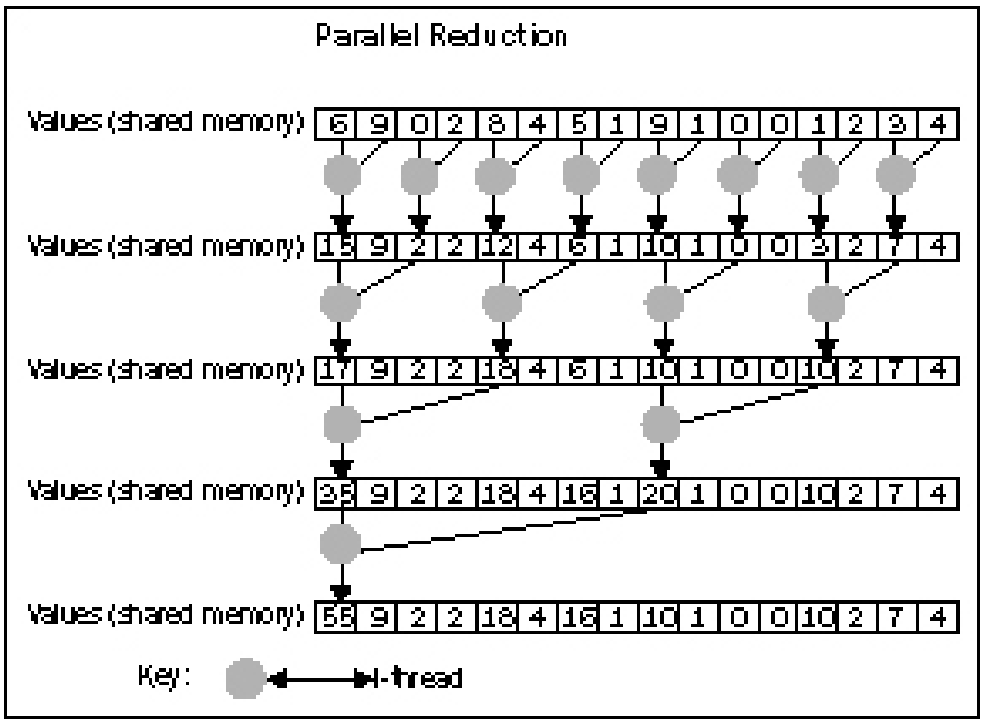
How it works

This vector kernel takes advantage of two facts:

- The kernel is executed by groups of threads and those threads execute in lock step
- **Parallel reduction**: Parallel reduction is rightfully a topic by itself and the variant technique we are using is known as **segmented reduction**

To help you understand how parallel reduction works, let's assume and imagine we have a one-dimensional array filled with 16 elements and each array element is given a number. Now, I like to ask you how you would go about calculating the sum of all elements in this given array? There are definitely more than two ways in which you can do this, but let's say you are giving the fact that eight work items can execute in lock step. how can you take advantage of that?

One way is to have each work item add two array elements and that would give you the partial sums, but how would you be able to add all of these partial sums to produce one single sum that represents the summation of the array? Without going into too much detail, let's use the following diagram and see if you can figure out how it would have worked:



www.EngineeringBooksPdf.com

Understanding how to solve SpMV using the CSR format

After viewing all these different data representations for sparse matrices, you will probably realize there's more to the picture than we earlier imagined, and this serves to highlight the fact that researchers and engineers have spent a lot of time and effort to solve what looks like a deceptively simple problem in an efficient manner. Hence in this section, we are going to take a look at how to solve the SpMV problem using the CSR format looking at various recipes from sequential, scalar, and finally vector kernels in that order.

Getting ready

Now, let us take a look at what SpMV code would look like in its sequential form, that is, when executed on a modern CPU, using the CSR format, and then let's take a look at a naïve implementation of the SpMV:

```
// num_rows – number of rows in matrix
// ptr – the array that stores the offset to the i-th row in ptr[i]
// indices – the array that stores the column indices for non-zero
//           values in the matrix
// x       - the dense vector
// y       - the output
void spmv_csr_cpu(const int num_rows,
                  const int * ptr;
                  const int * indices;
                  const float * data,
                  const float * vec, float * out) {
for( int row = 0; row < num_rows, row++) {
    float temp = 0;
    int start_row = ptr[row];
    int end_row = ptr[row+1];
    for(int jj = start_row; jj < end_row; jj++)
        temp += data[jj] * vec [indices[jj]];
    out[row] += temp;
}
}
```

Examining the preceding code, you will notice that the array `ptr` is being used to pick the non-zero elements in the array—`data`—which is desirable, and `ptr` is also being used to index into the `indices` array to retrieve the correct element in the vector `vec` so that we never conduct operations that multiply a zero value. This point is important to note from a computational point of view because it means we are not wasting precious processor cycles performing work we will never use; from another perspective, this representation also means that the caches are always filled with values we will need and not stored with values that are inherently zero valued.

www.EngineeringBooksPdf.com

As promised, let us take a look at another solution that focuses on matrix-vector multiplication executing on a modern desktop CPU, and in both these examples, the only difference is the fact that the previous code took into account the matrix is sparse while the following code assumes the matrix is dense:

```
// M - the matrix with dimensions 'height' x 'width'
// V - the dense vector of length 'width'
// W - the output
void matvec_cpu(const float* M, const float* V, int width, int height,
float* W)
{
    for (int i = 0; i < height; ++i) {
        double sum = 0;
        for (int j = 0; j < width; ++j) {
            double a = M[i * width + j];
            double b = V[j];
            sum += a * b;
        }
        W[i] = (float)sum;
    }
}
```

Take a few moments and examine both code bases, and you will realize the amount of computational cycles and memory bandwidth that was saved and wasted needlessly.

It is always recommended to compare the sequential form against the parallel form so that you can derive basic metrics about your transformed algorithm.

How to do it

Now that we have made done some basic comparisons, we need to figure out what our parallelization strategy is going to be. For this, we need to put on our parallel developer hat again and scrutinize the code for the SpMV CSR serial kernel shown earlier and look for parallelizable portions. One of the things you might have already recognized is the fact that the dot product between a row of the matrix and the vector `vec`, may be computed independently of all other rows.

www.EngineeringBooksPdf.com

The following code demonstrates the implementation where we have one work item process a row of the matrix, and some literature would call this the scalar kernel. In this kernel, as before, our strategy focuses on looking at the two loop structures, and we discover that the outer loop structure can be flattened out and replaced by work items / threads, and we know how to achieve that; focusing back on the inner loop structure which is essentially what one work item /thread is executing on, we find that we can retain all of its execution flow and mimic that in the OpenCL kernel.

Next, let's take a look at how the SpMV kernel is written with the CSR format in mind:

```
__kernel void
spmv_csr_scalar_kernel( __global const float * restrict val,
                        __global const float * restrict vec,
                        __global const int * restrict cols,
                        __global const int * restrict ptr,
                        const int dim, __global float * restrict out){
    int row = get_global_id(0);

    if (row < dim) {
        float temp=0;
        int start = ptr[row];
        int end = ptr[row+1];
        for (int j = start; j < end; j++) {
            int col = cols[j];
            temp += val[j] * vec[col];
        }
        out[row] = temp;
    }
}
```

If you can recall, in the previous chapter we noted that such an execution model uses really fine-grained parallelism, and such a kernel will probably not perform very well. The issue does not lie within the CSR representation, it lies within the fact that the work items / threads are not accessing those values in the CSR simultaneously. In fact, each thread that was working on each row of the matrix produces a memory access pattern in the following diagram. After tracing the execution of this SpMV CSR kernel for four work items / threads, you will notice that each thread would refer to a different portion of the array `val` (which contains all non-zero entries in the matrix *A*), and memory loads will be latched on the caches (which contain memory banks and memory lanes/lines) and finally the hardware registers will execute upon them.

From this point onwards, you should be thinking in terms of how GPUs work on a low-level basis.

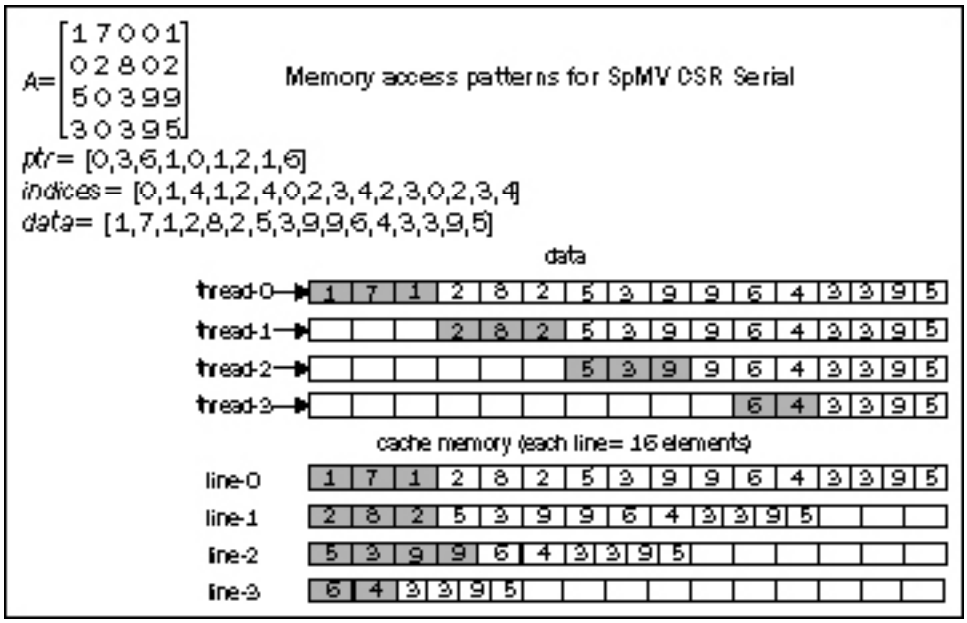
www.EngineeringBooksPdf.com

Let's use the matrix found in the CSR format earlier as an example to illustrate how this SpMV CSR is not really working too well. Each cache is actually implemented by lanes/lines such that each line can hold a number of bytes, and in our example, it assumes each line can hold 16 elements (assuming each element is of the size 4 bytes which translates to 64 bytes).

It should be obvious to you by now that there's a lot of wastage of cache bandwidth. Since our kernel is parallel, we could conceptually have four different lines holding various parts of the input array. What would have been desirable is to allow all the data in at once and keeping the cache hot while processing it.

One way of achieving this is to apply the previous techniques you've learned. Kudos for thinking about that. However, let's learn another technique and in some literature it is known as warp-/wavefront-level programming. We saw it in action in the previous section.

Recall in another chapter, where we introduced the fact that threads of some of the OpenCL devices, GPUs notably execute a bunch of threads in lock step in the processor. The following figure illustrates the memory access pattern for a SpMV CSR kernel when building and executing on a CPU in a serial fashion:



To optimize your algorithm with respect to memory access, have your work items in a single wavefront/warp access the memory locations from the same cache line.

www.EngineeringBooksPdf.com

Next, you would want to ask yourself the question on how you go about working out a kernel that is able to load the elements you need into the same cache line and take advantage of the fact that threads in a warp or wavefront execute in the lock step. This fact also implies that you need coordination, but don't worry, we won't have to use the atomic functions found in OpenCL for this.

When I see the term *lock step*, I immediately conjure the image of 10 runners, akin to executing threads in a warp/wavefront, lined up for a 100 meter dash, and the exception here as compared to the warp-/wavefront-level programming is that all these runners need to reach the finishing line together. Weird, I know, but that's how it works. Coordinating this batch of runners is like strapping leashes on eight horses dragging a wagon and the cowboy driving the carriage using his whip to accelerate or decelerate.

At this point, I like to digress a little and point out to you that **Intel Math Kernel Library** (**Intel MKL**) 11.0 implements sparse solvers using data storage formats based on the CSR formats and has good performance for running on Intel CPUs as they not only optimize memory management but also take advantage of **Instruction Level Parallelism** (**ILP**).

Now, you have to recognize and imagine your kernel to be executed by a bunch of threads and for starters, let's imagine 32 or 64 of them running at once. Each of these threads have an ID and that's the primary method in which you identify and control them, that is, placing the control-flow constructs that allows or restrict threads from running. To illustrate the point, let us take a look at the following improved SpMV CSR vector kernel.

The SpMV CSR OpenCL kernel is found in `Ch8/SpMV/spmv.cl`:

```
#define VECTOR_SIZE 32
// Nvidia is 32 threads per warp, ATI is 64 per wavefront
__kernel void
spmv_csr_vector_kernel(__global const float * restrict val,
                       __global const float * restrict vec,
                       __global const int * restrict cols,
                       __global const int * restrict ptr,
                       const int dim, __global float * restrict out){
    int tid = get_local_id(0);
    int id = tid & (VECTOR_SIZE-1);
    // One row per warp
    int threadsPerBlock = get_local_size(0) / VECTOR_SIZE;
    int row = (get_group_id(0) * threadsPerBlock) + (tid / VECTOR_
SIZE);

    __local volatile float partialSums[128];
    partialSums[t] = 0;
```


www.EngineeringBooksPdf.com

```
        if (row < dim)
        {
            int vecStart = ptr[row];
            int vecEnd   = ptr[row+1];
            float sum = 0;
            for (int j = vecStart + id; j < vecEnd; j += VECTOR_SIZE) {
                int col = cols[j];
                sum += val[j] * vec[col];
            }
            partialSums[tid] = sum;
            barrier(CLK_LOCAL_MEM_FENCE);

            // Reduce partial sums
            // Needs to be modified if there is a change in vector length
            if (id < 16) partialSums[tid] += partialSums[t +16];
            barrier(CLK_LOCAL_MEM_FENCE);
            if (id <  8) partialSums[tid] += partialSums[tid + 8];
            barrier(CLK_LOCAL_MEM_FENCE);
            if (id <  4) partialSums[tid] += partialSums[tid + 4];
            barrier(CLK_LOCAL_MEM_FENCE);
            if (id <  2) partialSums[tid] += partialSums[tid + 2];
            barrier(CLK_LOCAL_MEM_FENCE);
            if (id <  1) partialSums[tid] += partialSums[tid + 1];
            barrier(CLK_LOCAL_MEM_FENCE);

            // Write result
            if (id == 0)
            {
                out[row] = partialSums[tid];
            }
        }
    }
```

Now that we have taken a good look at the OpenCL kernel, we need to build an executable form on which to execute. As before, the compilation will look familiar to you. On my setup with an Intel Core i7 CPU and AMD HD6870x2 GPU running Ubuntu 12.04 LTS, the compilation looks like the following and it'll create an executable called `SpMV` into the working directory:

```
gcc -std=c99 -Wall -DUNIX -g -DDEBUG -arch i386 -o SpMV -framework OpenCL
```


www.EngineeringBooksPdf.com

At this point, the executable should be available to you on the directory. To run the program, simply execute the program `SpMV` in the directory, and you should notice an output that resembles the following:

```
Passed!
```

How it works

The way this works deserves a significant number of explanations, but first of all is the fact that we have adapted our parallel reduction into another form, which is otherwise known as segmented reduction. By this time, you should be relatively familiar with the rest of the code, so I won't walk you through that as you may doze off.

Parallel reduction, in all its forms, is a very effective way to conduct reduction across processors and even architectures. The famous Hadoop framework is an example of parallel reduction across architectures, and the form we are seeing now is that confined to the processor residing on the OpenCL GPU.

Let me walk you through what happened here in our segmented reduction example for the SpMV CSR vector kernel. Initially, we set up a shared memory space in our kernel to hold 128 elements of the type `float`:

```
__local volatile float partialSums[128];
```

You might be curious as to why we need the keyword `volatile` when defining the array `partialSums`. The main reason is because on the level of warp/wavefront-level programming, OpenCL does not have synchronization functions like the memory fences we have encountered so far, and when you do not place the `volatile` keyword when declaring shared memory, the compiler is free to replace the store to and load from `__local` memory with register storage, and execution errors will arise.

The intention was for each thread in the warp/wavefront to store its own computation into its own slot marked by its thread ID.

Next, we see the following bunch of code:

```
if (id < 16) partialSums[tid] += partialSums[t +16];
barrier(CLK_LOCAL_MEM_FENCE);
if (id <  8) partialSums[tid] += partialSums[tid + 8];
barrier(CLK_LOCAL_MEM_FENCE);
if (id <  4) partialSums[tid] += partialSums[tid + 4];
barrier(CLK_LOCAL_MEM_FENCE);
if (id <  2) partialSums[tid] += partialSums[tid + 2];
barrier(CLK_LOCAL_MEM_FENCE);
if (id <  1) partialSums[tid] += partialSums[tid + 1];
```


www.EngineeringBooksPdf.com

```
barrier(CLK_LOCAL_MEM_FENCE);

// Write result
if (id == 0) {
    out[row] = partialSums[tid];
}
```

This code does two things—first is that it only allows threads with certain IDs to execute and the second thing it does is to only allow the thread with ID `0`, that is, zero to write out the total sum into the appropriate element of the output array, `out`.

Let's get into the details. When an executing thread / work item attempts to execute the following piece of code, the kernel will first determine if its ID is allowed, and the threads with IDs ranging from 0 to 15 will get to execute, while those in the following code will not execute, and we will have **thread divergence**:

```
if (id < 16) partialSums[tid] += partialSums[t +16];
barrier(CLK_LOCAL_MEM_FENCE);
```

Recall that thread divergence occurs at branches, that is, `if-then-else`, switches, and so on, which basically partition`s a warp/ wavefront into two, where one part of the group executes code while the other part doesn't.

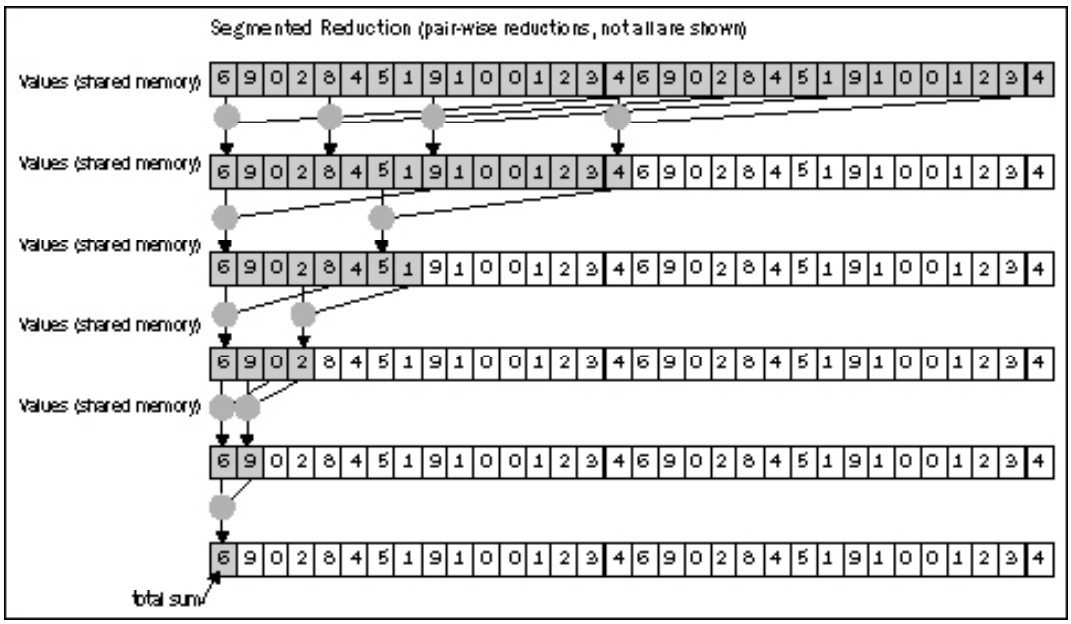
At this point, you should convince yourself that pair-wise reduction takes place for the entire shared-memory array, `partialSums`, and I find it helpful when I trace it on paper or the computer (whatever is your preference). When the executing threads have finished the parallel reduction, notice that there are no overlapping writes (this is intentional), and we need to place a memory fence at that point just to make sure every thread has reached that point before proceeding. This memory fence is important, otherwise bad things will happen. Next, the parallel reduction occurs again, but this time we only need to process half of the array, and we restrict the number of threads to `8`:

```
if (id < 8) partialSums[tid] += partialSums[t +8];
barrier(CLK_LOCAL_MEM_FENCE);
```

We repeat this cycle by dropping the number of executable threads by the power of two till it reaches `1`, and at that point, the final aggregated value will be in the zeroth position in the array, `partialSums`.

www.EngineeringBooksPdf.com

Once we have our final aggregated value in the zero[th] position of the array `partialSums`, we can write it out to its appropriate position in the array `out` indexed by the row we've processed. This segmented reduction is drawn out in the following diagram:



Understanding how to solve SpMV using VexCL

Finally, I would like to present solving the SpMV CSR kernel using the conjugate gradient method. We have studied this method in the beginning of this chapter and hopefully, we still remember what it is. Let me help you by refreshing your memory of the core equations on the CG method:

$$d_{(i)} = r_{(i)} = b - Ax_{(i)},$$
$$\alpha_{(i)} = \frac{r_{(i)}^T r_{(i)}}{d_{(i)}^T A d_{(i)}},$$
$$x_{(i+1)} = x_{(i)} + \alpha_{(i)} d_{(i)},$$
$$r_{(i+1)} = r_{(i)} - a_{(i)} A d_{(i)},$$
$$\beta_{(i+1)} = \frac{r_{(i+1)}^T r_{(i+1)}}{r_{(i)}^T r_{(i)}},$$
$$d_{(i+1)} = r_{(i+1)} + \beta_{(i+1)} d_{(i)}$$

www.EngineeringBooksPdf.com

So far, we have developed a pretty good idea about how to solve SpMV problems using various ways through the SpMV ELLPACK, ELLPACK-R, and CSR formats in both scalar and vector forms, but it took us a while to get there for sure. In this section, you will be introduced to an OpenCL framework for solving problems, and its called VexCL. It can be downloaded from:

- VexCL main page: https://github.com/ddemidov/vexcl
- VexCL Wiki: https://github.com/ddemidov/vexcl/wiki

OpenCL has suffered, in the author's opinion, on the lack of tooling support, and VexCL is again, in the author's opinion, one of the better wrappers around OpenCL C++ and I like to take this section to briefly introduce you to it and you can go download it.

Getting ready

For VexCL to work with you, you will need a C++11 compliant compiler, and GNU GCC 4.6 and the Boost Libs fit the bill. On my setup, I've got the GCC 4.7 compiled with Boost List Version 1.53 without much trouble. That means I won't list the installation instructions as the installation process is relatively straightforward.

How to do it

The following OpenCL kernel is found in `Ch8/SpMV_VexCL/SpMV.cpp`:

```
#define VEXCL_SHOW_KERNELS
// define this macro before VexCL header inclusion to view output
// kernels

#include <vexcl/vexcl.hpp>
typedef double real;
#include <iostream>
#include <vector>
#include <cstdlib>

void gpuConjugateGradient(const std::vector<size_t> &row,
                          const std::vector<size_t> &col,
                          const std::vector<real> &val,
                          const std::vector<real> &rhs,
                          std::vector<real> &x) {
    /*
     Initialize the OpenCL context
     */
    vex::Context oclCtx(vex::Filter::Type(CL_DEVICE_TYPE_GPU) &&
                        vex::Filter::DoublePrecision);
```


www.EngineeringBooksPdf.com

```
    size_t n = x.size();
    vex::SpMat<real> A(oclCtx, n, n, row.data(), col.data(), val.
data());
    vex::vector<real> f(oclCtx, rhs);
    vex::vector<real> u(oclCtx, x);
    vex::vector<real> r(oclCtx, n);
    vex::vector<real> p(oclCtx, n);
    vex::vector<real> q(oclCtx, n);

    vex::Reductor<real,vex::MAX> max(oclCtx);
    vex::Reductor<real,vex::SUM> sum(oclCtx);


    /*
     Solve the equation Au = f with the "conjugate gradient" method
     See http://en.wikipedia.org/wiki/Conjugate_gradient_method
     */
    float rho1, rho2;
    r = f - A * u;

    for(uint iter = 0; max(fabs(r)) > 1e-8 && iter < n; iter++) {
        rho1 = sum(r * r);
        if(iter == 0 ) {
          p = r;
        } else {
          float beta = rho1 / rho2;
          p = r + beta * p;
        }

        q = A * p;

        float alpha = rho1 / sum(p * q);
        u += alpha * p;
        r -= alpha * q;
        rho2 = rho1;
    }

    using namespace vex;
    vex::copy(u, x); // copy the result back out to the host vector
}
```


www.EngineeringBooksPdf.com

How it works

The host code basically fills the one-dimensional arrays with the required values so that they can conform to the CSR format. After this, the device vectors are declared with their appropriate data types and linked with their appropriate host vectors (the copying will take place but it happens behind the scenes), and two reductors are defined (they are basically the reduction kernels we have seen before); the reductor will only execute in the OpenCL device using a single thread of execution, so it isn't quite the same as the parallel reduction we have seen back then; its reduction is alright, but it is carried out in a sequential fashion.

Next, we initialized an ADT known as `SpMAT` which holds the representation of a sparse matrix, and this ADT has the capability to span multiple devices, which is very desirable property since the written code is transparent to its actual underlying computing devices.

In the background, the C++ code you have been shown will cause code generation to occur, and that is the code that will be used, compiled, and executed again; if you like to see the generated kernel code, simply place the C macro `VEXCL_SHOW_KERNELS`. We finally transfer the processed data from the device memory to the host memory using the `copy` function from the `vex` namespace.

www.EngineeringBooksPdf.com

www.EngineeringBooksPdf.com

9

Developing the Bitonic Sort with OpenCL

In this chapter, we will cover the following recipes:

- Understanding sorting networks
- Understanding bitonic sorting
- Developing bitonic sorting in OpenCL

Introduction

Sorting is one of the most important problems in computer science and the ability to sort large amounts of data efficiently is absolutely critical. Sorting algorithms were traditionally been implemented on CPUs and they work very well there, but on the flipside implementing them on GPUs can be challenging. In the OpenCL programming model, we have both task and data parallelism and getting a sorting algorithm to work on the OpenCL model can be challenging, but mostly from the algorithm point of view, that is, how to create an algorithm that takes advantage of the massive data and task parallelism that OpenCL offers.

Sorting methods can largely be categorized into two types: data-driven and data-independent. Data-driven sorting algorithms execute the next step of the algorithm depending on the value of the key under consideration, for example, the QuickSort. Data-independent sorting algorithms is rigid from this perspective because they do not change the order of processing according to the values of the key, so in that sense it doesn't behave like data-driven sorting algorithms. They can be implemented in GPUs to exploit the massive data and task parallelism it offers. Hence we are going to explore the bitonic sort, as it's a classic example of data-independent sorting algorithm and we'll see how it can be represented by sorting networks, and eventually how they can be implemented efficiently in OpenCL to execute on GPUs.

www.EngineeringBooksPdf.com

Ken Batcher invented bitonic sort in 1968. And for n items it would have a size of $O(n\log^2 n)$ and a depth of $O(\log^2 n)$.

The bitonic sort works effectively by comparing two elements at any point in time and what this means is that it consumes two inputs and decides whether a is equal to b, a is less than b, or a is greater than b, that is, the algorithm primarily operates on two elements, given an input. The bitonic sort is an example of a non-adaptive sorting algorithm.

A non-adaptive sorting algorithm is the one where the sequence of operations performed is independent of the order of the data also known as data-independent.

To give you a more concrete idea of what non-adaptive sorting methods are like, let's create a fictitious instruction `cmpxchg`, which has the semantics of comparing two elements and exchanging them when necessary. This is how it would look if we were to implement a compare-swap operation between two elements. In the following example, we illustrate the fact that non-adaptive methods are equivalent to straight line programs for sorting and they can be expressed as a list of compare-exchange operations to be performed.

```
cmpxchg(a[0], a[1]);
cmpxchg(a[1], a[2]);
cmpxchg(a[0], a[1]);
```

For example, the preceding sequence is a straight line program for sorting three elements; and quite often the goal of developing such an algorithm is to define for each *n*, a fixed sequence of the `cmpxchg` operations that can sort any set of *n* keys. To put it in another way, the algorithm doesn't take into account whether the data to be sorted is sorted prior or partially sorted.

Understanding sorting networks

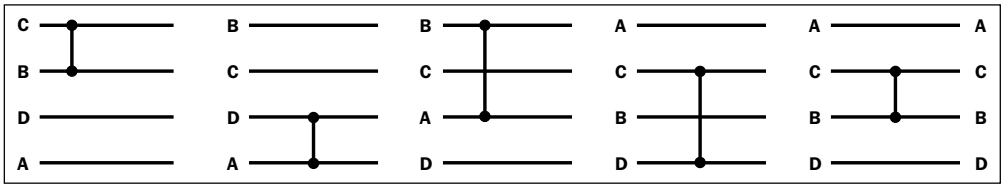
In the previous section, we looked at a non-adaptive sorting algorithm and what it's nature is in its fundamental form. In this section, let's look at a model frequently used to study non-adaptive sorting algorithms. Technical literature has called this model, the sorting network. This form of sorting is also known as comparator networks, and is the idea behind the bitonic sort.

Sorting networks are the simplest model for this study, as they represent an abstract machine which accesses the data only through compare-exchange operations, and it comprises of atomic compare-exchanges also known as comparators which are wired together to implement the capability of general sorting.

www.EngineeringBooksPdf.com

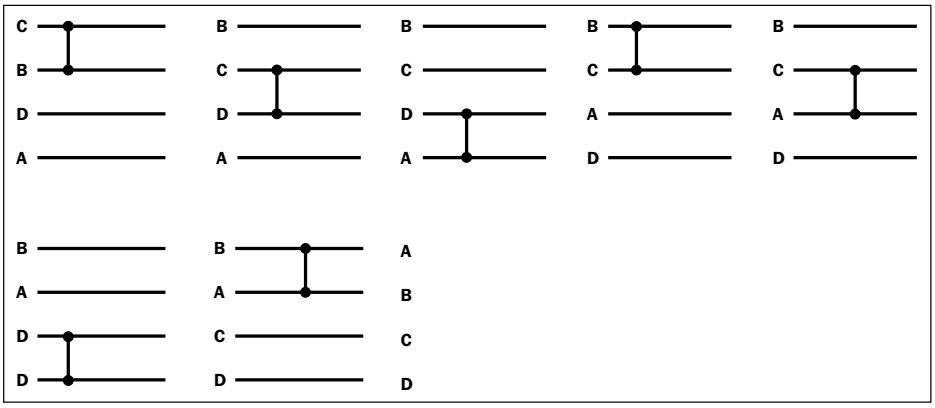
How to do it...

The following is an illustration for sorting four keys. By convention, we draw a sorting network for *n* items as a sequence of *n* horizontal lines, with comparators connecting a pair of lines. We also imagine that the keys to be sorted pass from right to left through the network, with a pair of numbers exchanged if necessary to put the smaller on the top whenever the comparator is encountered:



From the preceding diagram, you will notice that the keys move from left to right on the lines in the network. The comparators that they encounter would exchange the keys if necessary and continually push the smaller key towards the top of this network. An astute reader will notice that no exchanges were done on the fourth comparator. This sorting network will sort any permutation of four keys.

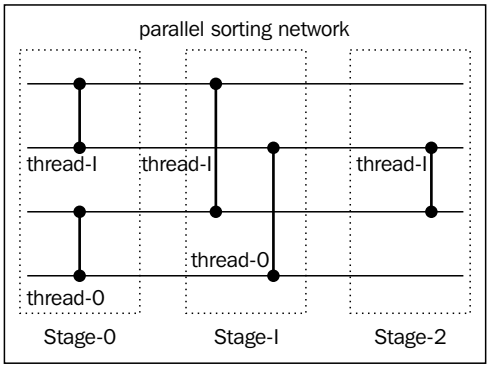
There are other sorting networks other than this and the following network also sorts the same input as before, but it takes two more compare-exchange operations as compared to the previous sorting network. It is interesting to study and that's why this is left as an exercise for you to research on your own.



www.EngineeringBooksPdf.com

How it works...

This sorting network exhibits a particular property and that is as long as the comparators do not overlap, then we can actually conduct the compare-exchange operations in parallel. Next, we need to understand how we can exact parallelism from this by grouping what can be done in parallel and needs to be performed in the next stage. Here's the sorting network that is optimal for sorting any four keys and we show the operations that can be conducted in parallel which are broken into three stages of sorting:



Although it is not the most efficient, the earlier diagram illustrates a possible parallel sorting network for any four keys. In this parallel sorting network, we could potentially launch threads where it will conduct the compare-exchange operations in three stages, and the result is that the input is sorted.

Notice that this sorting network for sorting four keys is optimal from a computational point of view, as it has only to perform five compare-exchange operations in three stages.

Understanding bitonic sorting

Previously we have discussed sorting networks and it closely relates to bitonic sorting, because sorting networks are employed to implement non-adaptive sorting algorithms, for example, bitonic sort. In bitonic sorting, we basically have an input (defined elsewhere) that's a bitonic sequence. A bitonic sequence is one that monotonically increases (decreases), reaches a single maximum (minimum), and then monotonically decreases (increases). A sequence is considered bitonic if it can be made so by cyclically shifting the sequence.

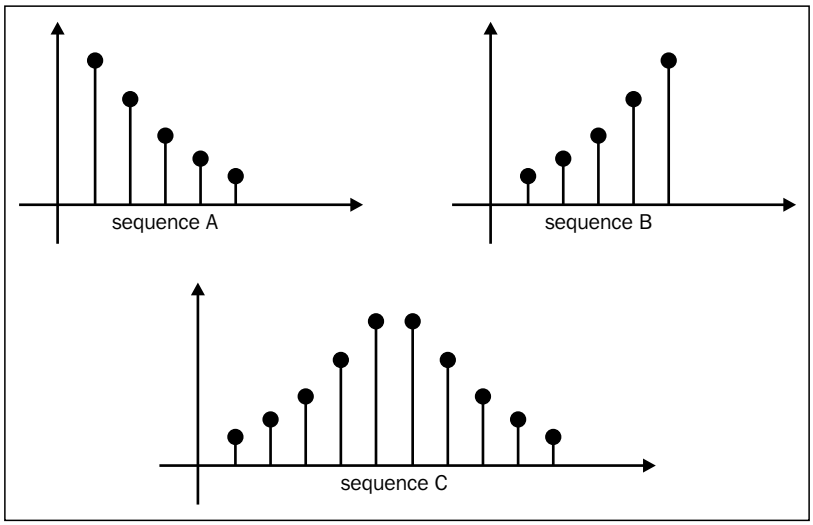
www.EngineeringBooksPdf.com

In general, we consider a few scenarios for determining whether the input is suitable for sorting (after all processor cycles are precious and it is a good idea not to waste them doing needless work). In fact, when we wish to sort some input based on a particular sorting algorithm, we would always consider whether the input is already sorted based on our criteria. In the context of bitonic sorting, we could possibly receive a bitonic sequence, and what we do for that is apply what is known as a bitonic split sequence or an arbitrary sequence, in the case of an operation on the input sequence and keep doing this until we reach the final sorted state.

A bitonic split is an operation on a bitonic sequence, such that if $a_i > a_{i+n/2}$ the two elements are exchanged, $1 \leq i < n$ and the operation produces two bitonic sequences A and B, such that the elements in A are less than the elements in B.

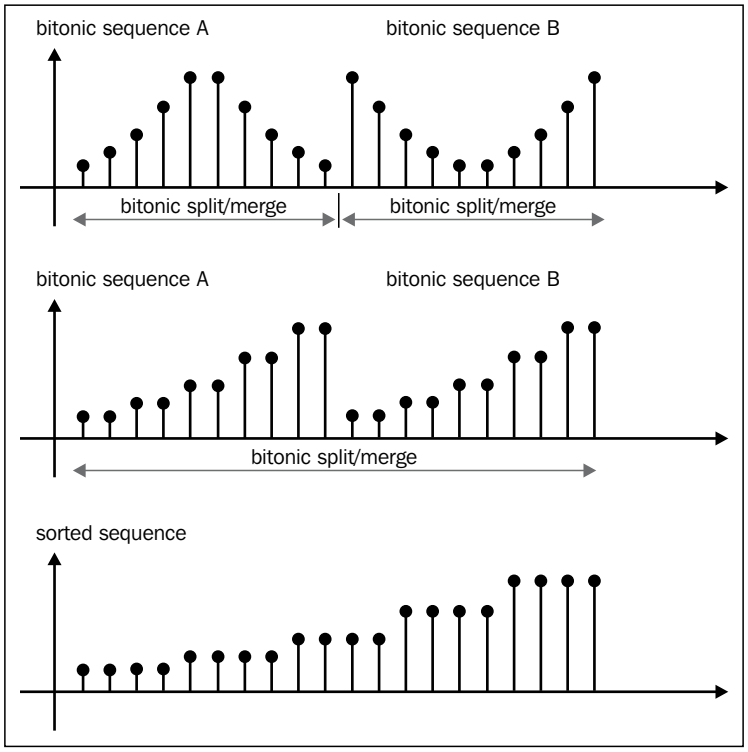
How to do it...

The diagram shows how two bitonic sequences (at the top of the diagram) can be conceptually combined to a larger sequence (at the bottom of the diagram) by repeated application of this sorting algorithm:



www.EngineeringBooksPdf.com

In the situation where we receive an arbitrary sequence, that is, unsorted and not in bitonic order, we have to basically produce a bitonic sequence from this unsorted input and then apply the same trick as before using the bitonic splits until we reach the final sorted state. The following diagram illustrates how a bitonic split or merge (as it's often called) operates on separate sequences and produces the final sorted sequence either in ascending or descending order:



In either case, we will know when to terminate if the split sizes have reached two, because at this point, it's a comparison operation between a and b, where either a is greater than or equal to b or b is greater than or equal to a. And it holds and depending on the sorting order, we will place them into their appropriate position in the output.

Bitonic Sorting uses a principle created by Donald Knuth and it's known as the Knuth's 0/1 principle, which is: If a sorting algorithm that performs only element comparisons and exchanges on all sequences of zeros and ones, and then it sorts all sequences of arbitrary numbers.

Before we proceed to develop the bitonic sort algorithm using OpenCL, it's proper that we only introduce it through its sequential form from which we can begin to look for opportunities for parallelism.

www.EngineeringBooksPdf.com

The following code snippet is from `src/Ch9/BitonicSort_CPU_02/BitonicSort.c` and the relevant portions of the code are shown. This implementation is a translation of Batcher's algorithm, that for illustration purpose is a recursive one and looks like this:

```
void merge(int a[], int l, int r) {
  int i, m = (l+r)/2;
  if (r == (l+1)) compareXchg(a, l, r);
  if (r < (l+2)) return;

  unshuffle(a, l, r);
  merge(a, l, m);
  merge(a,  m+1, r);
  shuffle(a, l, r);
  // In the original algorithm the statement was the following:
  // for(i = l+1; i < r; i+= 2) compareXchg(a, i, i+1);
  for(i = l; i < r; i+= 2) compareXchg(a, i, i+1);
}
```

This recursive program works is by repeatedly splitting its original input by half and it proceeds to sort each of the halves and merges those halves into bigger segments. This process is continued until the segment reaches the original size. Notice that it uses two other supporting functions to accomplish this and they're called `shuffle` and `unshuffle`. They work similarly to the same functions in OpenCL (which isn't a wonder because the same functions in OpenCL drew inspiration from them). Here are those functions:

```
void shuffle(int a[], int l, int r) {
  int* aux = (int*)malloc(sizeof(int) * r);
  int i, j, m = (l+r)/2;
  for(i = l, j = 0; i <= r; i += 2, j++ ) {
    aux[i] = a[l+j];
    aux[i+1] = a[m+1+j];
  }
  for(i = l; i <= r; i++) a[i] = aux[i];
}

void unshuffle(int a[], int l, int r) {
  int* aux = (int*)malloc(sizeof(int) * r);
  int i, j, m = (l+r)/2;
  for(i = l, j = 0; i <= r; i += 2, j++ ) {
    aux[l+j] = a[i];
    aux[m+1+j] = a[i+1];
  }
  for(i = l; i <= r; i++) a[i] = aux[i];
}
void compareXchg(int* arr, int offset1, int offset2) {
```

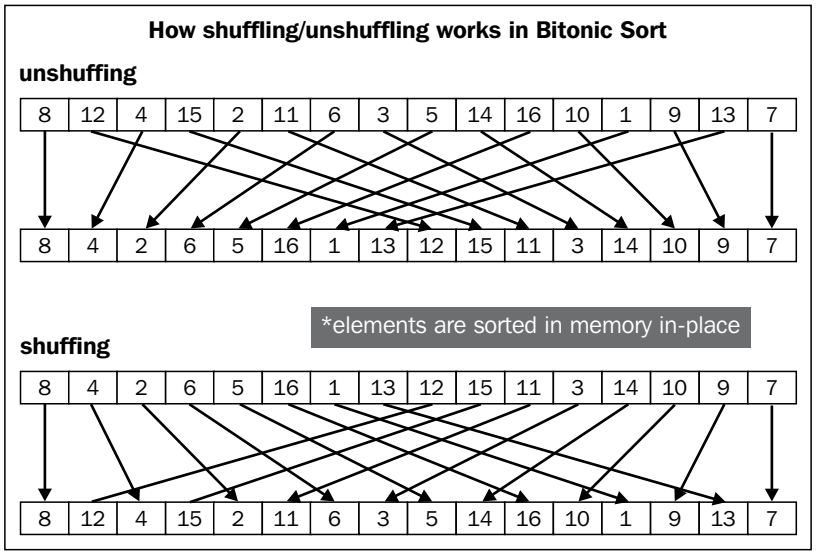

www.EngineeringBooksPdf.com

```
    if (arr[offset1] >= arr[offset2]) {
      int t = arr[offset1];
      arr[offset1] = arr[offset2];
      arr[offset2] = t;
    }
  }
```

And what they do is this: shuffling actually splits the input into halves again and picks each element from each half and place them side-by-side until it reaches the end of both halves. Unshuffling does exactly the opposite by removing those elements and placing them into their original positions and for those algorithm geeks in you, you would recognize that this is the program implementation of the top-down mergesort algorithm and belongs to the class of algorithms that uses the divide-and-conquer approach. As a refresher, an illustration is shown in the *How it works...* section of this recipe, which depicts how both shuffling and un-shuffling works in this algorithm.

How it works...

The concept of shuffling and unshuffling was explored in *Chapter 4*, *Using OpenCL Functions* and we invite you to head back there and refresh yourself with the concepts. The following diagram illustrates how `shuffle` and `unshuffle` (as defined before) would work given an imaginary input: **8**, **12**, **4**, **15**, **2**, **11**, **6**, **3**, **5**, **14**, **16**, **10**, **1**, **9**, **13**, **7**:



www.EngineeringBooksPdf.com

Recursive algorithms similar to the one we have just presented are good for understanding the general flow of the algorithm, but it doesn't work well when you wish to run this algorithm on OpenCL GPUs because recursion isn't fully supported on GPUs. Even though you were to choose an implementation that runs on the CPU via OpenCL, it'll work but it won't be portable.

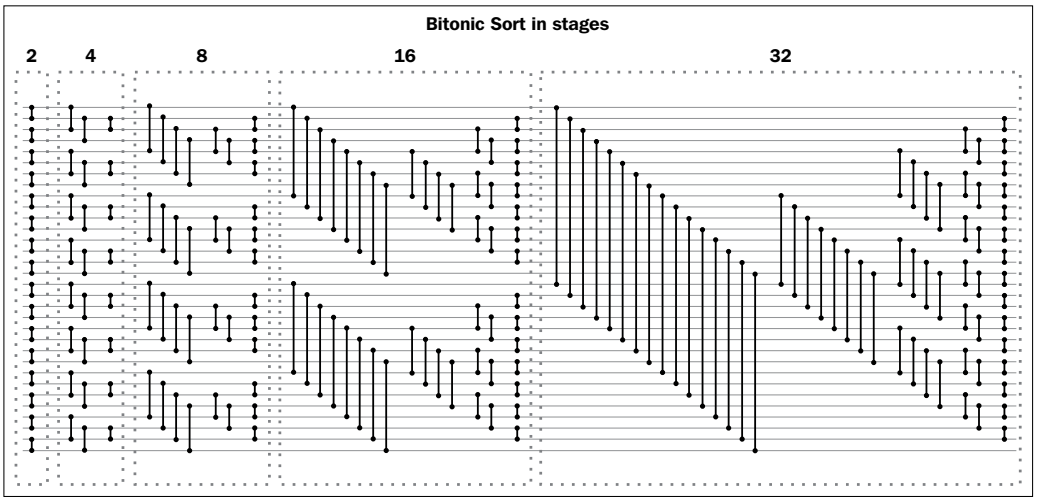
We need an iterative version of this algorithm we just discussed, and fortunately for us we can convert this recursive algorithm to an iterative one. We will look at the following solution from `src/Ch9/BitonicSort_CPU_02/BitonicSort.c`:

```
void merge_iterative(int a[], int l, int r) {
  int i, j , k, p, N = r -l+1;
  for(p = 1; p < N; p += p)
    for(k = p; k > 0; k /= 2)
     for(j = k%p; j+k < N; j += (k+k))
      for(i = 0; i < k; i++)
        if(j+i+k < N)
          if((j+i)/(p+p) == (j+i+k)/(p+p))
            compareXchg(a, l+j+i, l+j+i+k);
}
```

This algorithm is divided into phases indexed by the `p` variable. The last phase, which is when `p` is `N`, and each phase applies the sorting and merging to segments of sizes `N / 2`, `N / 4`, `N / 8` to $\frac{N}{2^k}$. When examining this code deeper by tracing the execution flow, you would notice that it is actually computing the sorting network that accepts 32 inputs (corresponding to the number of inputs in our input buffer), and when you read the diagram from left to right, you will notice that it approaches solving this problem in a bottom-up manner:

www.EngineeringBooksPdf.com

What I meant by bottom-up approach is that figure should be read from left to right (that's also the flow of the data through this sorting network). When you draw columns around the first column, you'll notice that the algorithm creates segments of sizes two. Then the second and third columns form segments of sizes 4, then the fourth, fifth, and sixth columns form segments of size eight. They continue to form to sort/merge segments of sizes that are a power of two up to the point where it sorts and merges all the `N` elements in the input array. You will probably have realized that the algorithm doesn't create any temporary data structures to hold temporary values and it's actually sorting in-place. The immediate consequence of a sorting algorithm that sorts in-place is that it is memory efficient, since the output is written into the input and doesn't create any memory storage at all. The following is an illustration of the partition sizes that the algorithm works on while at every stage:



To develop our understanding of the bitonic sort and sorting networks, it is important to understand how parallelism can be subsequently extracted from.

Developing bitonic sorting in OpenCL

In this section, we will walk through an implementation of sorting an arbitrary input by using the bitonic sort in OpenCL which runs better on a GPU.

We recall that bitonic sorting recursively sorts elements in the input by building up sequences and merging those into bigger sized sequences and then repeats the cycle, and the two key operations performs it really does is to conduct: a pairwise comparison to determine the greater/smaller of the two elements in a sequence, and merging the two sequences by applying the bitonic sort between them.

www.EngineeringBooksPdf.com

Getting ready

So far we have seen how we can apply the bitonic sort to bitonic sequences. The question we need to address next is what do we do with an input that is entirely arbitrary? The answer to that question is to make it into a bitonic sequence and then apply a series of bitonic splits/merge. At the beginning, pairwise compare-exchange operations are conducted for elements in the input, and at the end of this stage we have sorted segments of size two. The next stage is to group two segments of size two and perform compare-exchange producing segments of size four. The cycle repeats itself and the algorithm keeps creating bigger segments of size 2^k.

Recall from the previous section, where we saw the iterative version of the bitonic sort (the algorithm is repeated here) which uses an array index, `p`, to denote the phases in which the sort will take place and with each phase of the algorithm, the algorithm sorts and merges segments of sizes two, four, eight, and so on. And building up on that idea, each phase of the sort is going to be parallel. Also remember that we need to do two things:

- Build a comparator network (bitonic split/sort) that sorts two smaller bitonic sequences into a large one, remembering the fact that sizes are powers of two. This pairwise comparison between two elements will be conducted by a single executing thread/work item.
- Build bitonic sequences on each half, such that one half is monotonically increasing and the other half is monotonically decreasing.

How to do it...

Our strategy focuses on using a single executable thread performing the compare-exchange operation, and following is the Bitonic Sort OpenCL kernel which uses this simple strategy.

The following code excerpt is taken from `Ch9/BitonicSort_GPU/BitonicSort.cl`:

```
__kernel
void bitonicSort(__global uint * data,
                 const uint stage,
                 const uint subStage,
                 const uint direction) {

    uint sortIncreasing = direction;
    uint threadId = get_global_id(0);

    // Determine where to conduct the bitonic split
    // by locating the middle-point of this 1D array
    uint distanceBetweenPairs = 1 << (stage - subStage);
    uint blockWidth   = 2 * distanceBetweenPairs;

    // Determine the left and right indexes to data referencing
    uint leftId = (threadId % distanceBetweenPairs) +
```


www.EngineeringBooksPdf.com

```
                    (threadId / distanceBetweenPairs) * blockWidth;

        uint rightId = leftId + distanceBetweenPairs;

        uint leftElement = data[leftId];
        uint rightElement = data[rightId];

        // Threads are divided into blocks of size
        // 2^sameDirectionBlockWidth
        // and its used to build bitonic subsequences s.t the sorting is
        // monotically increasing on the left and decreasing on the right
        uint sameDirectionBlockWidth = 1 << stage;

        if((threadId/sameDirectionBlockWidth) % 2 == 1)
            sortIncreasing = 1 - sortIncreasing;

        uint greater;
        uint lesser;
        // perform pairwise comparison between two elements and depending
        // whether its to build the bitonic that is monotically increasing
        // and decreasing.
        if(leftElement > rightElement) {
            greater = leftElement;
            lesser  = rightElement;
        } else {
            greater = rightElement;
            lesser  = leftElement;
        }

        if(sortIncreasing) {
            input[leftId]  = lesser;
            input[rightId] = greater;
        } else {
            input[leftId]  = greater;
            input[rightId] = lesser;
        }
    }
```

Using the preceding OpenCL kernel code we need to build an executable, so that it can execute on our platform. As before, the compilation will look familiar to you. On my setup with an Intel Core i7 CPU and AMD HD6870x2 GPU running Ubuntu 12.04 LTS, the compilation looks as follows, and it'll create an executable called `BitonicSort` into the working directory:

```
gcc -std=c99 -Wall -DUNIX -g -DDEBUG -arch i386 -o BitonicSort -framework
OpenCL
```


www.EngineeringBooksPdf.com

At this point, you should have an executable deposited in that directory. All you need to do now is to run the program, simply execute the `BitonicSort` program in the directory and you should have noticed an output that resembles this:

```
Passed!
Execution of the Bitonic Sort took X.Xs
```

How it works...

The algorithm starts from the basic strategy of using a thread to conduct the pairwise comparison-exchange operation. The details is that the host code will break down the original input into its respective phases, and for our testing purposes we have an input of 16 million elements which works out to 24 phases. In the host code, we use the `stage` variable to indicate that. Next at each phase, the algorithm will apply the bitonic split/sort and merge segments of sizes progressively from the least power of two to the greatest power of two, smaller or equal to the phases, for example if we are sorting for elements of size eight, then we would sort to produce segments of size two, then four, and finally we will sort and merge 4-by-4 sequences to get eight.

In detail when the kernel starts executing, it has to start building the bitonic subsequences by using the bitonic split. And to do that the kernel needs to know where to split the array, taking into account the current stage of the sort and it does this with the following code:

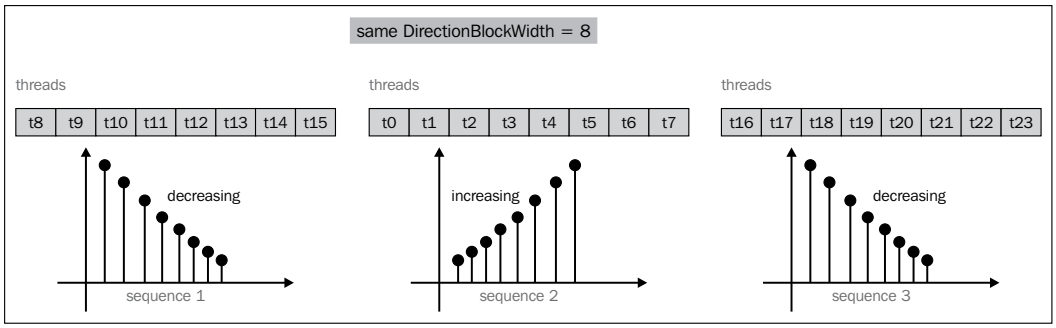
```
uint distanceBetweenPairs = 1 << (stage - subStage);
uint blockWidth   = 2 * distanceBetweenPairs;

// Determine the left and right indexes to data referencing
uint leftId = (threadId % distanceBetweenPairs) +
              (threadId / distanceBetweenPairs) * blockWidth;

uint rightId = leftId + distanceBetweenPairs;
```

Next, the kernel loads the data values from the array by using the `leftId` and `rightId` indices and stores them in the thread's local register memory. The next part of the algorithm is to build bitonic sequences, such that one half is monotonically increasing and the other half is monotonically decreasing. And we use the variable, `sameDirectionBlockWidth`, as a heuristic to guide whether we are going to sort increasingly or decreasingly. The following code does that:

```
uint sameDirectionBlockWidth = 1 << stage;

if((threadId/sameDirectionBlockWidth) % 2 == 1)
    sortIncreasing = 1 - sortIncreasing;
```


www.EngineeringBooksPdf.com

As an example, let's assume that stage is three which implies that `sameDirectionBlockWidth` is eight. The following figure demonstrates what will eventually happen when the `sortIncreasing` variable flips based on the (above) computation, and hence creates the desired effect of bitonic sequencing:



The rest of the kernel code is concerned with the pairwise comparison-exchange operation, which we are familiar with by now.

Another aspect of this implementation is that the algorithm is compute bound and it's executed iteratively on the OpenCL GPU via the CPU, and the kernel is notified of which stage it's at including its substages. This can be accomplished in the host code like this:

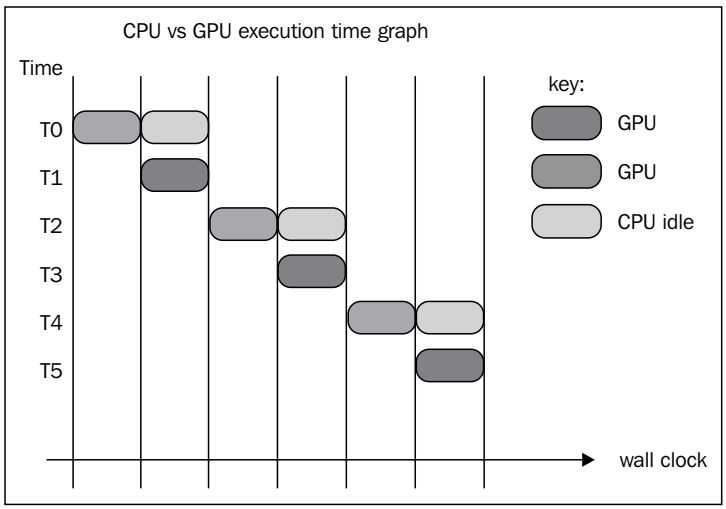
```
for(cl_uint stage = 0; stage < stages; ++stage) {
  clSetKernelArg(kernel, 1, sizeof(cl_uint),(void*)&stage);

  for(cl_uint subStage = 0; subStage < stage +1; subStage++) {
    clSetKernelArg(kernel, 2, sizeof(cl_uint),(void*)&subStage);
                cl_event exeEvt;
                cl_ulong executionStart, executionEnd;
                error = clEnqueueNDRangeKernel(queue,
                                               kernel,
                                               1,
                                               NULL,
                                               globalThreads,
                                               threadsPerGroup,
                                               0,
                                               NULL,
                                               &exeEvt);
                clWaitForEvents(1, &exeEvt);
```


www.EngineeringBooksPdf.com

The code basically iterates over all the stages and its substages, and invokes the GPU to work on the same input buffer notifying the kernel which stage and substage the kernel is executing by invoking `clSetKernelArg` for the appropriate parameter. And then waits until the sorting is done in that phase before starting work on another (this is critical, otherwise the input buffer would be corrupted). In order to make the input buffer be both readable and writeable by the algorithm, it was created like this:

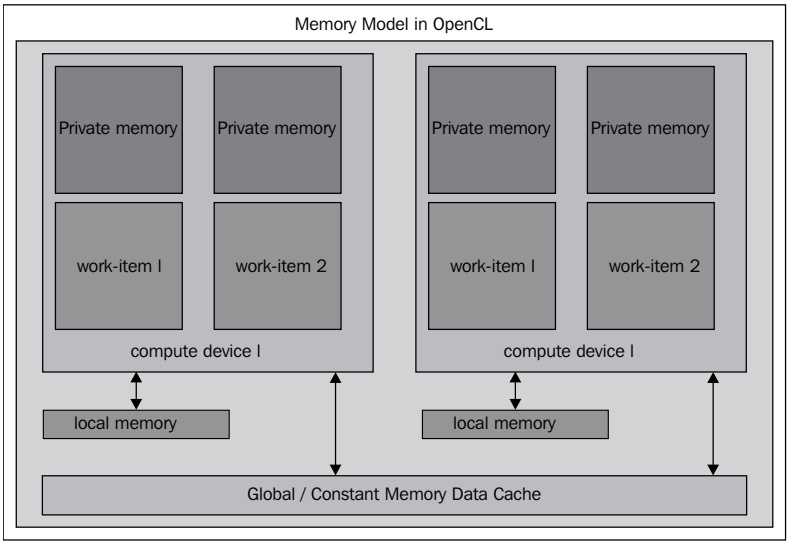
```
device_A_in = clCreateBuffer(context,
                             CL_MEM_READ_WRITE|CL_MEM_COPY_HOST_PTR,
                             LENGTH * sizeof(cl_int),
                             host_A_in,
                             &error);
```

The execution of this algorithm will see the execution flow entering the host, and then leaving for the GPU and continuing to do this until the stages run out. This process is illustrated in the following diagram, though it cannot be scaled:



www.EngineeringBooksPdf.com

We can actually apply an optimization on this kernel by employing a technique we have understood quite well so far, and that is using the shared memory. Shared memory, as you probably know by now, allows the developer to reduce global memory traffic since the program does not have to repeatedly request elements from the global memory space, but instead use what has been stored in its internal memory. Here's a refresher on how the memory model in OpenCL looks like:



Applying the techniques we have learnt so far, we actually have one possible point in which we can apply shared memory techniques by looking out for code that is fetching data from the global memory. We will develop a solution using shared memory and expanding it slightly to have our program load it in strides. We'll get into that in a short while. Let's start at a plausible point for reworking our `bitonicSort` program taking into account the presence of shared memory:

```
uint leftElement = data[leftId];
uint rightElement = data[rightId];
```

We present the following kernel that uses shared memory, we'll explain how it works, found in `Ch9/BitonicSort_GPU/BitonicSort.cl`:

```
__kernel
void bitonicSort_sharedmem(__global uint * data,
                           const uint stage,
                           const uint subStage,
                           const uint direction,
                           __local uint* sharedMem) {
    // more code omitted here
```


www.EngineeringBooksPdf.com

```
// Copy data to shared memory on device
if (threadId == 0) {
    sharedMem[threadId] = data[leftId];
    sharedMem[threadId+1] = data[rightId];
} else {
    sharedMem[threadId+1] = data[leftId];
    sharedMem[threadId+2] = data[rightId];
}
barrier(CLK_LOCAL_MEM_FENCE);

// more code omitted
uint greater;
uint lesser;

if (threadId == 0) {
    if(sharedMem[threadId] > sharedMem[threadId+1]) {
        greater = sharedMem[threadId];
        lesser  = sharedMem[threadId+1];
    } else {
        greater = sharedMem[threadId+1];
        lesser  = sharedMem[threadId];
    }
} else {
    if(sharedMem[threadId+1] > sharedMem[threadId+2]) {
        greater = sharedMem[threadId+1];
        lesser  = sharedMem[threadId+2];
    } else {
        greater = sharedMem[threadId+2];
        lesser  = sharedMem[threadId+1];
    }
}
```

What we did basically was to introduce a variable called `sharedMem` and the strategy for loading those values is simple: each thread will store two values (adjacent) in the shared memory data store, where it will be read out in the subsequent section and all reads which used to refer to the global memory is now conducted in the local/shared memory.

www.EngineeringBooksPdf.com

The host code that is responsible for allocating this memory space is the following code snippet from `Ch9/BitonicSort_GPU/BitonicSort.c` taking into account that each thread writes two adjacent values. And hence it requires twice the amount of memory for a work group of 256 threads:

```
#ifdef USE_SHARED_MEM
clSetKernelArg(kernel, 4, (GROUP_SIZE << 1) *sizeof(cl_uint),NULL);
#endif
```

And to see it in action you can compile the program like this (invoking `gcc` directly):

```
gcc -DUSE_SHARED_MEM -Wall -std=c99 -lOpenCL ./BitonicSort.c -o
BitonicSort_GPU
```

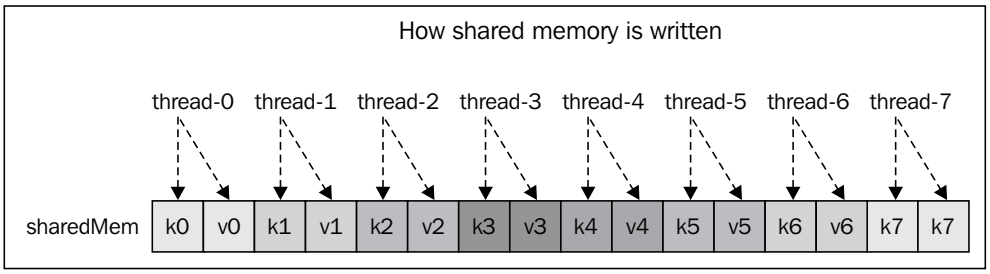
This deposits the `BitonicSort_GPU` program into that directory; another way is to invoke `cmake` at the root of this code base like this:

```
cmake -DUSE_SHARED_MEM=1 -DDEBUG .
```

And navigate to `Ch9/BitonicSort_GPU/` and invoke `make` like this:

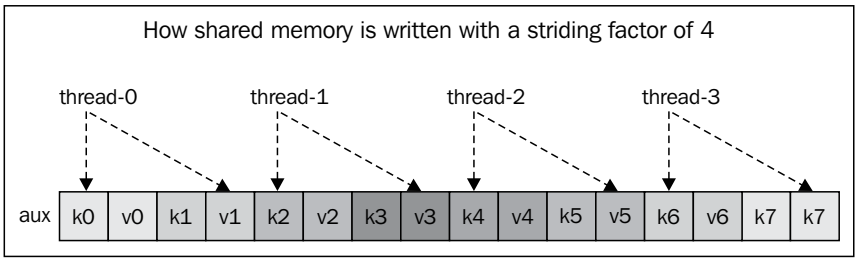
```
make clean;make
```

The following is a diagram of how the writes to the shared memory are done with respect to the scheme we just described. Remember that all subsequent reads is through `sharedMem` instead of the global memory traffic, which means that a significant amount of bandwidth is saved:



We can explore the algorithm a little further by examining the original kernel, `bitonicSort`, where the last part of the algorithm involves essentially a comparison-exchange operation before writing that result back out to global memory. In this situation, we can extrapolate the shared memory concept further by applying it again and our strategy is rather simple here: we have each executing thread writing two pairs, where each pair is this $[(key_{left|rightId}, value_{lesser|greater})]$, and referenced by a key and a value. And in our algorithm the key refers to the output index (that is, `leftId`, `rightId`) and the value refers to the sorted value (that is, `lesser`, `greater`) that will reside at that key. The following diagram illustrates how each thread would have written the two pairs into the `aux` shared memory, and how they could be laid out in memory:

www.EngineeringBooksPdf.com



The following kernel modifications are found at `Ch9/BitonicSort_GPU/BitonicSort.cl` in the kernel named `bitonicSort_sharedmem_2`. We will look at the portions where the changes were different relative to the `bitonicSort_sharedmem` kernel:

```
// Each thread will write the data elements to its own
// partition of the shared storage without conflicts.
const uint stride = 4;
if(sortIncreasing) {
    aux[threadId*stride] = leftId;
    aux[threadId*stride+1] = lesser;
    aux[threadId*stride+2] = rightId;
    aux[threadId*stride+3] = greater;
} else {
    aux[threadId*stride] = leftId;
    aux[threadId*stride+1] = greater;
    aux[threadId*stride+2] = rightId;
    aux[threadId*stride+3] = lesser;
}
barrier(CLK_LOCAL_MEM_FENCE);

if(threadId == 0) {
    for(int i = 0; i < GROUP_SIZE * stride; ++i) {
       data[aux[i*stride]] = aux[i*stride+1];
       data[aux[i*stride+2]] = aux[i*stride+3];
    }
}
```

The final section of the kernel illustrates how we allow only one executing thread, that is, the thread with ID zero, from each work group to conduct the actual write back to global memory from the shared memory, `aux`. Do note that the memory fence is necessary, since the memory in `aux` may not have been filled by the time the thread with ID zero has begun execution. Therefore, it's placed there to ensure memory coherency.

www.EngineeringBooksPdf.com

www.EngineeringBooksPdf.com

10
Developing the Radix Sort with OpenCL

In this chapter, we are going to explore the following recipes:

- Understanding the Radix sort
- Understanding the MSD and LSD Radix sorts
- Understanding reduction
- Developing the Radix sort in OpenCL

Introduction

In the previous chapter, we learned about developing the Bitonic sort using OpenCL. In this chapter, we are going to explore how to develop the Radix sort with OpenCL. Radix sorting is also known as **bucket sorting**, and we'll see why later on.

The first Radix sort algorithms came from a machine called the **Hollerith machine** that was used in 1890 to tabulate the United States census, and though it may not be quite as famous as the machine created by *Charles Babbage*, it does have its place in computing history.

www.EngineeringBooksPdf.com

Understanding the Radix sort

The Radix sort is not a comparison-based sorting algorithm, and it has a few qualities that make it more suitable to parallel computation, especially on vector processors such as GPU and modern CPUs.

I am somewhat reluctant to use the term *modern* since processor technology has evolved so quickly over time that the use of this word somehow seems dated.

The way the Radix sort works is rather interesting when you compare it with the comparison-based sorting algorithms such as quicksort; the main difference between them is how they process the keys of the input data. The Radix sort does this by breaking down a key into smaller sequences of sub-keys, if you will, and sorts these sub-keys one by one.

Numbers can be translated in binary and can be viewed as a sequence of bits; the same analogy can be drawn from strings where they are sequences of characters. The Radix sort, when applied to such keys, does not compare the individual keys, but rather it works on processing and comparing pieces of those keys.

Radix sort algorithms treat the keys like numbers in a base-R number system. *R* is known as the radix, hence the given name of this algorithm. Different values of *R* can be applied to different types of sorting. Examples could be:

- *R = 256* would be sorting strings where each character is an 8-bit ASCII value
- *R = 65536* would be sorting Unicode strings where each character is a 16-bit Unicode value
- *R = 2* would be sorting binary numbers

How to do it...

At this point, let's examine an example to see how the Radix sort would sort the numbers 44565, 23441, 16482, 98789, and 56732, assuming that each number is a five-digit number laid out in memory in contiguous locations

44565	23441	16482	98789	56732

We are going to extract each digit in a right-to-left fashion examining the least significant digit first. Therefore, we have the following:

5	1	2	9	2

www.EngineeringBooksPdf.com

Let's assume we apply counting sort to this array of numbers and it becomes the following:

1	2	2	5	9

This translates to the following order. Take note that the sorting is stable:

23441	16482	56732	44565	98789

Next, we shift to the left by one digit. Notice that now the array of numbers is:

4	8	3	6	8

Applying the counting sort again and translating it back to the order of the numbers, we have:

56732	23441	16482	98789	44565

For the 1000th digit we have:

23441	16482	56732	98789	44565

For the 10,000th digit we have:

23441	44565	16482	56732	98789

For the 100,000th digit we have:

16482	23441	44565	56732	98789

Voila! Radix sorting sorted the array of five-digit numbers. We should note that the sort is *stable*.

www.EngineeringBooksPdf.com

Stable sorting refers to the capability of the algorithm to be able to maintain the relative order between any two elements with equal keys. Let us assume that an array, `int a[5]`, of the values `1`, `2`, `3`, `4`, `9`, and `2`, through some sorting algorithm, X, will sort the elements to `1`, `2`, `2`, `3`, `4`, and `9`. The point here is that the two equal values we saw, which are both the number `2`, occur at positions `1` and `5` (assuming arrays are zero indexed). Then, through X, the sorted list will be such that `a[1]` is always before `a[5]`.

There are actually two basic approaches to Radix sorting. We have seen one approach in which we examine the least-significant digit and sort it. This is commonly referred to as **LSD Radix sorting** since we work our way from right to left. The other approach would be to work from left to right.

The key consideration in Radix sorting is the concept of the *key*. Depending on the context, a key may be a word or a string, and each of them would be of fixed length or variable length.

Understanding the MSD and LSD Radix sorts

Let us take some time to understand how the MSD Radix sort and the LSD Radix sort work before we start working on developing the equivalent on OpenCL.

How to do it...

The Radix sort assumes that we wish to sort Radix-R numbers by considering the most significant digit first. For this to happen, we can partition the input into *R* rather than just two, and we have actually seen this done before. This is data binning, but it extends that with the counting sort. A Radix sort can be run on ASCII characters, Unicode characters, integer numbers (32-bit / 64-bit), or floating-point numbers (sorting floating-point numbers is tricky). You need to figure out what constitutes a key. Keys can be thought of as 8-bit keys, 16-bit keys, and so on, and we know by now that Radix sorts require repeated iterations to extract the keys and sort and bin them based on base *R*.

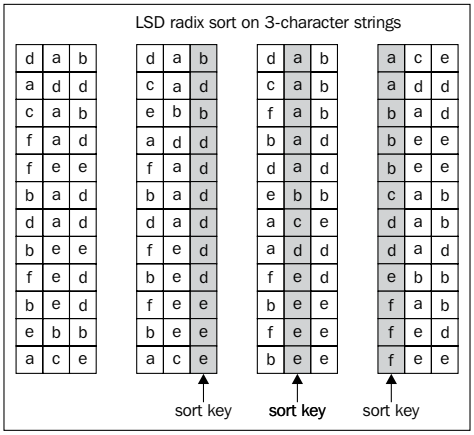
In the following code snippet, we have an MSD Radix sort that sorts the characters in a given string in the programming language C, and the radix we use is 256 (the maximum value of an unsigned 8-bit number, otherwise a signed 8-bit would be -128 to 127):

```
#define N // integers to be sorted with values from 0 – 256
void MSD(char[] s) {
  msd_sort(s, 0, len(s), 0);
}
void msd_sort(char[][] s, int lhs, int rhs, int d) {
  if (rhs <= lhs + 1) return;
  int* count = (int*)malloc(257 *sizeof(int));
```


www.EngineeringBooksPdf.com

```
  for(int i = 0; i < N; ++i)
    count[s[i][d]+1]++;
  for(int k = 1; k < 256; ++k)
    count[k] += count[k-1];
  for(int j = 0; j < N; ++j)
    temp[count[s[i][d]]++] = a[i];
  for(int i = 0; i < N; ++i)
    s[i] = temp[i];
  for(int i = 0; i<255;++i)
    msd_sort(s, 1 + count[i], 1 + count[i+1], d+1);
}
```

The second approach in Radix sorting scans the input from right to left and examines each element by applying a similar operation as in an MSD Radix sort. This is known as the **Least Significant Digit** (**LSD) Radix sort**. LSD Radix sorting works because when any two elements differ, the sorting will place them in the proper relative order, and even when these two elements differ, the fact that LSD exhibits stable sorting means that their relative order is still maintained. Let's take a look at how it would work for sorting three character strings:



A typical LSD Radix sort for sorting characters in a given string might look like the following code (assuming all keys have a fixed width; let's call it `W`):

```
void lsd_sort(char[][] a) {
  int N = len(a);
  int W = len(a[0]);
  for(int d = W - 1; d >= 0; d--) {
    int[] count = (int*) malloc(sizeof(int) * 256);
    for(int i = 0; i< N; ++i)
      count[a[i][d]+1]++;
    for(int k = 1; k < 256; k++)
      count[k] += count[k-1];
    for(int i = 0; i< N; ++i)
```


www.EngineeringBooksPdf.com

```
        temp[count[a[i][d]]++] = a[i];
      for(int i= 0; i< N; ++i)
        a[i] = temp[i];
    }
  }
```

How it works...

Both approaches are similar as they both bin the characters into *R* bins, that is, 256 bins, and they also use the idea of the counting sort to work out where the final sorting arrangement is going to be using a temporary storage, `temp`, and then use that temporary storage and move the data to their sorted places. The nice thing about MSD over LSD Radix sorts is that MSD may not examine all of the keys and works for variable-length keys; although, in that lies another problem—MSD can experience sub-linear sorts; in practice LSD is generally preferred when the size of the key is fixed.

The runtime of an LSD Radix sort is $O(n)$ when compared to the runtimes of other sorting algorithms that are based on the divide-conquer approach, which generally have a runtime of $O(n \log_2 n)$ you might be tempted to conclude that Radix sorting would be faster than comparison-based sorts like quicksort, and you could be right. But, in practice, a well-tuned quicksort can outperform a Radix sort by 24 percent by applying more advanced techniques to improve cache friendliness during the execution. However, technology is constantly evolving, and researchers and engineers will find opportunities to maximize the performance.

You may wish to read the papers *The influence of cache on sorting* by *LaMarca* and *Adapting Radix Sort to the memory hierarchy* by *Rahman and Raman* for more algorithmic improvements that they have worked on.

www.EngineeringBooksPdf.com

Understanding reduction

Radix sorting employs two techniques: **reduction** and **scan**. These are classified as data collection patterns as they occur frequently in parallel computing. This recipe will focus on reduction, which allows data to be condensed to a single element using associative binary operators. The scan pattern can be easily mistaken for the reduction pattern and the key difference is that this pattern reduces every subsequence of a collection up to every position in the input. We'll defer the discussion of scans until we get to the next section.

In the reduction pattern, we typically have an associative binary operator, $f(a,b)=(a \otimes b)$ that we use to collate all elements in a container in a pair-wise fashion. The fact that we need an associative binary operator is an important one, because it implies that the developer can reorganize the combination function to check if it performs efficiently; we'll go into that a little later. Let's take a look at a serial algorithm for conducting reduction in the following code snippet:

```
template<typename T>
T reduce(T (*f)(T, T),
         size_t n,
         T a[],
         T identity) {
  T accumulator = identity;
  for(size_t i = 0; i < n ; ++i)
        accumulator = f(accumulator, a[i]);
  return accumulator;
}
```

The algorithm basically takes an associative binary operator, `f` (that is, a pointer to a function), and an array `a`, of length `n` and computes the operation $(((identify \otimes a_0) \otimes a_1) \otimes a_2)....\otimes a_{n-1}$ over the array with an initial value identified by `identity`.

An associative binary operator can allow the developer to extract parallelism from it because associativity means that the operator would produce the same result regardless of the order in which it is applied to the elements. That is to say:

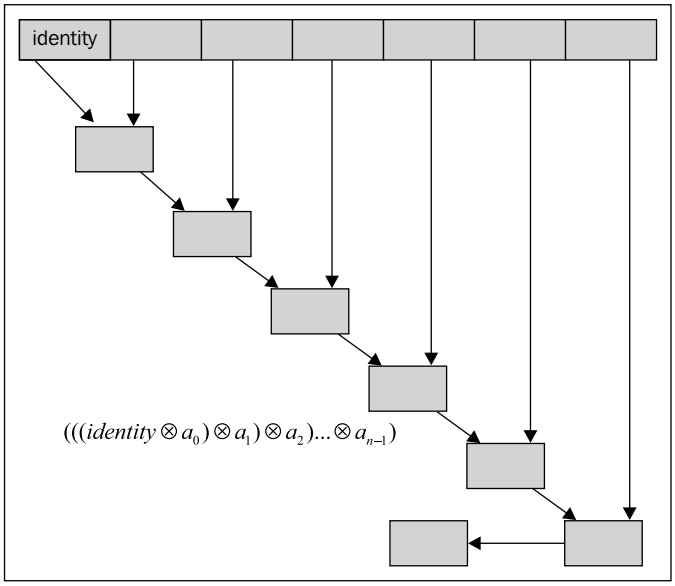
$$((((identify \otimes a_0) \otimes a_1) \otimes a_2)....\otimes a_{n-1})$$

The previous expression is equivalent to:

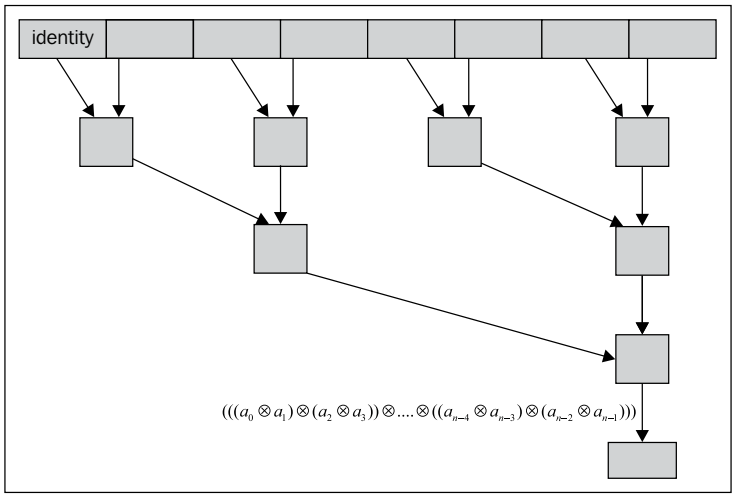
$$(((a_0 \otimes a_1) \otimes (a_2 \otimes a_3)) \otimes .... \otimes ((a_{n-4} \otimes a_{n-3}) \otimes (a_{n-2} \otimes a_{n-1})))$$

www.EngineeringBooksPdf.com

Putting on the many core hat, we can actually imagine a tree of computations in which the sub-trees represent the computation of the form $((a_0 \otimes a_1) \otimes (a_2 \otimes a_3))$. The first sweep would compute the result of this sub-tree while the second sweep would collate the results of the other sub-trees. This will be evident once you have had a chance to examine them visually in the next two diagrams:

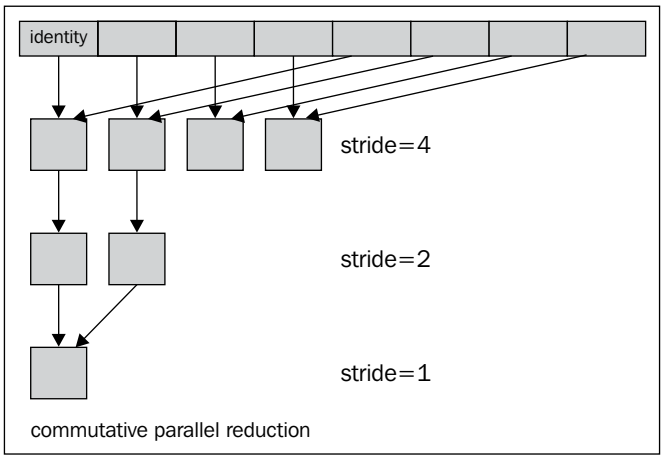


It will be very useful for you to contrast the manner in which these diagrams differ. One of the ways is that the former implies a sequence of operations in traversal order, and this is very different from the latter (as shown in the following diagram):



www.EngineeringBooksPdf.com

It's great news to know that associative operators allow the reduction to be parallelized, but it's not the entire story, because associativity only allows us to group the operations and does not reveal to us whether these groups of binary operations need to occur in a specific order. If you are wondering whether we are talking about commutativity, you are spot on! Commutativity gives us the important property of changing the order of application. We know that some operations exhibit one of these while others exhibit both; for example, we know that addition and multiplication of numbers is both associative and commutative. The following is what a commutative parallel reduction might look like:



Now, seeing this information, you might wonder how this can be translated into OpenCL. We are going to demonstrate a few reductions kernels in this recipe where each one will provide you with an improvement over the previous one.

How to do it...

For this recipe, we are going to assume that we have a large array of a few million elements and that we like to apply the reduction algorithm to compute the sum of all elements. The first thing to do is produce a parallel algorithm for the serial version we saw earlier. All the kernels we are demonstrating are in `Ch10/Reduction/reduction.cl`.

In the serial version of the algorithm, you would have noticed that we simply pass the accumulator into the binary function to perform the operation. However, we cannot use this method in the GPU since it cannot support tens of thousands of executing threads and also the device can contain many more processors than an x86 CPU has. The only solution is to partition the data across the processors so that each block processes a portion of the input, and when all of the processors are executing in parallel, we should expect the work to be completed in a short span of time.

www.EngineeringBooksPdf.com

Assuming that a block has computed its summed value, we still need a way to collate all those partial sums from all blocks, and considering that OpenCL does not have a global synchronization primitive or API, we have two options: have OpenCL collate the partial sums or have the host code collate the partial sums; for our examples, the second option is chosen.

The first kernel, `reduce0`, is a direct translation of the serial algorithm:

```
__kernel void reduce0(__global uint* input,
                      __global uint* output,
                      __local uint* sdata) {
    unsigned int tid = get_local_id(0);
    unsigned int bid = get_group_id(0);
    unsigned int gid = get_global_id(0);
    unsigned int blockSize = get_local_size(0);

    sdata[tid] = input[gid];

    barrier(CLK_LOCAL_MEM_FENCE);
    for(unsigned int s = 1; s < BLOCK_SIZE; s <<= 1) {
        // This has a slight problem, the %-operator is rather slow
        // and causes divergence within the wavefront as not all
threads
        // within the wavefront is executing.
        if(tid % (2*s) == 0)
        {
            sdata[tid] += sdata[tid + s];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    // write result for this block to global mem
    if(tid == 0) output[bid] = sdata[0];
}
```

How it works...

This kernel block would load the elements to its shared memory, `sdata`, and we conduct the reduction in `sdata` in various stages governed by the `for` loop, allowing work items with IDs that are multiples of two to perform the pair-wise reduction. Therefore, in the first iteration of the loop, work items with IDs *{0, 2, 4, 6, 8, 10, 12, 14, ..., 254}* would execute, in the second iteration, only work items with IDs *{0, 4, 8, 12, 252}* would execute, and so on. Following the reduction algorithm, the partial sum would be deposited into `sdata[0]`, and finally this value would be copied out by one thread which happens to have an ID value equal to `0`. Admittedly, this kernel is pretty good but it suffers from two problems: the modulus operator takes a longer time to execute and wavefronts are diverged. The larger issue here is the problem of

www.EngineeringBooksPdf.com

wavefront divergence since it means that some work items in the wavefronts are executing while some are not, and in this case, the work items with odd IDs are not executing while those with even IDs are, GPUs deal with this problem by implementing predication, and this means that all work items in the following code snippet actually get executed. However, the predication unit on the GPU will apply a mask so that only those work items whose IDs matched the condition, `if(tid % (2*s) == 0)`, will execute the statement in the `if` statement, while those work items who fail the condition, `false`, would invalidate their results. Obviously, this is a waste of computing resources:

```
if(tid % (2*s) == 0)
{
    sdata[tid] += sdata[tid + s];
}
```

Fortunately, this can be solved with little effort, and the next kernel code demonstrates this:

```
__kernel void reduce1(__global uint* input,
                      __global uint* output,
                      __local uint* sdata) {
    unsigned int tid = get_local_id(0);
    unsigned int bid = get_group_id(0);
    unsigned int gid = get_global_id(0);
    unsigned int blockSize = get_local_size(0);

    sdata[tid] = input[gid];

    barrier(CLK_LOCAL_MEM_FENCE);
    for(unsigned int s = 1; s < BLOCK_SIZE; s <<= 1) {
        int index = 2 * s * tid;
        if(index < BLOCK_SIZE)
        {
            sdata[index] += sdata[index + s];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    // write result for this block to global mem
    if(tid == 0) output[bid] = sdata[0];
}
```

We replaced the conditional evaluation after the modulus operator has been applied to something more palatable. The appetizing portion is the fact that we no longer have diverging wavefronts, and we have also made strided accesses to the shared memory.

www.EngineeringBooksPdf.com

There's more...

So far, we have seen how we can apply our understanding of associativity to build the reduction kernel and also how to make use of our new understanding of commutativity in the reduction process. The commutative reduction tree is actually better than the associative reduction tree because it makes better use of the shared memory by compacting the reduced values and hence raising efficiency; the following kernel, `reduce2`, reflects this:

```
__kernel void reduce2(__global uint* input,
                      __global uint* output,
                      __local uint* sdata) {
    unsigned int tid = get_local_id(0);
    unsigned int bid = get_group_id(0);
    unsigned int gid = get_global_id(0);
    unsigned int blockSize = get_local_size(0);

    sdata[tid] = input[gid];

    barrier(CLK_LOCAL_MEM_FENCE);
    for(unsigned int s = BLOCK_SIZE/2; s > 0 ; s >>= 1) {
        // Notice that half of threads are already idle on first
iteration
        // and with each iteration, its halved again. Work efficiency
isn't very good
        // now
        if(tid < s)
        {
            sdata[tid] += sdata[tid + s];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    // write result for this block to global mem
    if(tid == 0) output[bid] = sdata[0];
}
```

However, this isn't very good because now during the first iteration, we have already made half of those work items idle and efficiency is definitely affected. Fortunately, however, the remedy is simple. We reduce half the number of blocks and during the hydration of the shared memory, we load two elements and store the sum of these two elements instead of just loading values from global memory and storing them into shared memory. The kernel, `reduce3`, reflects this:

```
__kernel void reduce3(__global uint* input,
                      __global uint* output,
```


www.EngineeringBooksPdf.com

```
                        __local uint* sdata) {
    unsigned int tid = get_local_id(0);
    unsigned int bid = get_group_id(0);
    unsigned int gid = get_global_id(0);

    // To mitigate the problem of idling threads in 'reduce2' kernel,
    // we can halve the number of blocks while each work-item loads
    // two elements instead of one into shared memory
    unsigned int index = bid*(BLOCK_SIZE*2) + tid;
    sdata[tid] = input[index] + input[index+BLOCK_SIZE];

    barrier(CLK_LOCAL_MEM_FENCE);
    for(unsigned int s = BLOCK_SIZE/2; s > 0 ; s >>= 1) {
        // Notice that half of threads are already idle on first
iteration
        // and with each iteration, its halved again. Work efficiency
isn't very good
        // now
        if(tid < s)
        {
            sdata[tid] += sdata[tid + s];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    // write result for this block to global mem
    if(tid == 0) output[bid] = sdata[0];
}
```

Now, things are starting to look much better and we've used what we call **reversed loop** (which is basically counting backwards) to get rid of the problem of divergent wavefronts; in the meantime, we have also not reduced our capacity to reduce elements because we've performed that while hydrating the shared memory. The question is whether there's more we can do? Actually, there is another idea we can qualify and that is to take advantage of atomicity of wavefronts or warps executing on GPUs. The next kernel, `reduce4`, demonstrates how we utilized wavefront programming to reduce blocks atomically:

```
__kernel void reduce4(__global uint* input,
                        __global uint* output,
                        __local uint* sdata) {
    unsigned int tid = get_local_id(0);
    unsigned int bid = get_group_id(0);
    unsigned int gid = get_global_id(0);
    unsigned int blockSize = get_local_size(0);
```


www.EngineeringBooksPdf.com

```
        unsigned int index = bid*(BLOCK_SIZE*2) + tid;
        sdata[tid] = input[index] + input[index+BLOCK_SIZE];

        barrier(CLK_LOCAL_MEM_FENCE);
        for(unsigned int s = BLOCK_SIZE/2; s > 64 ; s >>= 1) {
            // Unrolling the last wavefront and we cut 7 iterations of
    this
            // for-loop while we practice wavefront-programming
            if(tid < s)
            {
                sdata[tid] += sdata[tid + s];
            }
            barrier(CLK_LOCAL_MEM_FENCE);
        }

        if (tid < 64) {
            if (blockSize >= 128) sdata[tid] += sdata[tid + 64];
            if (blockSize >=  64) sdata[tid] += sdata[tid + 32];
            if (blockSize >=  32) sdata[tid] += sdata[tid + 16];
            if (blockSize >=  16) sdata[tid] += sdata[tid +  8];
            if (blockSize >=   8) sdata[tid] += sdata[tid +  4];
            if (blockSize >=   4) sdata[tid] += sdata[tid +  2];
            if (blockSize >=   2) sdata[tid] += sdata[tid +  1];
        }
        // write result for this block to global mem
        if(tid == 0) output[bid] = sdata[0];
    }
```

In the code block demarcated by the statement `if (tid < 64)`, we no longer need to place the memory barriers because the code block only hosts one wavefront which executes atomically in the lock step.

Developing the Radix sort in OpenCL

From this section onwards, we are going to develop this sorting method for OpenCL. We are going to do two things: implement the parallel Radix sort described in the paper that *Marco Zagha* and *Guy E. Blelloch* wrote in 1991 titled *Radix Sort for Vector Multiprocessors*. The former algorithm was crafted for the CRAY Y-MP computer (which, in turn, was adapted from the parallel Radix sort algorithm that worked on the **Connection Machine (CM-2)**).

www.EngineeringBooksPdf.com

Getting ready

Radix sorting attempts to treat keys as multi-digit numbers, where each digit is an integer depending on the size of the Radix, *R*. An example would be sorting a large array of 32-bit numbers. We can see that each such number is made up of four bytes (each byte is 8-bits on today's CPU and GPU processors), and if we decide to assume that each digit would be 8-bits, we naturally would treat a 32-bit number as comprised of four digits. This notion is most natural when you apply the concept back to a string of words, treating each word as comprising of more than one character.

The original algorithm worded in the 1999 paper basically uses the counting sort algorithm and it has three main components which will in turn sort the input by iterating all three components until the job is done. The pseudo code, which is a serial algorithm, is presented as follows:

```
COUNTING-SORT
  HISTOGRAM-KEYS
    do i = 0 to 2^r -1
      Bucket[i] = 0
    do i = 0 to N - 1
      Bucket[D[j]] = Bucket[D[j]] + 1
  SCAN-BUCKETS
    Sum = 0
    do i = 0 to 2^r - 1
      Val = Bucket[i]
      Bucket[i] = Sum
      Sum = Sum + Val
  RANK-AND-PERMUTE
    do j = 0 to N - 1
      A = Bucket[D[j]]
      R[A] = K[j]
      Bucket[D[j]] = A + 1
```

The algorithm `HISTOGRAM-KEYS` is something that we have already encountered a few chapters ago, and it is really the histogram. This algorithm computes the distribution of the keys that it encounters during the sort. This algorithm is expressed in a serial fashion, that is, it is supposed to run on a single executing thread; we have already learned how to parallelize that and you can apply those techniques here. However, what we are going to do now deviates from what you have seen in that previous chapter, and we'll reveal that soon enough.

The next algorithm is `SCAN-BUCKETS`, and it is named as such because it actually scans the entire histogram to compute the prefix sums (we'll examine prefix sums in fair detail later). In this scan operation, `Bucket[i]` contains the number of digits with a value, `j`, such that `j` is greater than `i`, and this value is also the position, that is, the array index in the output.

www.EngineeringBooksPdf.com

The final algorithm is `RANK-AND-PERMUTE`, and each key with a digit of value of `i` is placed in its final location by getting the offset from `Bucket[i]` and incrementing the bucket so that the next key with the same value `i` gets placed in the next location. You should also notice that `COUNTING SORT` is stable.

Before we dive into parallelization of the algorithms and how they work in a cohesive manner, it's important to take the next few paragraphs to understand what prefix sums are; the next paragraph highlights why they matter in Radix sorts.

In the previous sections, we introduced MSD and LSD Radix sorts and the prefix sums computation is embedded in the code. However, we didn't flag it out for you then. So, now's the time and the following is the code (taken from the previous `lsd_sort` and `msd_sort` sections):

```
for(int k = 1; k < 256; k++)
      count[k] += count[k-1];
```

If you recall how MSD/LSD works, we basically create a histogram of the values we have encountered and, at each stage of the sorting, we compute the prefix sums so that the algorithm can know where to place the output in a sorted order. If you are still doubtful, you should stop now and flip back to that section and work through the LSD sorting for strings of three characters.

The prefix sums is actually a generalization of the global sum, and its original formulation goes something like the following:

The prefix sum operation takes a binary associative operator $\oplus$, and an ordered set of n elements, $[a_0, a_1, ..., a_{n-1}]$, and returns the ordered set $[a_0, (a_0 \oplus a_1), (a_0 \otimes a_1 \otimes a_2), ..., (a_0 \otimes a_1 ... \otimes a_{n-1})]$.

We use a concrete example like taking a summation over an arbitrary array like `[39, 23, 44, 15, 86]`. Using the addition operator, the output would be `[39, 62, 108, 125, 211]`, and it is not obvious why this sort of computation is important or is even needed. In fact it is not even clear whether there is a direct way to parallelize this algorithm because of dependencies that each subsequent computation relies on the previous.

A sequential version of the prefix sums which has a runtime of $O(n)$ can be expressed as follows, assuming there are two arrays `in_arr` and `out_arr`, and `out_arr` is designed to contain the prefix sums for `in_arr`:

```
sum = 0
out_arr[0] = 0
do i = 0 to lengthOf(in_arr)
  t = in_arr[i+1]
  sum = sum + t
  out_arr[i] = sum
```

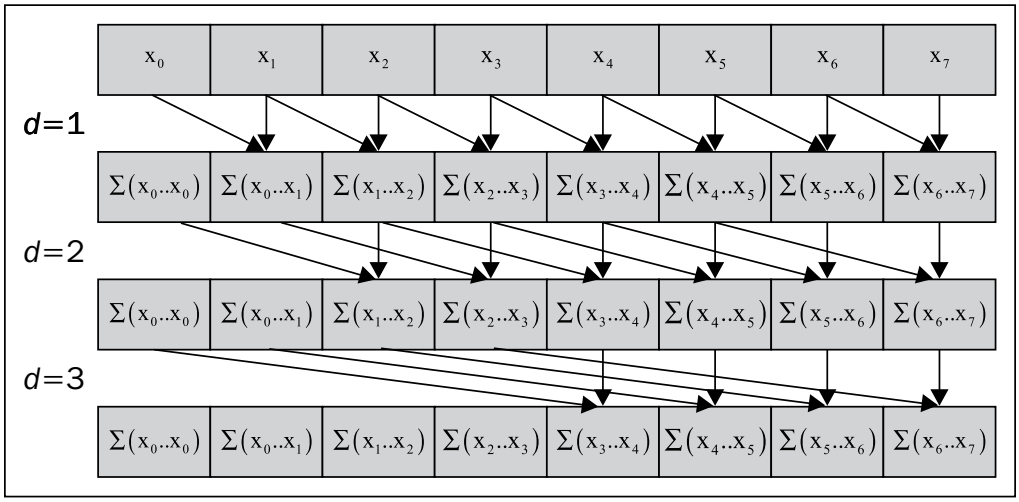

www.EngineeringBooksPdf.com

To extract parallelism from this, we need to adjust the way we view the arbitrary array of input values, and the adjustment we are talking about is actually imagining the array to be consumed by a tree of computations. Let's go on a little further to see why.

At this point, we think it's important to step back into history and see who came up with the original prefix sum computation. As far as I am aware, two researchers in 1986, *Daniel Hillis* and *Guy L. Steele*, presented a version of the prefix sum as part of an article titled *Data Parallel Algorithms* in the *ACM (Association for Computing Machinery)* magazine, and the algorithm they presented worked as follows (cited as such in that article):

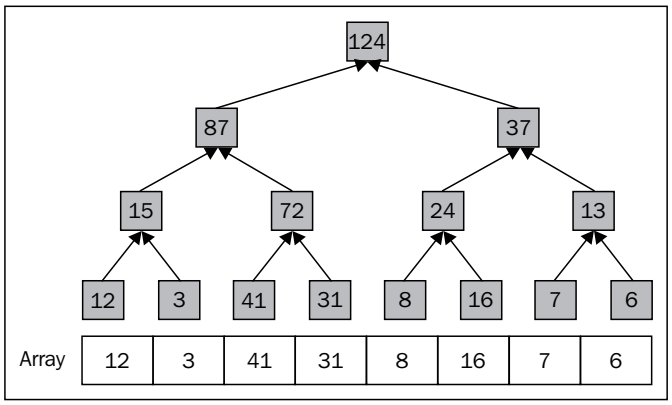
```
for j = 1 to log2n do
  for all k in parallel do
    if (k >= 2^j) then
      x[k] = x[k - 2^(j-1)] + x[k]
    fi
  endfor
endfor
```

The following diagram (courtesy of *Mark Harris* from the NVIDIA Corporation), pictorially illustrates what the Hillis and Steele algorithm does. It starts at the level where all eight elements are looked upon as leaves of the binary tree and proceeds to work its way through computing the partial sums. Each level of the computation, `d`, will compute partial sums based on the previous level's computation. An assumption found in the algorithm is that it assumes that there are as many processors as there are elements and this is demonstrated by the conditional statement in the algorithm, `if (k >= 2j)`. Another problem it has got is that it's not very efficient; it has a runtime complexity of $O(n\log_2 n)$, and you will recall that our sequential scan runs at $O(n)$, so it is definitely slower.



www.EngineeringBooksPdf.com

However, *Guy Blelloch* found ways to improve this, and they are based on the idea of building a balanced binary tree and building out that tree by performing addition on each node (conceptually speaking). Because such a tree with *n* leaves (which is corresponding to the number of elements in the array) would have $d = \log_2 n$ levels and each level has 2^d nodes, the runtime complexity is $O(n)$. The following diagram is an illustration of how a balanced binary tree can compute the array of arbitrary values:



The previous diagram created juxtaposition, and it alters the way the same piece of data you saw, that is, one dimensional flat array containing arbitrary values. Imagine a tree of computations that scans and operates on two values. One way of storing those partial sums is to write the value in place to the array and another way is to use shared memory on the device.

The astute reader in you would notice that we can probably parallelize the computation at each level of the tree by allowing one thread to read two elements, sum them up, and write them back into the array, and then you just read off the last element of that array for the final sum. This algorithm that we just described is known as a **reduction** kernel or an **up-sweep** kernel (since we are sweeping values up to the root of the tree), and we have seen how it works in the chapter where we discussed about sparse matrix computations in OpenCL. The following is the more formal definition of the reduction phase by *Guy Blelloch* when it's applied to a balanced binary tree with depth $\lg n$:

```
for d from 0 to (log2 n) – 1
  in parallel for i from 0 to n – 1 by 2^(d+1)
    array[i + 2^(d+1) – 1] = array[i + 2^d – 1] + array[i + 2^(d+1) – 1]
```

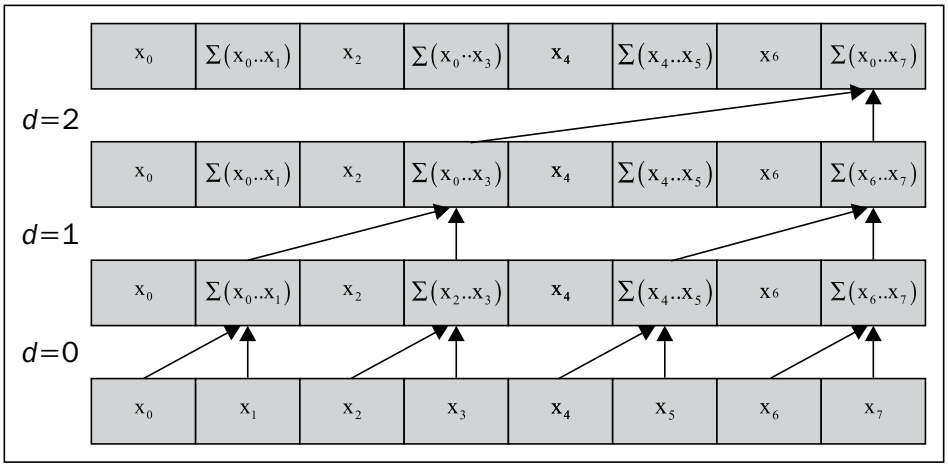

www.EngineeringBooksPdf.com

You might think that this up-sweep kernel still doesn't compute the prefix sums, but we do appear to have found a solution to solving summation in parallel; at this point, the following diagram will help us learn what actually goes on during a run of the up-sweep, and we find it helpful to flatten the loop a little to examine its memory access pattern.

Assuming we have eight elements in our array (that is, `n = 8`), our tree would have a depth of `3` and `d` would range from `0` to `2`. Imagining that we are at `d = 0`, through to `2` we would have the following expressions:

```
d = 0 => i = [0..7,2] array[i + 1] = array[i] + array[i + 1]
d = 1 => i = [0..7,4] array[i + 3] = array[i + 1] + array[i + 3]
d = 2 => i = [0..7,8] array[i + 7] = array[i + 3] + array[i + 7]
```

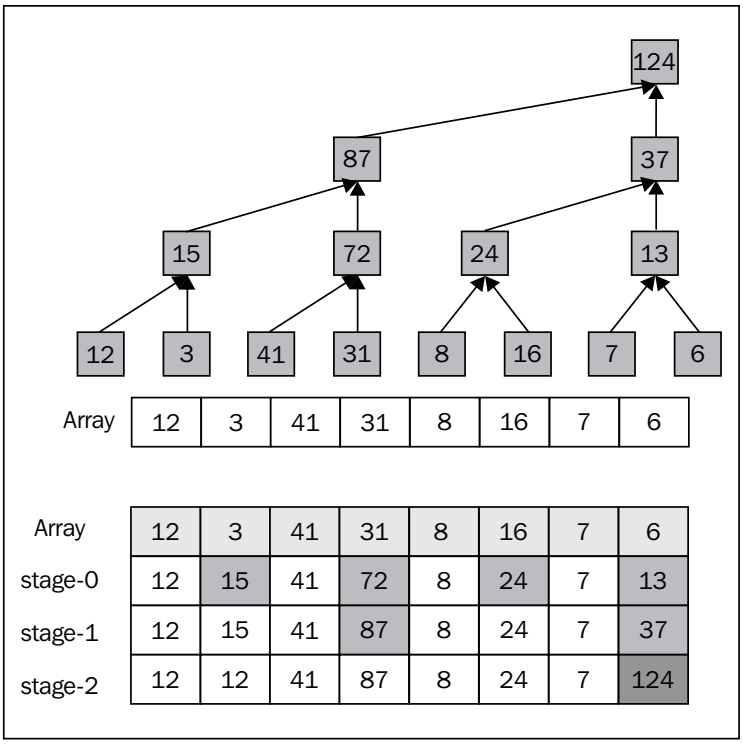
The next diagram best explains the evaluation of the preceding expressions, and a picture does reveal more about the story than plain equations:



From this diagram, we can observe that partial sums are built up at each level of the tree and one of the efficiencies introduced here is not repeating any addition, that is, no redundancies. Let's demonstrate how this would work for an array of eight elements, and we will also employ the up-sweep algorithm.

www.EngineeringBooksPdf.com

The following diagram illustrates the writes that occurred at each level of the tree we're scanning; in that diagram, the boxes colored blue represent the partial sums that were built up at each level of the tree, and the red box represents the final summed value:



Array	12	3	41	31	8	16	7	6
stage-0	12	15	41	72	8	24	7	13
stage-1	12	15	41	87	8	24	7	37
stage-2	12	12	41	87	8	24	7	124

To be able to compute the prefix sums from the up-sweep phase we need to proceed from the root of this tree and perform a *down-sweep* using this algorithm by *Guy Blelloch*:

```
x[n-1]=0
for d = log2 n – 1 to 0 do
  for all k = 0 to n – 1 by 2^(d+1) in parallel do
    temp = x[k + 2^d – 1]
    x[k + 2^d – 1] = x[k + 2^(d+1) – 1]
    x[k + 2^(d+1) – 1] = temp + x[k + 2^(d+1) – 1]
  endfor
endfor
```

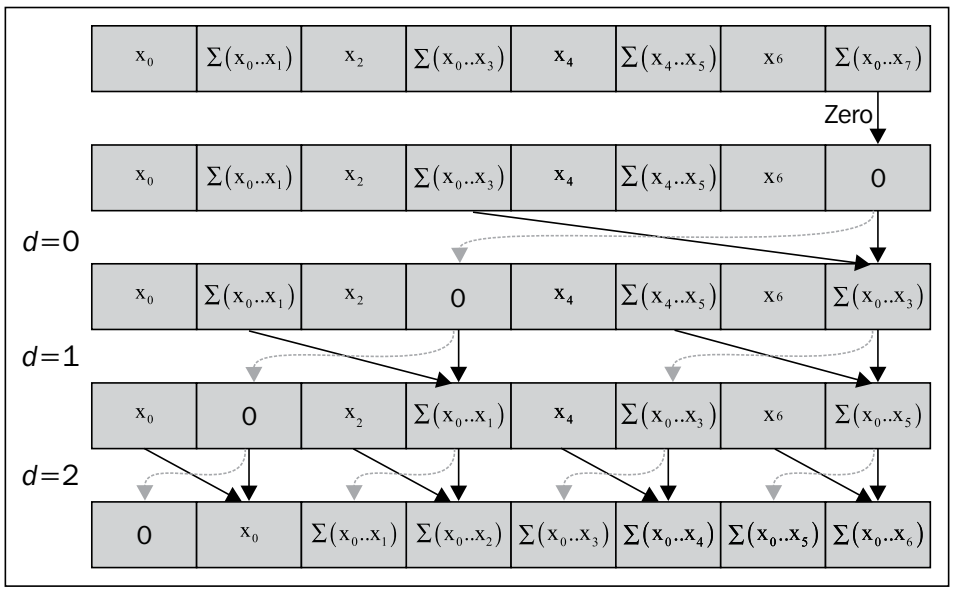

www.EngineeringBooksPdf.com

This down-sweep works its way down from the top (or root) of the tree after the reduce phase and builds the prefix sums. Let's flatten the loop to examine its memory access pattern.

As before with the up-sweep, let's assume that we have eight elements (that is, `n = 8`); we would have a depth of `3`, and that implies `d` would range from `0` to `2`. The following are the flattened expressions:

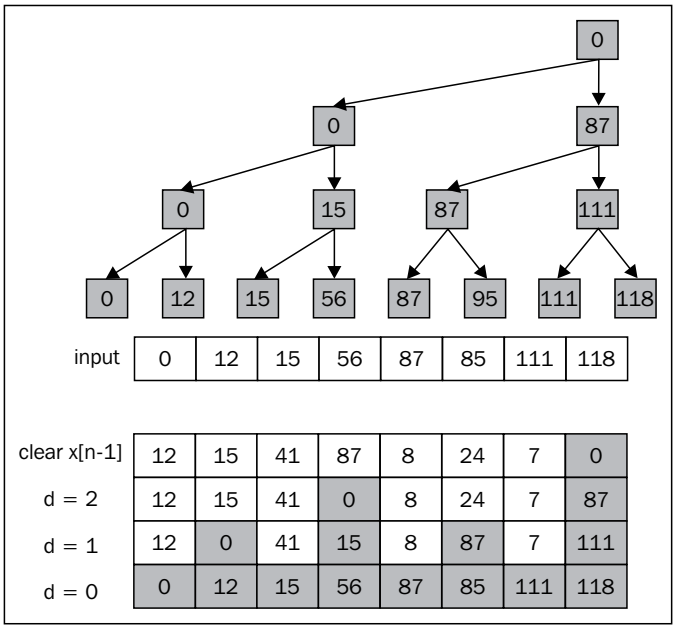
```
d = 2 => k = [0..7,8]
    temp = x[k + 3]
    x[k + 3] = x[k + 7]
    x[k + 7] = temp + x[k + 7]
d = 1 => k = [0..7,4]
    temp = x[k + 1]
    x[k + 1] = x[k + 3]
    x[k + 3] = temp + k[x + 3]
d = 0 => k = [0..7,2]
    temp = x[k]
    x[k] = x[k + 1]
    x[k + 1] = temp + x[k + 1]
```

The following diagram best expresses how the prefix sums are computed from the reduce/up-sweep phase:



www.EngineeringBooksPdf.com

Let us concretize these ideas by looking at how the down-sweep phase would proceed after the reduce/up-sweep phase using the following diagram; the `input` array is the original array, and we have kept it there for you to verify that the prefix sum computation according to the previous algorithm is correct. The lower portion of the diagram illustrates how memory is accessed. Keep in mind that updates are done in place, and when you combine the diagrams of the up-sweep and down-sweep phases, you'll notice that we make two passes over the original input array to arrive at the solution of prefix sums, which is what we wanted:



How to do it ...

The kernel we present here is found in `Ch10/RadixSort_GPU/RadixSort.cl`, and the implementation drew inspiration from the academic paper entitled *Radix Sort for Vector Multiprocessors* by *Mark Zagha* and *Guy E. Blelloch* for 32-bit integers. The algorithm is based on the LSD Radix sort, and it iterates all the keys while shifting the keys based on the chosen Radix and executing OpenCL kernels in sequence; this is best described in the previous diagram.

As before, we present the sequential version of the Radix sort that was translated based on *Zagha* and *Blelloch*, and like what we have done previously, this is the golden reference which we will use to determine the correctness of the data calculated by the OpenCL equivalent. We won't spend too much time discussing about this implementation here, but rather it serves as a reference point where you can draw the similarities and contrasts when we demonstrate how the parallel and sequential code differs:

www.EngineeringBooksPdf.com

```
int radixSortCPU(cl_uint* unsortedData, cl_uint* hSortedData) {

    cl_uint *histogram = (cl_uint*) malloc(R * sizeof(cl_uint));
    cl_uint *scratch = (cl_uint*) malloc(DATA_SIZE * sizeof(cl_uint));

    if(histogram != NULL && scratch != NULL) {

        memcpy(scratch, unsortedData, DATA_SIZE * sizeof(cl_uint));
        for(int bits = 0; bits < sizeof(cl_uint) * bitsbyte ; bits +=
bitsbyte) {

            // Initialize histogram bucket to zeros
            memset(histogram, 0, R * sizeof(cl_uint));

            // Calculate 256 histogram for all element
            for(int i = 0; i < DATA_SIZE; ++i)
            {
                cl_uint element = scratch[i];
                cl_uint value = (element >> bits) & R_MASK;
                histogram[value]++;
            }

            // Apply the prefix-sum algorithm to the histogram
            cl_uint sum = 0;
            for(int i = 0; i < R; ++i)
            {
                cl_uint val = histogram[i];
                histogram[i] = sum;
                sum += val;
            }

            // Rearrange the elements based on prescanned histogram
            // Thus far, the preceding code is basically adopted from
            // the "counting sort" algorithm.
            for(int i = 0; i < DATA_SIZE; ++i)
            {
                cl_uint element = scratch[i];
                cl_uint value = (element >> bits) & R_MASK;
                cl_uint index = histogram[value];
                hSortedData[index] = scratch[i];
                histogram[value] = index + 1;
            }

            // Copy to 'scratch' for further use since we are not done
```


www.EngineeringBooksPdf.com

```
yet
            if(bits != bitsbyte * 3)
                memcpy(scratch, hSortedData, DATA_SIZE * sizeof(cl_
uint));
        }
    }

    free(scratch);
    free(histogram);
    return 1;
}
```

This sequential code is akin to the `lsd_sort` code we showed earlier, and it essentially builds a histogram of the examined keys that uses the counting sort to sort them, and it keeps doing this until all data is acted upon.

The following kernels are taken from `Ch10/RadixSort_GPU/RadixSort.cl`, and we'll refer to the appropriate code when we explain the internal workings of the algorithm:

```
#define bitsbyte 8
#define R (1 << bitsbyte)

__kernel void computeHistogram(__global const uint* data,
                               __global uint* buckets,
                               uint shiftBy,
                               __local uint* sharedArray) {

    size_t localId = get_local_id(0);
    size_t globalId = get_global_id(0);
    size_t groupId = get_group_id(0);
    size_t groupSize = get_local_size(0);

    /* Initialize shared array to zero i.e. sharedArray[0..63] = {0}*/
    sharedArray[localId] = 0;
    barrier(CLK_LOCAL_MEM_FENCE);

    /* Calculate thread-histograms local/shared memory range from 32KB
to 64KB */

    uint result= (data[globalId] >> shiftBy) & 0xFFU;
    atomic_inc(sharedArray+result);

    barrier(CLK_LOCAL_MEM_FENCE);

    /* Copy calculated histogram bin to global memory */
```


www.EngineeringBooksPdf.com

```
    uint bucketPos = groupId * groupSize + localId ;
    buckets[bucketPos] = sharedArray[localId];
}
__kernel void rankNPermute(__global const uint* unsortedData,
                           __global const uint* scannedHistogram,
                           uint shiftCount,
                           __local ushort* sharedBuckets,
                           __global uint* sortedData) {

    size_t groupId = get_group_id(0);
    size_t idx = get_local_id(0);
    size_t gidx = get_global_id(0);
    size_t groupSize = get_local_size(0);

    /* There are now GROUP_SIZE * RADIX buckets and we fill
       the shared memory with those prefix-sums computed previously
     */
    for(int i = 0; i < R; ++i)
    {
        uint bucketPos = groupId * R * groupSize + idx * R + i;
        sharedBuckets[idx * R + i] = scannedHistogram[bucketPos];
    }

    barrier(CLK_LOCAL_MEM_FENCE);

    /* Using the idea behind COUNTING-SORT to place the data values in
its sorted
       order based on the current examined key
     */
    for(int i = 0; i < R; ++i)
    {
        uint value = unsortedData[gidx * R + i];
        value = (value >> shiftCount) & 0xFFU;
        uint index = sharedBuckets[idx * R + value];
        sortedData[index] = unsortedData[gidx * R + i];
        sharedBuckets[idx * R + value] = index + 1;
        barrier(CLK_LOCAL_MEM_FENCE);
    }
}
__kernel void blockScan(__global uint *output,
                        __global uint *histogram,
                        __local uint* sharedMem,
                        const uint block_size,
```


www.EngineeringBooksPdf.com

```
                        __global uint* sumBuffer) {
  int idx = get_local_id(0);
  int gidx = get_global_id(0);
  int gidy = get_global_id(1);
  int bidx = get_group_id(0);
  int bidy = get_group_id(1);

  int gpos = (gidx << bitsbyte) + gidy;
  int groupIndex = bidy * (get_global_size(0)/block_size) + bidx;

  /* Cache the histogram buckets into shared memory
     and memory reads into shared memory is coalesced
  */
  sharedMem[idx] = histogram[gpos];
  barrier(CLK_LOCAL_MEM_FENCE);

/*
   Build the partial sums sweeping up the tree using
   the idea of Hillis and Steele in 1986
 */
uint cache = sharedMem[0];
for(int stride = 1; stride < block_size; stride <<= 1)
{
    if(idx>=stride)
    {
        cache = sharedMem[idx-stride]+block[idx];
    }
    barrier(CLK_LOCAL_MEM_FENCE); // all threads are blocked here

    sharedMem[idx] = cache;
    barrier(CLK_LOCAL_MEM_FENCE);
}

/* write the array of computed prefix-sums back to global memory
*/
    if(idx == 0)
    {
        /* store the value in sum buffer before making it to 0 */
        sumBuffer[groupIndex] = sharedMem[block_size-1];
        output[gpos] = 0;
    }
    else
    {
        output[gpos] = sharedMem[idx-1];
```


www.EngineeringBooksPdf.com

```
    }
}
__kernel void unifiedBlockScan(__global uint *output,
                               __global uint *input,
                               __local uint* sharedMem,
                               const uint block_size) {

    int id = get_local_id(0);
    int gid = get_global_id(0);
    int bid = get_group_id(0);

    /* Cache the computational window in shared memory */
    sharedMem[id] = input[gid];

    uint cache = sharedMem[0];

    /* build the sum in place up the tree */
    for(int stride = 1; stride < block_size; stride <<= 1)
    {
        if(id>=stride)
        {
            cache = sharedMem[id-stride]+sharedMem[id];
        }
        barrier(CLK_LOCAL_MEM_FENCE);

        sharedMem[id] = cache;
        barrier(CLK_LOCAL_MEM_FENCE);

    }
    /*write the results back to global memory */
    if(tid == 0) {
        output[gid] = 0;
    } else {
        output[gid] = sharedMem[id-1];
    }
}
__kernel void blockPrefixSum(__global uint* output,
                             __global uint* input,
                             __global uint* summary,
                             int stride) {

     int gidx = get_global_id(0);
     int gidy = get_global_id(1);
     int Index = gidy * stride +gidx;
```


www.EngineeringBooksPdf.com

```
        output[Index] = 0;

         // Notice that you don't need memory fences in this kernel
         // because there is no race conditions and the assumption
         // here is that the hardware schedules the blocks with lower
         // indices first before blocks with higher indices
        if(gidx > 0)
        {
            for(int i =0;i<gidx;i++)
                output[Index] += input[gidy * stride +i];
        }
        // Write out all the prefix sums computed by this block
        if(gidx == (stride - 1))
            summary[gidy] = output[Index] + input[gidy * stride + (stride
-1)];
}

__kernel void blockAdd(__global uint* input,
                       __global uint* output,
                       uint stride) {

        int gidx = get_global_id(0);
        int gidy = get_global_id(1);
        int bidx = get_group_id(0);
        int bidy = get_group_id(1);


        int gpos = gidy + (gidx << bitsbyte);

        int groupIndex = bidy * stride + bidx;

        uint temp;
        temp = input[groupIndex];

        output[gpos] += temp;
}
__kernel void mergePrefixSums(__global uint* input,
                         __global uint* output) {

    int gidx = get_global_id(0);
    int gidy = get_global_id(1);
    int gpos = gidy + (gidx << bitsbyte );
    output[gpos] += input[gidy];
}
```


www.EngineeringBooksPdf.com

How it works...

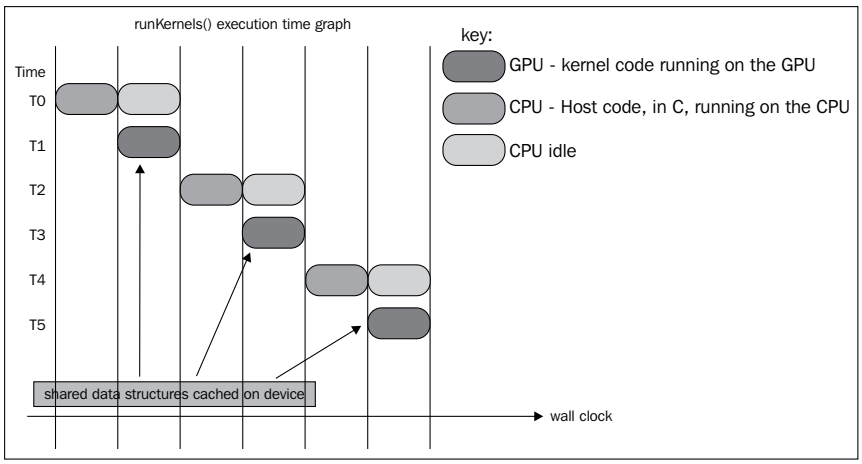
The strategy we present here is to break keys, that is, break 32-bit integers into 8-bit digits, and then sort them one at a time starting from the least significant digit. Based on this idea, we are going to loop four times and at each loop number *i*, we are going to examine the *i* numbered 8-bit digit.

The general looping structure based on the previous description is given in the following code:

```
void runKernels(cl_uint* dSortedData, size_t numOfGroups, size_t
groupSize) {
   for(int currByte = 0; currByte < sizeof(cl_uint) * bitsbyte;
currByte += bitsbyte) {
    computeHistogram(currByte);
    computeBlockScans();
    computeRankingNPermutations(currByte,groupSize);
  }
}
```

The three invocations in the loop are the work horses of this implementation and they invoke the kernels to compute the histogram from the input based on the current byte we are looking at. The algorithm will basically compute the histogram of the keys that it has examined; the next phase is to compute the prefix sums (we'll be using the Hillis and Steele algorithm for this), and finally we will update the data structures and write out the values in a sorted order. Let's go into detail about how this works.

In the host code, you will need to prepare the data structures slightly differently than what we have shown you so far, because these structures need to be shared across various kernels while we swing between host code and kernel code. The following diagram illustrates this general idea for `runKernels()`, and this situation is because we created a single command queue which all kernels will latch on to in program order; this applies to their execution as well:



www.EngineeringBooksPdf.com

For this implementation, the data structure that holds the unsorted data (that is, `unsortedData_d`) needs to be read and shared across the kernels. Therefore, you need to create the device buffer with the flag `CL_MEM_USE_HOST_PTR` since the OpenCL specification guarantees that the implementations cached it across multiple kernel invocations. Next, we will look at how the histogram is computed on the GPU.

The computation of the histogram is based on the threaded histogram we introduced in a previous chapter, but this time around, we decided to show you another implementation which is based on using atomic functions in OpenCL, and in particular using `atomic_inc()`. The `atomic_inc` function will update the value pointed by the location by one. The histogram works on the OpenCL-supported GPU because we have chosen to use the shared memory and CPU doesn't support that yet. The strategy is to divide our input array into blocks of *N x R* elements where *R* is the radix (in our case *R = 8* since each digit is 8-bits wide and $2^8=256$) and *N* is the number of threads executing the block. This strategy is based on the assumption that our problem sizes are always going to be much larger than the amount of threads available, and we configure it programmatically on the host code prior to launching the kernel as shown in the following code:

```
void computeHistogram(int currByte) {
    cl_event execEvt;
    cl_int status;
    size_t globalThreads = DATA_SIZE;
    size_t localThreads  = BIN_SIZE;
    status = clSetKernelArg(histogramKernel, 0, sizeof(cl_mem),
             (void*)&unsortedData_d);
    status = clSetKernelArg(histogramKernel, 1, sizeof(cl_mem),
             (void*)&histogram_d);
    status = clSetKernelArg(histogramKernel, 2, sizeof(cl_int),
             (void*)&currByte);
    status = clSetKernelArg(histogramKernel, 3, sizeof(cl_int) *
             BIN_SIZE, NULL);
    status = clEnqueueNDRangeKernel(
        commandQueue,
        histogramKernel,
        1,
        NULL,
        &globalThreads,
        &localThreads,
        0,
        NULL,
        &execEvt);
    clFlush(commandQueue);
    waitAndReleaseDevice(&execEvt);
}
```


www.EngineeringBooksPdf.com

By setting up the OpenCL thread block to be equal to `BIN_SIZE`, that is, 256, the kernel waits for the computation to complete by polling the OpenCL device for its execution status; this poll-release mechanism is encapsulated by `waitAndReleaseDevice()`.

When you have multiple kernel invocations and one kernel waits on the other, you need synchronization, and OpenCL provides this via `clGetEventInfo` and `clReleaseEvent`.

In the histogram kernel, we built up the histogram by reading the inputs into shared memory (after initializing it to zero), and to prevent any threads from executing kernel code that reads from shared memory before all data is loaded into it, we placed a memory barrier as follows:

```
/* Initialize shared array to zero i.e. sharedArray[0..63] = {0}*/
sharedArray[localId] = 0;
barrier(CLK_LOCAL_MEM_FENCE);
```

It's debatable whether we should initialize the shared memory, but it's best practice to initialize data structures, just like you would do in other programming languages. The trade off, in this case, is program correctness versus wasting processor cycles.

Next, we shift the data value (residing in shared memory) by a number, `shiftBy`, which is the key we are sorting, extract the byte, and then update the local histogram atomically. We placed a memory barrier thereafter. Finally, we write out the binned values to their appropriate location in the global histogram, and you will notice that this implementation performs what we call *scattered writes*:

```
uint result= (data[globalId] >> shiftBy) & 0xFFU; //5
atomic_inc(sharedArray+result);                   //6

barrier(CLK_LOCAL_MEM_FENCE);                     //7

/* Copy calculated histogram bin to global memory */

uint bucketPos = groupId  * groupSize + localId ; //8
buckets[bucketPos] = sharedArray[localId];        //9
```

Once the histogram is established, the next task that `runKernels()` performs is to execute the computations of prefix sums in the kernels `blockScan`, `blockPrefixSum`, `blockAdd`, `unifiedBlockScan`, and `mergePrefixSums` in turn. We'll explain what each kernel does in the following sections.

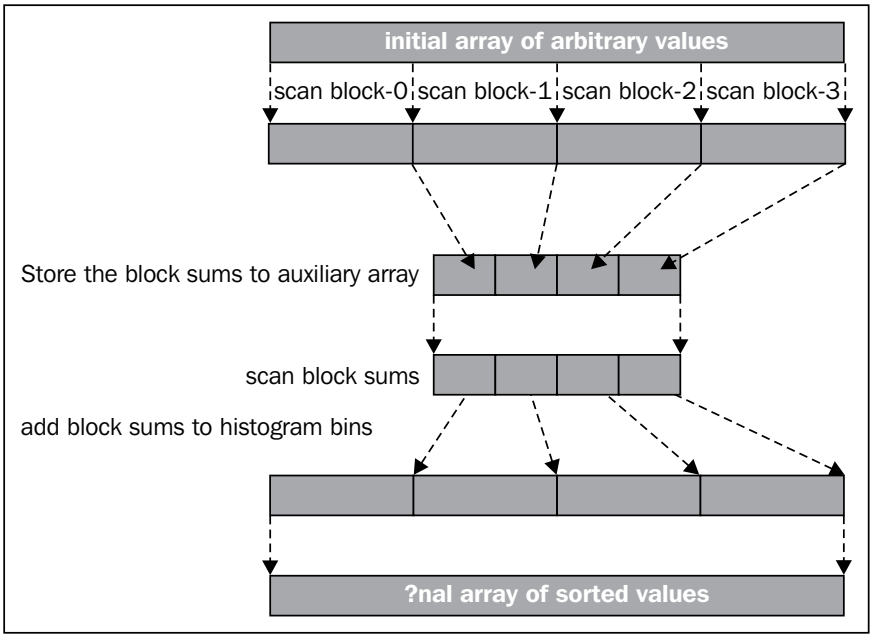
www.EngineeringBooksPdf.com

The general strategy for this phase (encapsulated in `computeBlockScans()`) is to pre-scan the histogram bins so that we generate the prefix sums for each bin. We then write out that value to an auxiliary data structure, `sum_in_d`, and write out all intermediary sums into another auxiliary data structure, `scannedHistogram_d`. The following is the configuration we sent to the `blockScan` kernel:

```
    size_t numOfGroups = DATA_SIZE / BIN_SIZE;
    size_t globalThreads[2] = {numOfGroups, R};
    size_t localThreads[2] = {GROUP_SIZE, 1};
    cl_uint groupSize = GROUP_SIZE;

status = clSetKernelArg(blockScanKernel, 0, sizeof(cl_mem),
(void*)&scannedHistogram_d);
    status = clSetKernelArg(blockScanKernel, 1, sizeof(cl_mem),
(void*)&histogram_d);
    status = clSetKernelArg(blockScanKernel, 2, GROUP_SIZE *
sizeof(cl_uint), NULL);
    status = clSetKernelArg(blockScanKernel, 3, sizeof(cl_uint),
&groupSize);
    status = clSetKernelArg(blockScanKernel, 4, sizeof(cl_mem), &sum_
in_d);
    cl_event execEvt;
    status = clEnqueueNDRangeKernel(
                commandQueue,
                blockScanKernel,
                2,
                NULL,
                globalThreads,
                localThreads,
                0,
                NULL,
                &execEvt);
    clFlush(commandQueue);
    waitAndReleaseDevice(&execEvt);
```

The general strategy behind scanning is illustrated in the following diagram, where the input is divided into separate blocks and each block will be submitted for a block scan. The generated results are prefix sums, but we need to collate these results across all blocks to obtain a cohesive view. After which, the histogram bins are updated with these prefix sum values, and then finally we can use the updated histogram bins to sort the input array.

www.EngineeringBooksPdf.com



Let's look at how the block scan is done by examining `blockScan`. First, we load the values from the previously computed histogram bin into its shared memory as shown in the following code:

```
__kernel void blockScan(__global uint *output,
                        __global uint *histogram,
                        __local uint* sharedMem,
                        const uint block_size,
                        __global uint* sumBuffer) {
    int idx = get_local_id(0);
    int gidx = get_global_id(0);
    int gidy = get_global_id(1);
    int bidx = get_group_id(0);
    int bidy = get_group_id(1);

    int gpos = (gidx << bitsbyte) + gidy;
    int groupIndex = bidy * (get_global_size(0)/block_size) + bidx;

    /* Cache the histogram buckets into shared memory
       and memory reads into shared memory is coalesced
    */
    sharedMem[idx] = histogram[gpos];
    barrier(CLK_LOCAL_MEM_FENCE);
```


www.EngineeringBooksPdf.com

Next, we perform the Hillis and Steele prefix sum algorithm locally, and build the summed values for the current block:

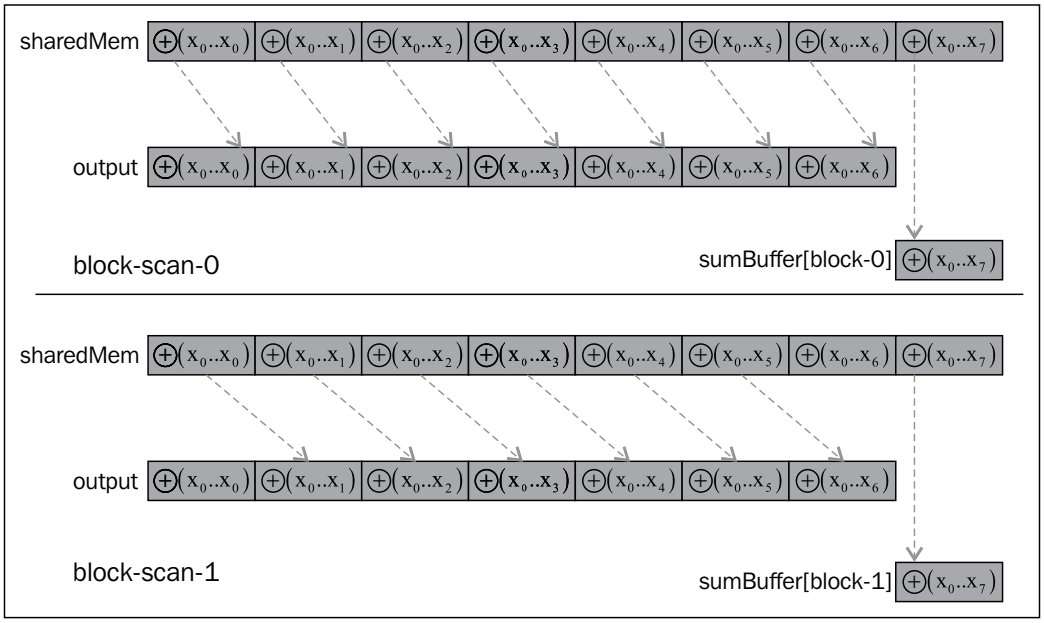
```
    /*
       Build the partial sums sweeping up the tree using
       the idea of Hillis and Steele in 1986
     */
    uint cache = sharedMem[0];
    for(int dis = 1; dis < block_size; dis <<= 1)
    {
        if(idx>=dis)
        {
            cache = sharedMem[idx-dis]+block[idx];
        }
        barrier(CLK_LOCAL_MEM_FENCE); // all threads are blocked here

        sharedMem[idx] = cache;
        barrier(CLK_LOCAL_MEM_FENCE);
    }
```

Finally, we write out a prefix sum for this block to `sum_in_d`, represented in the following code by `sumBuffer`, and the intermediary prefix sums to the `scannedHistogram_d` object, represented here by `output`:

```
    /* write the array of computed prefix-sums back to global memory
*/
    if(idx == 0)
    {
        /* store the value in sum buffer before making it to 0 */
        sumBuffer[groupIndex] = sharedMem[block_size-1];
        output[gpos] = 0;
    } else {
        output[gpos] = sharedMem[idx-1];
    }
}
```


www.EngineeringBooksPdf.com

The following diagram illustrates this concept for two parallel block scans (assuming we have a shared memory that holds eight elements) and shows how it's stored into the output:



At this phase of the computation, we have managed to compute the prefix sums for all the individual blocks. We need to collate them through the next phase, which is in the kernel `blockPrefixSum` where the individual block's summed value is accumulated by each work item. The work done by each thread will compute the sum across different blocks. Depending on the thread with ID, `i`, will gather all sums from block number `0` to `(i - 1)`. The following code in `blockPrefixSum` illustrates this process:

```
__kernel void blockPrefixSum(__global uint* output,
                             __global uint* input,
                             __global uint* summary,
                             int stride) {

    int gidx = get_global_id(0);
    int gidy = get_global_id(1);
    int Index = gidy * stride +gidx;
    output[Index] = 0;

    if(gidx > 0) {
        for(int i =0;i<gidx;i++)
            output[Index] += input[gidy * stride +i];
    }
```

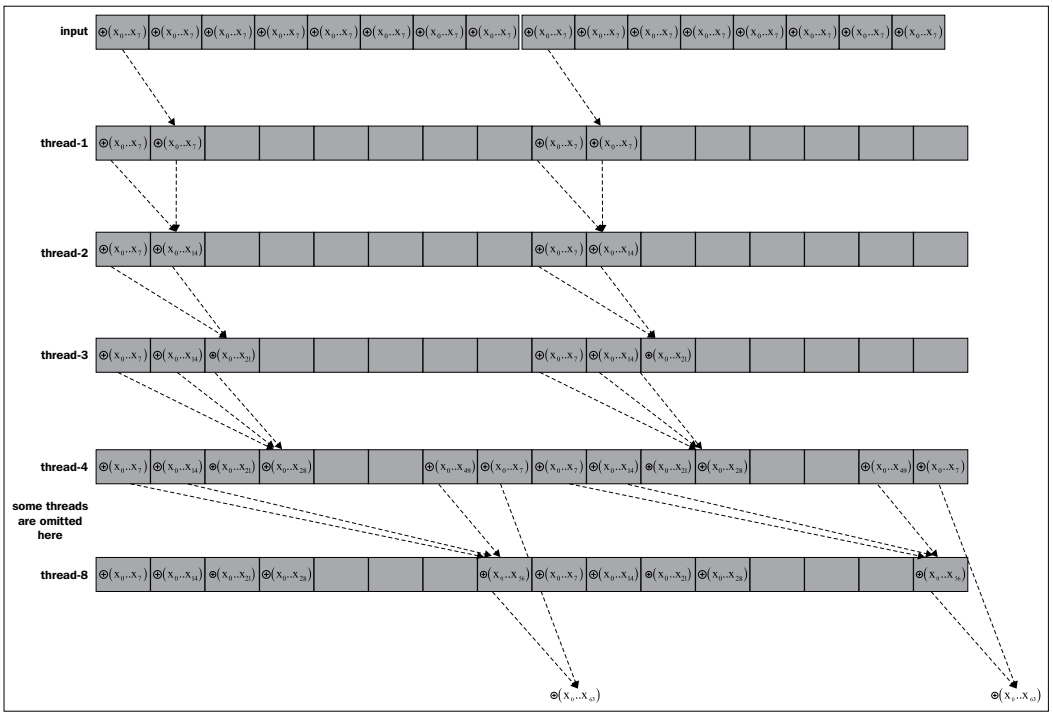

www.EngineeringBooksPdf.com

The astute reader will notice that we have left out the prefix sum for one block, and the following remedies are obtained by computing the final accumulated prefix sums for this block:

```
        // Write out all the prefix sums computed by this block
        if(gidx == (stride - 1))
            summary[gidy] = output[Index] + input[gidy * stride + (stride
    -1)];
```

The following diagram best represents what computation goes on in the previous kernel code. It assumes that we have a block scan for 16 elements that has been completed in `blockScanKernel`, and each element contains the prefixed sum. To collate these sums, we configure our kernel to run eight threads with a striding factor of `8` (assuming a block size of eight), and the diagram expresses what each of the eight threads are working on.
The threads collate the sums by working out the summation of the entire input, progressively computing $\oplus(x_0...x_7)\oplus(x_0...x_{14})\oplus(x_0...x_{21})\oplus(x_0...x_{28})\oplus(x_0...x_{49})\oplus(x_0...x_{56})\oplus(x_0...x_{63})$ and writing them out to `sum_out_d` and `summary_in_d`.

The following is a diagram that illustrates the process where given an input, all elements of that input are the summed values of the block scan for all blocks; the algorithm basically sums everything and writes to the output array:



www.EngineeringBooksPdf.com

At this point, we have to collate the intermediary prefix sums computed, that is, $\oplus(x_0 \ldots x_7) \oplus (x_0 \ldots x_{14}) \oplus (x_0 \ldots x_{21}) \oplus (x_0 \ldots x_{28}) \oplus (x_0 \ldots x_{28}) \oplus (x_0 \ldots x_{49}) \oplus (x_0 \ldots x_{56})$ inside `sum_out_d`, and with that from `scannedHistogram_d`. We basically add the two intermediary sums together using `blockAddKernel`. The following is how we prepare the kernel prior to launch:

```
        cl_event execEvt2;
        size_t globalThreadsAdd[2] = {numOfGroups, R};
        size_t localThreadsAdd[2] = {GROUP_SIZE, 1};
        status = clSetKernelArg(blockAddKernel, 0, sizeof(cl_mem),
(void*)&sum_out_d);
        status = clSetKernelArg(blockAddKernel, 1, sizeof(cl_mem),
(void*)&scannedHistogram_d);
        status = clSetKernelArg(blockAddKernel, 2, sizeof(cl_uint),
(void*)&stride);
        status = clEnqueueNDRangeKernel(
                    commandQueue,
                    blockAddKernel,
                    2,
                    NULL,
                    globalThreadsAdd,
                    localThreadsAdd,
                    0,
                    NULL,
                    &execEvt2);
        clFlush(commandQueue);
        waitAndReleaseDevice(&execEvt2);
```

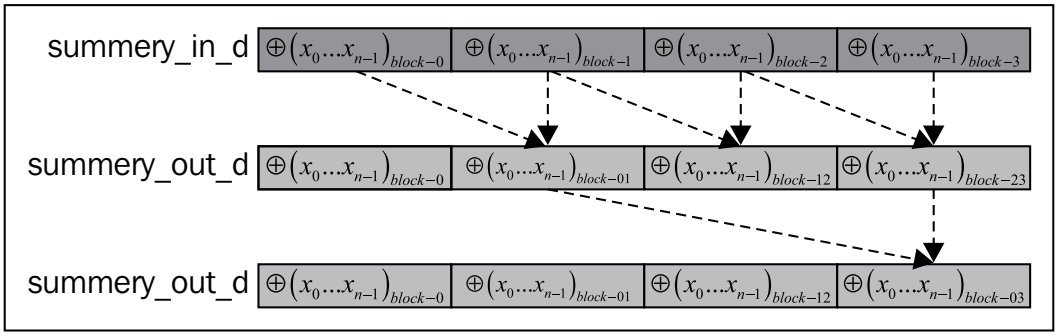
We then basically collate them back to `scannedHistogram_d` with `blockAddKernel` whose code is shown as follows:

```
__kernel void blockAdd(__global uint* input,
                       __global uint* output,
                       uint stride) {

      int gidx = get_global_id(0);
      int gidy = get_global_id(1);
      int bidx = get_group_id(0);
      int bidy = get_group_id(1);


      int gpos = gidy + (gidx << bitsbyte);

      int groupIndex = bidy * stride + bidx;

      uint temp;
```


www.EngineeringBooksPdf.com

```
        temp = input[groupIndex];

        output[gpos] += temp;
}
```

Finally, we perform another prefix sum to collate the values in `summary_in_d`, as all elements inside that array contains each individual block's prefix sum. Because our chosen Radix value is `256`, we need to work out the prefix sums computation for blocks `0` to `y` using $\oplus(x_0 \ldots x_{n-1})_{block-0}$ through to $\oplus(x_0 \ldots x_{n-1})_{block-y}$. This is illustrated in the following diagram, and it is encapsulated in the `unifiedBlockScan` kernel. We won't show the kernel code as it's similar to the `blockPrefixSum` kernel.



At this point in time, we are left with writing the collated prefix sums we have just performed previously into `scannedHistogram_d`. This collation exercise is different from the previous one where we gather the intermediary prefix sums across the blocks, but nonetheless, it's still a collation exercise, and we need to push in the values from `summary_in_d`. We accomplished this with `mergePrefixSumsKernel` with the inputs reflected in the following host code:

```
        cl_event execEvt4;
        size_t globalThreadsOffset[2] = {numOfGroups, R};
        status = clSetKernelArg(mergePrefixSumsKernel, 0, sizeof(cl_
mem), (void*)&summary_out_d);
        status = clSetKernelArg(mergePrefixSumsKernel, 1, sizeof(cl_
mem), (void*)&scannedHistogram_d);
        status = clEnqueueNDRangeKernel(commandQueue,
mergePrefixSumsKernel, 2, NULL, globalThreadsOffset, NULL, 0, NULL,
&execEvt4);
        clFlush(commandQueue);
        waitAndReleaseDevice(&execEvt4);
```


www.EngineeringBooksPdf.com

The `mergePrefixSumsKernel` exercise is a relatively simple exercise to shift the values to their proper positions with the following kernel code:

```
__kernel void mergePrefixSums(__global uint* input,
                              __global uint* output) {

   int gidx = get_global_id(0);
   int gidy = get_global_id(1);
   int gpos = gidy + (gidx << bitsbyte );
   output[gpos] += input[gidy];
}
```

With this, the prefix sums are properly computed. The next phase of the algorithm will be to rank and permute the keys using each work item / thread to permute its 256 elements via the prescanned histogram bins, encapsulated in `computeRankNPermutations()`.
The following is the host code for the kernel launch:

```
void computeRankingNPermutations(int currByte, size_t groupSize) {
    cl_int status;
    cl_event execEvt;

    size_t globalThreads = DATA_SIZE/R;
    size_t localThreads = groupSize;

    status = clSetKernelArg(permuteKernel, 0, sizeof(cl_mem),
(void*)&unsortedData_d);
    status = clSetKernelArg(permuteKernel, 1, sizeof(cl_mem),
(void*)&scannedHistogram_d);
    status = clSetKernelArg(permuteKernel, 2, sizeof(cl_int),
(void*)&currByte);
    status = clSetKernelArg(permuteKernel, 3, groupSize * R *
sizeof(cl_ushort), NULL); // shared memory
    status = clSetKernelArg(permuteKernel, 4, sizeof(cl_mem),
(void*)&sortedData_d);

    status = clEnqueueNDRangeKernel(commandQueue, permuteKernel, 1,
NULL, &globalThreads, &localThreads, 0, NULL, &execEvt);
    clFlush(commandQueue);
    waitAndReleaseDevice(&execEvt);
```

Once the kernel has completed successfully, the data values will be in a sorted order and will be held in the device memory by `sortedData_d`. We need to copy those data into `unsortedData_d` again, and we will continue to do this until we have not completed the iteration of the keys.

www.EngineeringBooksPdf.com

In the `rankNPermute` kernel, we will again make use of shared memory. The data into shared memory, and the data is organized as `GROUP_SIZE * RADIX` where the `GROUP_SIZE = 64` and `RADIX = 256` expressions hold true, and because each work group is configured to execute with 64 threads, we basically have one thread hydrating 256 elements of its shared memory (which the following code snippet demonstrates):

```
__kernel void rankNPermute(__global const uint* unsortedData,
                           __global const uint* scannedHistogram,
                           uint shiftCount,
                           __local ushort* sharedBuckets,
                           __global uint* sortedData) {
    size_t groupId = get_group_id(0);
    size_t idx = get_local_id(0);
    size_t gidx = get_global_id(0);
    size_t groupSize = get_local_size(0);
    for(int i = 0; i < R; ++i) {
        uint bucketPos = groupId * R * groupSize + idx * R + i;
        sharedBuckets[idx * R + i] = scannedHistogram[bucketPos];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
```

Next, it ranks the elements based on the same idea as in the sequential algorithm, and you should refer back to that now. The difference is that we are pulling data values from `unsortedData` in global device memory, processing them in device memory, figuring out where the values should be, and writing them out to `sortedData`:

```
    for(int i = 0; i < R; ++i) {
        uint value = unsortedData[gidx * R + i];
        value = (value >> shiftCount) & 0xFFU;
        uint index = sharedBuckets[idx * R + value];
        sortedData[index] = unsortedData[gidx * R + i];
        sharedBuckets[idx * R + value] = index + 1;
        barrier(CLK_LOCAL_MEM_FENCE);
    }
```

After the ranking and permutation is done, the data values in the `sortedData_d` object are sorted based on the current examined key. The algorithm will copy the data in `sortedData_d` into `unsortedData_d` so that the entire process can be repeated for a total of four times.

www.EngineeringBooksPdf.com

Index

Symbols

1D, convolution 157, 159
2-bit rounding control (RC) field 129
2D, convolution 159-162
__constant address space name 102
__global address space name 101
__local memory space 101
__private memory space 101

A

Abstract Data Type (ADT) 152
alloca 44
arithmetic operation 129-131
array
  vectors, loading from 114-117
  vectors, storing to 110-114
associative reduction tree 252

B

base-R number system 242
bitonic sort 222, 241
bitonic sorting
  about 224, 226
  developing, in OpenCL 230-239
  working 226-230
bitonic split 225
bool data type 80
bucket sorting. *See* **Radix sort**
buffer 136
buffer objects, OpenCL
  creating 44-50
  information, retrieving about 50-53

C

CAS (Compare-And-Swap) 120
C/C++
  histogram, implementing 139-142
cl_APPLE_gl_sharing extension 23
clGetPlatformInfo() method 18
CLK_GLOBAL_MEM_FENCE barrier 154
cl_khr_3d_image_writes extension 22
cl_khr_byte_addressable_store extension 23
cl_khr_d3d10_sharing extension 23
cl_khr_fp16 extension 22
cl_khr_fp64 extension 22
cl_khr_gl_event extension 23
cl_khr_global_int32_base_atomics extension 22
cl_khr_global_int32_extended_atomics extension 22
cl_khr_gl_sharing extension 23
cl_khr_int64_base_atomics extension 22
cl_khr_int64_extended_atomics extension 22
cl_khr_local_int32_base_atomics extension 22
cl_khr_local_int32_extended_atomics extension 22
CLK_LOCAL_MEM_FENCE barrier 154
command events 61
command queues
  creating 38-42
commutative property 175
commutative reduction tree 252, 253
compute units (CUs) 10, 89
configuration, OpenCL projects
  double data type, enabling 103-107

www.EngineeringBooksPdf.com

conjugate gradient method 216
about 194, 195, 216
used, for solving SpMV 195-199
Connection Machine (CM-2) 254
constant memory 101
convolution
in 1D 157, 159
in 2D 159-162
convolution theory 156, 157
COO format 201
cratchpad memory 144
CRAY Y-MP computer 254
CSR format (Compressed Sparse Row)
about 201
used, for solving SpMV 208-215
CUDA 8

D

data
copying, between memory objects 64-71
data binning 141
data-driven sorting algorithms 221
data-independent sorting algorithms 221
data partitioning
work-items, using for 71-77
data storage formats, SpMV
Compressed Sparse Row (CSR) format 201
COO format 201
ELLPACK format 200
ELLPACK-R format 200
data transfer capabilities, OpenCL 1.1 44
data types, OpenCL
bool 80, 81
char 81
double 81
float 81
half 80, 81
int 81
intptr_t 80, 81
long 81
ptrdiff_t 80, 81
short 81
size_t 80, 81
uchar 81
uint 81
uintptr_t 80, 81
ulong 81
unsigned char 81
unsigned int 81
unsigned long 81
unsigned short 81
ushort 81
void 81
device fission 88
diagonal format (DIA) 201
double data type
enabling 103-107

E

edge detection algorithm 155
ELLPACK format 200, 204
ELLPACK-R format
about 200, 204
used, for solving SpMV 204-207
events, OpenCL
command events 61
host monitoring events 61
event-synchronization 62, 63
Execution Model, OpenCL architecture 11, 12

F

Fisher-Yates Shuffle(FYS) algorithm 132
floating-point functions
using 123-125
fma() function 123
frexp() function 123
Fused Multiply-Add (FMA) instruction 124

G

geometric functions
about 117
using 117-120
global memory
about 101
reducing, via shared memory data prefetching 187-191
golden reference implementation 140
GPRs (General-Purpose Registers) 89
Gram-Schimdt process/conjugation 198

www.EngineeringBooksPdf.com

H

half data type 80
half-precision data type 87
hash table 152
histogram
about 139
implementing, in C/C++ 139-142
implementing, in OpenCL 142-152
HISTOGRAM-KEYS algorithm 255
Hollerith machine 241
host events 62

I

image segmentation 155
implicit vectorization 91, 92
information
retrieving, about OpenCL buffer objects 50-53
retrieving, about OpenCL sub-buffer objects 58-60
Installable Client Driver (ICD) 58
Instruction Level Parallelism (ILP) 212
integer functions
about 120
using 120-123
Intel AVX (Advanced Vector Extensions) 114
Intel Math Kernel Library (Intel MKL) 212
intermediate language (IL) 96
intptr_t data type 80

K

kernels 88
key 244

L

least significant byte (LSB) 85
Least Significant Digit Radix sort. *See* **LSD Radix sort**
linear systems 118
local memory 101, 144
loop unrolling 111
LSD Radix sort
about 244
working 245, 246

M

mad() function 123
Makefile 71
malloc 44
matrix 194
matrix multiplication
about 174, 175
global memory, reducing via shared memory data prefetching 187-191
implementing 178-181
implementing, by thread coarsening 181-184
implementing, through register tiling 185-187
working 176-178
memory domains
conceptual diagrams 101
Memory Model, OpenCL architecture 13
memory objects
data, copying between 64-71
modf() function 123
most significant byte (MSB) 85
MSD Radix sort
about 244
working 244, 246
Multiply-Add (MAD) instruction 124
MXCSR register 129

N

NDRange 11
nextafter() function 123
non-adaptive sorting algorithm 222

O

OpenCL
about 7-10, 44
arithmetic operation 129-131
bitonic sorting, developing 230-239
buffer objects, creating 44-50
goals 7
histogram, implementing 142-152
implementation, of Sobel edge filter 162-168
matrix multiplication, implementing 178-181
matrix multiplication, implementing by thread coarsening 181-184
matrix multiplication, implementing through register tiling 185-187

www.EngineeringBooksPdf.com

profiling, implementing 169, 170
Radix sort, developing in 254-280
rounding operation 129-131
scalar data types, initializing 80-82
scalar data types, using 85-87
sub-buffer objects, creating 54-58
synchronization concept 153, 154
vector data types, initializing 82-84
vector data types, using 88-100
OpenCL 1.1
data transfer capabilities 44
OpenCL architecture
components 10
Execution Model 10-12
Memory Model 13
Platform Model 10
OpenCL buffer objects 44
OpenCL contexts
about 25
querying 25-29
OpenCL device extensions
cl_APPLE_gl_sharing 23
cl_khr_3d_image_writes 22
cl_khr_byte_addressable_store 23
cl_khr_d3d10_sharing 23
cl_khr_fp16 22
cl_khr_fp64 22
cl_khr_gl_event 23
cl_khr_global_int32_base_atomics 22
cl_khr_global_int32_extended_atomics 22
cl_khr_gl_sharing 23
cl_khr_int64_base_atomics 22
cl_khr_int64_extended_atomics 22
cl_khr_local_int32_base_atomics 22
cl_khr_local_int32_extended_atomics 22
querying for 22-25
OpenCL devices
querying, on platforms 18-22
OpenCL functions
floating-point functions 123-125
geometric functions 117-120
integer functions 120-123
select function 135-137
shuffle function 132-134
trigonometric functions 126-128
OpenCL histogram program
suggestions 143
OpenCL kernels
creating 35-38
enqueuing 38- 42
OpenCL platforms
querying 14-18
OpenCL program
querying 29-35
OpenCL programming model 103
OpenCL specification Version 1.2 13
Open Computing Language. *See* **OpenCL**
OSX Caveat 58

P

parallelizable routines 140
parallel reduction 207
parallel sorting network 224
Platform Model, OpenCL architecture 10
prefetching 115
prefix sums 256
processing elements (PEs) 9
profiling
about 168
in OpenCL 169, 170
ptrdiff_t data type 80

Q

QuickSort 221

R

Radix sort
about 242, 243
developing, in OpenCL 254-280
Radix sorting
reduction pattern 247-251
scan pattern 247
rand() function 133
random() function 133
RANK-AND-PERMUTE algorithm 256
reduction kernel 258
reduction pattern 247-251
register tiling
about 185

www.EngineeringBooksPdf.com

matrix multiplication, implementing through 185-187
reversed loop 253
rounding 129
rounding operation 129-131
round to nearest even rounding 131
rte-mode 118

S

scalar address space
examining 100-103
scalar data types, OpenCL
about 79
initializing 80-82
using 85-87
SCAN-BUCKETS algorithm 255
scan pattern 247
segmented reduction 207
select function
using, in OpenCL 135-137
shared memory data prefetching
global memory, reducing via 187-191
shuffle function
using, in OpenCL 132-134
shuffling 228
SIMD floating-point 129
Single Instruction Multiple Data (SIMD) 88
size_t data type 80
Sobel edge filter
implementing 162-168
Sobel filtering 156
Sobel operator 155
sorting 221
sorting algorithms 221
sorting methods
data-driven sorting algorithms 221
data-independent sorting algorithms 221
sorting networks
about 222
working 224
Sparse Matrix Vector Multiplication (SpMV)
data storage formats 199-203
solving, conjugate gradient method used 195-199
solving, CSR format used 208-215
solving, ELLPACK format used 204-207
solving, VexCL used 216-219
SpMV ELLPACK-R scalar kernel 204
SpMV ELLPACK-R vector kernel
about 205
facts 207
SSE2/3/4 114
SSE/SSE2 instructions 129
stable sorting 244
stdlib.h function 133
Streaming SIMD Extensions (SSE) 88
structs 45
sub-buffer objects, OpenCL
creating 54-58
information, retrieving about 58, 60
synchronization concept, OpenCL 153, 154

T

task parallelism 9
ternary selection 135
thread-based histograms 151
thread coarsening
matrix multiplication, implementing by 181-184
thread divergence 215
trigonometric functions
about 126
using 126-128
truncation function 131

U

uintptr_t data type 80
unshuffling 228
up-sweep kernel 258

V

vector address space
examining 100-103
vector data types, OpenCL
about 80
initializing 82-84
using 88-100
vectors
loading, from array 114-117
storing, to array 110-114

www.EngineeringBooksPdf.com

VexCL
used, for solving SpMV 216-219
vloadN functions 114
volatile keyword 111
vstoreN function 114

W

warp 121
wavefront 121
wavefront-/warp-level programming 204
work-groups 71
work-items
about 71
used, for data partitioning 71-77

X

x87 FPU control register 129

www.EngineeringBooksPdf.com

Thank you for buying
OpenCL Parallel Programming Development Cookbook

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

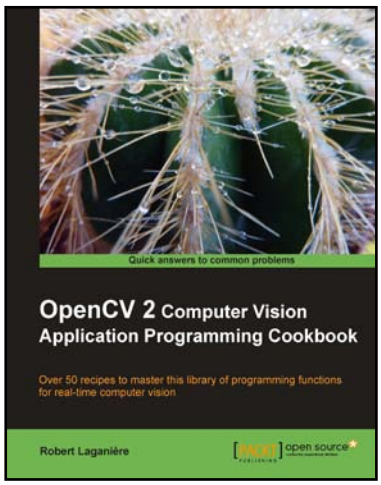
Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

www.EngineeringBooksPdf.com

[PACKT] PUBLISHING



OpenCV 2 Computer Vision Application Programming Cookbook

ISBN: 978-1-84951-324-1 Paperback: 304 pages

Over 50 recipes to master this library of programming functions for real-time computer vision

1. Teaches you how to program computer vision applications in C++ using the different features of the OpenCV library
2. Demonstrates the important structures and functions of OpenCV in detail with complete working examples
3. Describes fundamental concepts in computer vision and image processing
4. Gives you advice and tips to create more effective object-oriented computer vision programs

---

OpenGL 4.0 Shading Language Cookbook

ISBN: 978-1-84951-476-7 Paperback: 340 pages

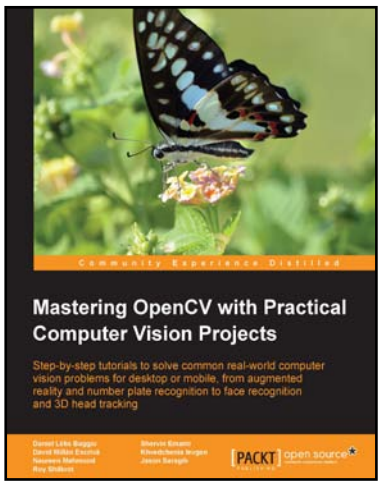
Over 60 highly focused, practical recipes to maximize your use of the OpenGL Shading language

1. A full set of recipes demonstrating simple and advanced techniques for producing high-quality, real-time 3D graphics using GLSL 4.0
2. How to use the OpenGL Shading Language to implement lighting and shading techniques
3. Use the new features of GLSL 4.0 including tessellation and geometry shaders

Please check **www.PacktPub.com** for information on our titles

www.EngineeringBooksPdf.com

[PACKT]
PUBLISHING

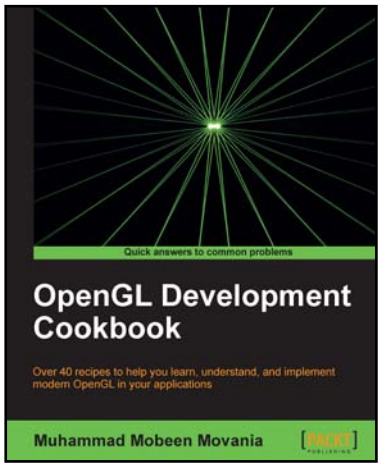


Mastering OpenCV with Practical Computer Vision Projects

ISBN: 978-1-84951-782-9 Paperback: 340 pages

Step-by-step tutorials to solve comman real-world computer vision problems for desktop or mobile, from augmented reality and number plate recognition to face recognition and 3D head tracking

1. Allows anyone with basic OpenCV experience to rapidly obtain skills in many computer vision topics, for research or commercial use
2. Each chapter is a separate project covering a computer vision problem, written by a professional with proven experience on that topic
3. All projects include a step-by-step tutorial and full source-code, using the C++ interface of OpenCV



OpenGL Development Cookbook

ISBN: 978-1-84969-504-6 Paperback: 326 pages

Over 40 recipes to help you learn, understand, and implement modern OpenGL in your applications

1. Explores current graphics programming techniques including GPU-based methods from the outlook of modern OpenGL 3.3
2. Includes GPU-based volume rendering algorithms
3. Discover how to employ GPU-based path and ray tracing

Please check **www.PacktPub.com** for information on our titles

www.EngineeringBooksPdf.com

www.EngineeringBooksPdf.com