OpenCL Programming Guide

Aaftab Munshi • Benedict R. Gaster
Timothy G. Mattson • James Fung • Dan Ginsburg

Foreword by Professor Pat Hanrahan, Stanford University

OpenCL Programming Guide

OpenGL® Series

Visit **informit.com/opengl** for a complete list of available products

---

The OpenGL graphics system is a software interface to graphics hardware. ("GL" stands for "Graphics Library.") It allows you to create interactive programs that produce color images of moving, three-dimensional objects. With OpenGL, you can control computer-graphics technology to produce realistic pictures, or ones that depart from reality in imaginative ways.

The **OpenGL Series** from Addison-Wesley Professional comprises tutorial and reference books that help programmers gain a practical understanding of OpenGL standards, along with the insight needed to unlock OpenGL's full potential.

PEARSON

Addison-Wesley Cisco Press EXAM/CRAM IBM Press QUE PRENTICE HALL SAMS | Safari Books Online

OpenCL Programming Guide

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

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data

OpenCL programming guide / Aaftab Munshi ... [et al.].
p. cm.
Includes index.
ISBN-13: 978-0-321-74964-2 (pbk. : alk. paper)
ISBN-10: 0-321-74964-2 (pbk. : alk. paper)
1. OpenCL (Computer program language) I. Munshi, Aaftab.
QA76.73.O213O64 2012
005.2'75—dc23

2011016523

ISBN-13: 978-0-321-74964-2
ISBN-10: 0-321-74964-2
Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.
Second printing, September 2011

Editor-in-Chief
Mark Taub

Acquisitions Editor
Debra Williams Cauley

Development Editor
Michael Thurston

Managing Editor
John Fuller

Project Editor
Anna Popick

Copy Editor
Barbara Wood

Indexer
Jack Lewis

Proofreader
Lori Newhouse

Technical Reviewers
Andrew Brownsword
Yahya H. Mizra
Dave Shreiner

Publishing Coordinator
Kim Boedigheimer

Cover Designer
Alan Clements

Compositor
The CIP Group

Contents

Figures

Tables

Listings

Foreword

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

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

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

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

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

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

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

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

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

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

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

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

—*Pat Hanrahan*
Stanford University

This page intentionally left blank

Preface

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

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

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

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

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

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

Intended Audience

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

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

Organization of the Book

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

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

Part I: The OpenCL 1.1 Language and API

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

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

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

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

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

Part II: OpenCL 1.1 Case Studies

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

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

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

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

Appendix

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

Example Code

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

Errata

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

This page intentionally left blank

Acknowledgments

From Aaftab Munshi

It has been a great privilege working with Ben, Dan, Tim, and James on this book. I want to thank our reviewers, Andrew Brownsword, Yahya H. Mizra, Dave Shreiner, and Michael Thurston, who took the time to review this book and provided valuable feedback that has improved the book tremendously. I want to thank our editor at Pearson, Debra Williams Cauley, for all her help in making this book happen.

I also want to thank my daughters, Hannah and Ellie, and the love of my life, Karen, without whom this book would not be possible.

From Benedict R. Gaster

I would like to thank AMD for supporting my work on OpenCL. There are four people in particular who have guided my understanding of the GPGPU revolution: Mike Houston, Justin Hensley, Lee Howes, and Laurent Morichetti.

This book would not have been possible without the continued enjoyment of life in Santa Cruz and going to the beach with Miranda, Maude, Polly, and Meg. Thanks!

From Timothy G. Mattson

I would like to thank Intel for giving me the freedom to pursue work on OpenCL. In particular, I want to thank Aaron Lefohn of Intel for bringing me into this project in the early days as it was just getting started. Most of all, however, I want to thank the amazing people in the OpenCL working group. I have learned a huge amount from this dedicated team of professionals.

From James Fung

It's been a privilege to work alongside my coauthors and contribute to this book. I would also like to thank NVIDIA for all its support during writing as well as family and friends for their support and encouragement.

From Dan Ginsburg

I would like to thank Dr. Rudolph Pienaar and Dr. Ellen Grant at Children's Hospital Boston for supporting me in writing this book and for their valuable contributions and insights. It has been an honor and a great privilege to work on this book with Affie, Ben, Tim, and James, who represent some of the sharpest minds in the parallel computing business. I also want to thank our editor, Debra Williams Cauley, for her unending patience and dedication, which were critical to the success of this project.

About the Authors

Aaftab Munshi is the spec editor for the OpenGL ES 1.1, OpenGL ES 2.0, and OpenCL specifications and coauthor of the book *OpenGL ES 2.0 Programming Guide* (with Dan Ginsburg and Dave Shreiner, published by Addison-Wesley, 2008). He currently works at Apple.

Benedict R. Gaster is a software architect working on programming models for next-generation heterogeneous processors, in particular looking at high-level abstractions for parallel programming on the emerging class of processors that contain both CPUs and accelerators such as GPUs. Benedict has contributed extensively to the OpenCL's design and has represented AMD at the Khronos Group open standard consortium. Benedict has a Ph.D. in computer science for his work on type systems for extensible records and variants. He has been working at AMD since 2008.

Timothy G. Mattson is an old-fashioned parallel programmer, having started in the mid-eighties with the Caltech Cosmic Cube and continuing to the present. Along the way, he has worked with most classes of parallel computers (vector supercomputers, SMP, VLIW, NUMA, MPP, clusters, and many-core processors). Tim has published extensively, including the books *Patterns for Parallel Programming* (with Beverly Sanders and Berna Massingill, published by Addison-Wesley, 2004) and *An Introduction to Concurrency in Programming Languages* (with Matthew J. Sottile and Craig E Rasmussen, published by CRC Press, 2009). Tim has a Ph.D. in chemistry for his work on molecular scattering theory. He has been working at Intel since 1993.

James Fung has been developing computer vision on the GPU as it progressed from graphics to general-purpose computation. James has a Ph.D. in electrical and computer engineering from the University of Toronto and numerous IEEE and ACM publications in the areas of parallel GPU Computer Vision and Mediated Reality. He is currently a Developer Technology Engineer at NVIDIA, where he examines computer vision and image processing on graphics hardware.

Dan Ginsburg currently works at Children's Hospital Boston as a Principal Software Architect in the Fetal-Neonatal Neuroimaging and Development Science Center, where he uses OpenCL for accelerating

neuroimaging algorithms. Previously, he worked for Still River Systems developing GPU-accelerated image registration software for the Monarch 250 proton beam radiotherapy system. Dan was also Senior Member of Technical Staff at AMD, where he worked for over eight years in a variety of roles, including developing OpenGL drivers, creating desktop and hand-held 3D demos, and leading the development of handheld GPU developer tools. Dan holds a B.S. in computer science from Worcester Polytechnic Institute and an M.B.A. from Bentley University.

Programming with OpenCL C

The OpenCL C programming language is used to create programs that describe data-parallel kernels and tasks that can be executed on one or more heterogeneous devices such as CPUs, GPUs, and other processors referred to as accelerators such as DSPs and the Cell Broadband Engine (B.E.) processor. An OpenCL program is similar to a dynamic library, and an OpenCL kernel is similar to an exported function from the dynamic library. Applications directly call the functions exported by a dynamic library from their code. Applications, however, cannot call an OpenCL kernel directly but instead queue the execution of the kernel to a command-queue created for a device. The kernel is executed asynchronously with the application code running on the host CPU.

OpenCL C is based on the ISO/IEC 9899:1999 C language specification (referred to in short as C99) with some restrictions and specific extensions to the language for parallelism. In this chapter, we describe how to write data-parallel kernels using OpenCL C and cover the features supported by OpenCL C.

Writing a Data-Parallel Kernel Using OpenCL C

As described in Chapter 1, data parallelism in OpenCL is expressed as an *N*-dimensional computation domain, where *N* = 1, 2, or 3. The N-D domain defines the total number of work-items that can execute in parallel. Let's look at how a data-parallel kernel would be written in OpenCL C by taking a simple example of summing two arrays of floats. A sequential version of this code would perform the sum by summing individual elements of both arrays inside a for loop:

```
void
scalar_add (int n, const float *a, const float *b, float *result)
{
    int i;
```

```
    for (i=0; i<n; i++)
        result[i] = a[i] + b[i];
}
```

A data-parallel version of the code in OpenCL C would look like this:

```
kernel void
scalar_add (global const float *a,
            global const float *b,
            global float *result)
{
    int id = get_global_id(0);
    result[id] = a[id] + b[id];
}
```

The `scalar_add` function declaration uses the `kernel` qualifier to indicate that this is an OpenCL C kernel. Note that the `scalar_add` kernel includes only the code to compute the sum of each individual element, aka the inner loop. The N-D domain will be a one-dimensional domain set to `n`. The kernel is executed for each of the `n` work-items to produce the sum of arrays `a` and `b`. In order for this to work, each executing work-item needs to know which individual elements from arrays `a` and `b` need to be summed. This must be a unique value for each work-item and should be derived from the N-D domain specified when queuing the kernel for execution. The `get_global_id(0)` returns the one-dimensional global ID for each work-item. Ignore the `global` qualifiers specified in the kernel for now; they will be discussed later in this chapter.

Figure 4.1 shows how `get_global_id` can be used to identify a unique work-item from the list of work-items executing a kernel.

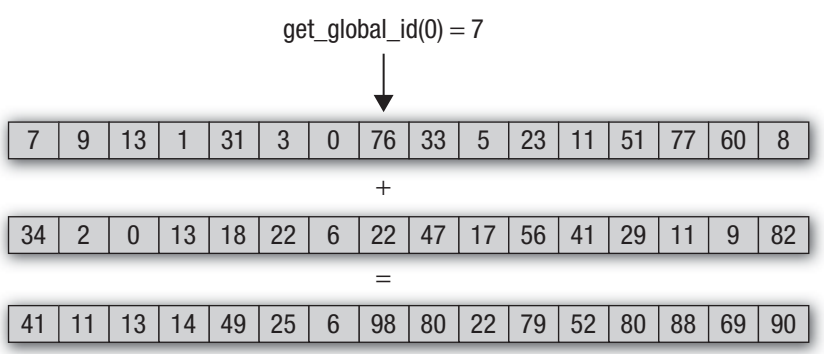

Figure 4.1 Mapping `get_global_id` to a work-item

The OpenCL C language with examples is described in depth in the sections that follow. The language is derived from C99 with restrictions that are described at the end of this chapter.

OpenCL C also adds the following features to C99:

- **Vector data types.** A number of OpenCL devices such as Intel SSE, AltiVec for POWER and Cell, and ARM NEON support a vector instruction set. This vector instruction set is accessed in C/C++ code through built-in functions (some of which may be device-specific) or device-specific assembly instructions. In OpenCL C, vector data types can be used in the same way scalar types are used in C. This makes it much easier for developers to write vector code because similar operators can be used for both vector and scalar data types. It also makes it easy to write portable vector code because the OpenCL compiler is now responsible for mapping the vector operations in OpenCL C to the appropriate vector ISA for a device. Vectorizing code also helps improve memory bandwidth because of regular memory accesses and better coalescing of these memory accesses.
- **Address space qualifiers.** OpenCL devices such as GPUs implement a memory hierarchy. The address space qualifiers are used to identify a specific memory region in the hierarchy.
- **Additions to the language for parallelism.** These include support for work-items, work-groups, and synchronization between work-items in a work-group.
- **Images.** OpenCL C adds image and sampler data types and built-in functions to read and write images.
- **An extensive set of built-in functions such as math, integer, geometric, and relational functions.** These are described in detail in Chapter 5.

Scalar Data Types

The C99 scalar data types supported by OpenCL C are described in Table 4.1. Unlike C, OpenCL C describes the sizes, that is, the exact number of bits for the integer and floating-point data types.

Table 4.1 Built-In Scalar Data Types

Type	Description
`bool`	A conditional data type that is either true or false. The value `true` expands to the integer constant `1`, and the value `false` expands to the integer constant `0`.
`char`	A signed two's complement 8-bit integer.
`unsigned char, uchar`	An unsigned 8-bit integer.
`short`	A signed two's complement 16-bit integer.
`unsigned short, ushort`	An unsigned 16-bit integer.
`int`	A signed two's complement 32-bit integer.
`unsigned int, uint`	An unsigned 32-bit integer.
`long`	A signed two's complement 64-bit integer.
`unsigned long, ulong`	An unsigned 64-bit integer.
`float`	A 32-bit floating-point. The float data type must conform to the IEEE 754 single-precision storage format.
`double`	A 64-bit floating-point. The `double` data type must conform to the IEEE 754 double-precision storage format. This is an optional format and is available only if the double-precision extension (`cl_khr_fp64`) is supported by the device.
`half`	A 16-bit floating-point. The `half` data type must conform to the IEEE 754-2008 half-precision storage format.
`size_t`	The unsigned integer type of the result of the `sizeof` operator. This is a 32-bit unsigned integer if the address space of the device is 32 bits and is a 64-bit unsigned integer if the address space of the device is 64 bits.
`ptrdiff_t`	A signed integer type that is the result of subtracting two pointers. This is a 32-bit signed integer if the address space of the device is 32 bits and is a 64-bit signed integer if the address space of the device is 64 bits.
`intptr_t`	A signed integer type with the property that any valid pointer to `void` can be converted to this type, then converted back to a pointer to `void`, and the result will compare equal to the original pointer.

Table 4.1 Built-In Scalar Data Types (*Continued*)

Type	Description
`uintptr_t`	An unsigned integer type with the property that any valid pointer to `void` can be converted to this type, then converted back to a pointer to `void`, and the result will compare equal to the original pointer.
`void`	The `void` type constitutes an empty set of values; it is an incomplete type that cannot be completed.

The `half` Data Type

The `half` data type must be IEEE 754-2008-compliant. `half` numbers have 1 sign bit, 5 exponent bits, and 10 mantissa bits. The interpretation of the sign, exponent, and mantissa is analogous to that of IEEE 754 floating-point numbers. The exponent bias is 15. The `half` data type must represent finite and normal numbers, denormalized numbers, infinities, and NaN. Denormalized numbers for the `half` data type, which may be generated when converting a `float` to a `half` using the built-in function `vstore_half` and converting a `half` to a `float` using the built-in function `vload_half`, cannot be flushed to zero.

Conversions from `float` to `half` correctly round the mantissa to 11 bits of precision. Conversions from `half` to `float` are lossless; all `half` numbers are exactly representable as `float` values.

The `half` data type can be used only to declare a pointer to a buffer that contains `half` values. A few valid examples are given here:

```
void
bar(global half *p)
{
    ...
}

void
foo(global half *pg, local half *pl)
{
    global half *ptr;
    int offset;

    ptr = pg + offset;
    bar(ptr);
}
```

Following is an example that is not a valid usage of the `half` type:

```
half a;
half a[100];

half *p;
a = *p;    // not allowed. must use vload_half function
```

Loads from a pointer to a `half` and stores to a pointer to a `half` can be performed using the `vload_half`, `vload_half`*n*, `vloada_half`*n* and `vstore_half`, `vstore_half`*n*, and `vstorea_half`*n* functions, respectively. The load functions read scalar or vector `half` values from memory and convert them to a scalar or vector float value. The `store` functions take a scalar or vector float value as input, convert it to a `half` scalar or vector value (with appropriate rounding mode), and write the `half` scalar or vector value to memory.

Vector Data Types

For the scalar integer and floating-point data types described in Table 4.1, OpenCL C adds support for vector data types. The vector data type is defined with the type name, that is, `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `float`, `long`, or `ulong` followed by a literal value *n* that defines the number of elements in the vector. Supported values of *n* are 2, 3, 4, 8, and 16 for all vector data types. Optionally, vector data types are also defined for `double` and `half`. These are available only if the device supports the double-precision and half-precision extensions. The supported vector data types are described in Table 4.2.

Variables declared to be a scalar or vector data type are always aligned to the size of the data type used in bytes. Built-in data types must be aligned to a power of 2 bytes in size. A built-in data type that is not a power of 2 bytes in size must be aligned to the next-larger power of 2. This rule does not apply to structs or unions.

For example, a `float4` variable will be aligned to a 16-byte boundary and a `char2` variable will be aligned to a 2-byte boundary. For 3-component vector data types, the size of the data type is 4 × `sizeof`(component). This means that a 3-component vector data type will be aligned to a 4 × `sizeof`(component) boundary.

The OpenCL compiler is responsible for aligning data items appropriately as required by the data type. The only exception is for an argument to a

Table 4.2 Built-In Vector Data Types

Type	Description
`char`*n*	A vector of n 8-bit signed integer values
`uchar`*n*	A vector of n 8-bit unsigned integer values
`short`*n*	A vector of n 16-bit signed integer values
`ushort`*n*	A vector of n 16-bit unsigned integer values
`int`*n*	A vector of n 32-bit signed integer values
`uint`*n*	A vector of n 32-bit unsigned integer values
`long`*n*	A vector of n 64-bit signed integer values
`ulong`*n*	A vector of n 64-bit unsigned integer values
`float`*n*	A vector of n 32-bit floating-point values
`double`*n*	A vector of n 64-bit floating-point values
`half`*n*	A vector of n 16-bit floating-point values

kernel function that is declared to be a pointer to a data type. For such functions, the compiler can assume that the pointee is always appropriately aligned as required by the data type.

For application convenience and to ensure that the data store is appropriately aligned, the data types listed in Table 4.3 are made available to the application.

Table 4.3 Application Data Types

Type in OpenCL Language	API Type for Application
`char`	`cl_char`
`uchar`	`cl_uchar`
`short`	`cl_short`
`ushort`	`cl_ushort`
`int`	`cl_int`

continues

Table 4.3 Application Data Types (*Continued*)

Type in OpenCL Language	API Type for Application
uint	cl_uint
long	cl_long
ulong	cl_ulong
float	cl_float
double	cl_double
half	cl_half
char*n*	cl_char*n*
uchar*n*	cl_uchar*n*
short*n*	cl_short*n*
ushort*n*	cl_ushort*n*
int*n*	cl_int*n*
uint*n*	cl_uint*n*
long*n*	cl_long*n*
ulong*n*	cl_ulong*n*
float*n*	cl_float*n*
double*n*	cl_double*n*
half*n*	cl_half*n*

Vector Literals

Vector literals can be used to create vectors from a list of scalars, vectors, or a combination of scalar and vectors. A vector literal can be used either as a vector initializer or as a primary expression. A vector literal cannot be used as an l-value.

A vector literal is written as a parenthesized vector type followed by a parenthesized comma-delimited list of parameters. A vector literal operates as an overloaded function. The forms of the function that are available are the set of possible argument lists for which all arguments have

the same element type as the result vector, and the total number of elements is equal to the number of elements in the result vector. In addition, a form with a single scalar of the same type as the element type of the vector is available. For example, the following forms are available for `float4`:

```
(float4)( float, float, float, float )
(float4)( float2, float, float )
(float4)( float, float2, float )
(float4)( float, float, float2 )
(float4)( float2, float2 )
(float4)( float3, float )
(float4)( float, float3 )
(float4)( float )
```

Operands are evaluated by standard rules for function evaluation, except that no implicit scalar widening occurs. The operands are assigned to their respective positions in the result vector as they appear in memory order. That is, the first element of the first operand is assigned to `result.x`, the second element of the first operand (or the first element of the second operand if the first operand was a scalar) is assigned to `result.y`, and so on. If the operand is a scalar, the operand is replicated across all lanes of the result vector.

The following example shows a vector `float4` created from a list of scalars:

```
float4  f = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
```

The following example shows a vector `uint4` created from a scalar, which is replicated across the components of the vector:

```
uint4   u = (uint4)(1); // u will be (1, 1, 1, 1)
```

The following examples show more complex combinations of a vector being created using a scalar and smaller vector types:

```
float4  f = (float4)((float2)(1.0f, 2.0f), (float2)(3.0f, 4.0f));
float4  f = (float4)(1.0f, (float2)(2.0f, 3.0f), 4.0f);
```

The following examples describe how *not* to create vector literals. All of these examples should result in a compilation error.

```
float4  f = (float4)(1.0f, 2.0f);
float4  f = (float2)(1.0f, 2.0f);
float4  f = (float4)(1.0f, (float2)(2.0f, 3.0f));
```

Vector Components

The components of vector data types with 1 to 4 components (aka elements) can be addressed as `<vector>.xyzw`. Table 4.4 lists the components that can be accessed for various vector types.

Table 4.4 Accessing Vector Components

Vector Data Types	Accessible Components
`char2, uchar2, short2, ushort2, int2, uint2, long2, ulong2, float2`	`.xy`
`char3, uchar3, short3, ushort3, int3, uint3, long3, ulong3, float3`	`.xyz`
`char4, uchar4, short4, ushort4, int4, uint4, long4, ulong4, float4`	`.xyzw`
`double2, half2`	`.xy`
`double3, half3`	`.xyz`
`double4, half4`	`.xyzw`

Accessing components beyond those declared for the vector type is an error. The following describes legal and illegal examples of accessing vector components:

```
float2 pos;
pos.x = 1.0f; // is legal
pos.z = 1.0f; // is illegal

float3 pos;
pos.z = 1.0f; // is legal
pos.w = 1.0f; // is illegal
```

The component selection syntax allows multiple components to be selected by appending their names after the period (.). A few examples that show how to use the component selection syntax are given here:

```
float4 c;

c.xyzw = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
c.z = 1.0f;
c.xy = (float2)(3.0f, 4.0f);
c.xyz = (float3)(3.0f, 4.0f, 5.0f);
```

The component selection syntax also allows components to be permuted or replicated as shown in the following examples:

```
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 = pox.xxyy;  // dup = (1.0f, 1.0f, 2.0f, 2.0f)
```

Vector components can also be accessed using a numeric index to refer to the appropriate elements in the vector. The numeric indices that can be used are listed in Table 4.5.

Table 4.5 Numeric Indices for Built-In Vector Data Types

Vector Components	Usable Numeric Indices
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

All numeric indices must be preceded by the letter `s` or `S`. In the following example `f.s0` refers to the first element of the `float8` variable `f` and `f.s7` refers to the eighth element of the `float8` variable `f`:

```
float8 f
```

In the following example `x.sa` (or `x.sA`) refers to the eleventh element of the `float16` variable `x` and `x.sf` (or `x.sF`) refers to the sixteenth element of the `float16` variable `x`:

```
float16 x
```

The numeric indices cannot be intermixed with the `.xyzw` notation. For example:

```
float4 f;
float4 v_A = f.xs123;  // is illegal
float4 v_B = f.s012w;  // is illegal
```

Vector data types can use the `.lo` (or `.odd`) and `.hi` (or `.even`) suffixes to get smaller vector types or to combine smaller vector types into a larger

vector type. Multiple levels of `.lo` (or `.odd`) and `.hi` (or `.even`) suffixes can be used until they refer to a scalar type.

The `.lo` suffix refers to the lower half of a given vector. The `.hi` suffix refers to the upper half of a given vector. The `.odd` suffix refers to the odd elements of a given vector. The `.even` suffix refers to the even elements of a given vector. Some examples to illustrate this concept are given here:

```
float4 vf;

float2 low = vf.lo;     // returns vf.xy
float2 high = vf.hi;    // returns vf.zw
float x = low.low;      // returns low.x
float y = low.hi;       // returns low.y

float2 odd = vf.odd;    // returns vf.yw
float2 even = vf.even; // returns vf.xz
```

For a 3-component vector, the suffixes `.lo` (or `.odd`) and `.hi` (or `.even`) operate as if the 3-component vector were a 4-component vector with the value in the `w` component undefined.

Other Data Types

The other data types supported by OpenCL C are described in Table 4.6.

Table 4.6 Other Built-In Data Types

Type	Description
`image2d_t`	A 2D image type.
`image3d_t`	A 3D image type.
`sampler_t`	An image sampler type.
`event_t`	An event type. These are used by built-in functions that perform async copies from global to local memory and vice versa. Each async copy operation returns an event and takes an event to wait for that identifies a previous async copy operation.

There are a few restrictions on the use of image and sampler types:

- The image and samplers types are defined only if the device supports images.
- Image and sampler types cannot be declared as arrays. Here are a couple of examples that show these illegal use cases:

```
kernel void
foo(image2d_t imgA[10]) // error. images cannot be declared
                        //        as arrays
{
    image2d_t imgB[4];  // error. images cannot be declared
                        //        as arrays
    ...
}


kernel void
foo(sampler_t smpA[10]) // error. samplers cannot be declared
                        //        as arrays
{
    sampler_t smpB[4];  // error. samplers cannot be declared
                        //        as arrays
    ...
}
```

- The `image2d_t`, `image3d_t`, and `sampler_t` data types cannot be declared in a struct.
- Variables cannot be declared to be pointers of `image2d_t`, `image3d_t`, and `sampler_t` data types.

Derived Types

The C99 derived types (arrays, structs, unions, and pointers) constructed from the built-in data types described in Tables 4.1 and 4.2 are supported. There are a few restrictions on the use of derived types:

- The struct type cannot contain any pointers if the struct or pointer to a struct is used as an argument type to a kernel function. For example, the following use case is invalid:

```
typedef struct {
    int  x;
    global float *f;
} mystruct_t;
```

```
kernel void
foo(global mystruct_t *p) // error. mystruct_t contains
                          //        a pointer
{
    ...
}
```

- The struct type can contain pointers only if the struct or pointer to a struct is used as an argument type to a non-kernel function or declared as a variable inside a kernel or non-kernel function. For example, the following use case is valid:

```
void
my_func(mystruct_t *p)
{
    ...
}


kernel void
foo(global int *p1, global float *p2)
{
    mystruct_t s;

    s.x = p1[get_global_id(0)];
    s.f = p2;
    my_func(&s);
}
```

Implicit Type Conversions

Implicit type conversion is an automatic type conversion done by the compiler whenever data from different types is intermixed. Implicit conversions of scalar built-in types defined in Table 4.1 (except `void`, `double`,[1] and `half`[2]) are supported. When an implicit conversion is done, it is not just a reinterpretation of the expression's value but a conversion of that value to an equivalent value in the new type.

Consider the following example:

```
float f = 3;      // implicit conversion to float value 3.0
int   i = 5.23f;  // implicit conversion to integer value 5
```

[1] Unless the double-precision extension (`cl_khr_fp64`) is supported by the device.

[2] Unless the half-precision extension (`cl_khr_fp16`) is supported by the device.

In this example, the value `3` is converted to a `float` value `3.0f` and then assigned to `f`. The value `5.23f` is converted to an `int` value `5` and then assigned to `i`. In the second example, the fractional part of the `float` value is dropped because integers cannot support fractional values; this is an example of an unsafe type conversion.

Warning Note that some type conversions are inherently unsafe, and if the compiler can detect that an unsafe conversion is being implicitly requested, it will issue a warning.

Implicit conversions for pointer types follow the rules described in the C99 specification. Implicit conversions between built-in vector data types are disallowed. For example:

```
float4 f;
int4   i;

f = i;  // illegal implicit conversion between vector data types
```

There are graphics shading languages such as OpenGL Shading Language (GLSL) and the DirectX Shading Language (HLSL) that do allow implicit conversions between vector types. However, prior art for vector casts in C doesn't support conversion casts. The *AltiVec Technology Programming Interface Manual* (www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf?fsrch=1), Section 2.4.6, describes the function of casts between vector types. The casts are conversion-free. Thus, any conforming AltiVec compiler has this behavior. Examples include XL C, GCC, MrC, Metrowerks, and Green Hills. IBM's Cell SPE C language extension (*C/C++ Language Extensions for Cell Broadband Engine Architecture*; see Section 1.4.5) has the same behavior. GCC and ICC have adopted the conversion-free cast model for SSE (http://gcc.gnu.org/onlinedocs/gcc-4.2.4/gcc/Vector-Extensions.html#Vector-Extensions). The following code example shows the behavior of these compilers:

```
#include <stdio.h>

// Declare some vector types. This should work on most compilers
// that try to be GCC compatible. Alternatives are provided
// for those that don't conform to GCC behavior in vector
// type declaration.
// Here a vFloat is a vector of four floats, and
// a vInt is a vector of four 32-bit ints.
#if 1
    // This should work on most compilers that try
    // to be GCC compatible
    // cc main.c -Wall -pedantic
    typedef float vFloat __attribute__ ((__vector_size__(16)));
```

```
    typedef int   vInt   __attribute__ ((__vector_size__(16)));
    #define init_vFloat(a, b, c, d)    (const vFloat) {a, b, c, d}
#else
    //Not GCC compatible
    #if defined( __SSE2__ )
        // depending on compiler you might need to pass
        // something like -msse2 to turn on SSE2
        #include <emmintrin.h>
        typedef __m128  vFloat;
        typedef __m128i vInt;
        static inline vFloat init_vFloat(float a, float b,
                                         float c, float d);
        static inline vFloat init_vFloat(float a, float b,
                                         float c, float d)
        { union{ vFloat v; float f[4];}u;
          u.f[0] = a; u.f[1] = b;
          u.f[2] = c; u.f[3] = d;
          return u.v;
        }
    #elif defined( __VEC__ )
        // depending on compiler you might need to pass
        // something like -faltivec or -maltivec or
        // "Enable AltiVec Extensions" to turn this part on
        #include <altivec.h>
        typedef vector float vFloat;
        typedef vector int   vInt;

        #if 1
            // for compliant compilers
            #define init_vFloat(a, b, c, d) \
                     (const vFloat) (a, b, c, d)
        #else
            // for FSF GCC
            #define init_vFloat(a, b, c, d) \
                     (const vFloat) {a, b, c, d}
        #endif
    #endif
#endif

void
print_vInt(vInt v)
{
    union{ vInt v; int i[4]; }u;
    u.v = v;

    printf("vInt: 0x%8.8x 0x%8.8x 0x%8.8x 0x%8.8x\n",
                        u.i[0], u.i[1], u.i[2], u.i[3]);
}
```

```
void
print_vFloat(vFloat v)
{
    union{ vFloat v; float i[4]; }u;
    u.v = v;

    printf("vFloat: %f %f %f %f\n", u.i[0], u.i[1], u.i[2], u.i[3]);
}

int
main(void)
{
    vFloat  f = init_vFloat(1.0f, 2.0f, 3.0f, 4.0f);
    vInt    i;

    print_vFloat(f);

    printf("assign with cast:  vInt i = (vInt) f;\n" );
    i = (vInt) f;

    print_vInt(i);

    return 0;
}
```

The output of this code example demonstrates that conversions between vector data types implemented by some C compilers[3] such as GCC are cast-free.

```
vFloat: 1.000000 2.000000 3.000000 4.000000
assign with cast:  vInt i = (vInt) f;
vInt: 0x3f800000 0x40000000 0x40400000 0x40800000
```

So we have prior art in C where casts between vector data types do not perform conversions as opposed to graphics shading languages that do perform conversions. The OpenCL working group decided it was best to make implicit conversions between vector data types illegal. It turns out that this was the right thing to do for other reasons, as discussed in the section "Explicit Conversions" later in this chapter.

[3] Some fiddling with compiler flags to get the vector extensions turned on may be required, for example, `-msse2` or `-faltivec`. You might need to play with the `#if`s. The problem is that there is no portable way to declare a vector type. Getting rid of the sort of portability headaches at the top of the code example is one of the major value-adds of OpenCL.

Usual Arithmetic Conversions

Many operators that expect operands of arithmetic types (integer or floating-point types) cause conversions and yield result types in a similar way. The purpose is to determine a common real type for the operands and result. For the specified operands, each operand is converted, without change of type domain, to a type whose corresponding real type is the common real type. For this purpose, all vector types are considered to have a higher conversion rank than scalars. Unless explicitly stated otherwise, the common real type is also the corresponding real type of the result, whose type domain is the type domain of the operands if they are the same, and complex otherwise. This pattern is called the **usual arithmetic conversions**.

If the operands are of more than one vector type, then a compile-time error will occur. Implicit conversions between vector types are not permitted.

Otherwise, if there is only a single vector type, and all other operands are scalar types, the scalar types are *converted* to the type of the vector element, and then *widened* into a new vector containing the same number of elements as the vector, by duplication of the scalar value across the width of the new vector. A compile-time error will occur if any scalar operand has greater rank than the type of the vector element. For this purpose, the rank order is defined as follows:

1. The rank of a floating-point type is greater than the rank of another floating-point type if the floating-point type can exactly represent all numeric values in the second floating-point type. (For this purpose, the encoding of the floating-point value is used, rather than the subset of the encoding usable by the device.)
2. The rank of any floating-point type is greater than the rank of any integer type.
3. The rank of an integer type is greater than the rank of an integer type with less precision.
4. The rank of an unsigned integer type is greater than the rank of a signed integer type with the same precision.
5. `bool` has a rank less than any other type.
6. The rank of an enumerated type is equal to the rank of the compatible integer type.

7. For all types `T1`, `T2`, and `T3`, if `T1` has greater rank than `T2`, and `T2` has greater rank than `T3`, then `T1` has greater rank than `T3`.

Otherwise, if all operands are scalar, the usual arithmetic conversions apply as defined by Section 6.3.1.8 of the C99 specification.

Following are a few examples of legal usual arithmetic conversions with vectors and vector and scalar operands:

```
short a;
int4  b;
int4  c = b + a;
```

In this example, the variable `a`, which is of type `short`, is converted to an `int4` and the vector addition is then performed.

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

In the preceding example, the variable `a`, which is of type `int`, is converted to a `float4` and the vector addition is then performed.

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

In this example, no conversions need to be performed because `a`, `b`, and `c` are all the same type.

Here are a few examples of illegal usual arithmetic conversions with vectors and vector and scalar operands:

```
int    a;
short4 b;
short4 c = b + a; // cannot convert & widen int to short4

double a;
float4 b;
float4 c = b + a; // cannot convert & widen double to float4

int4   a;
float4 b;
float4 c = b + a; // cannot cast between different vector types
```

Explicit Casts

Standard type casts for the built-in scalar data types defined in Table 4.1 will perform appropriate conversion (except `void` and `half`[4]). In the next example, `f` stores `0x3F800000` and `i` stores `0x1`, which is the floating-point value `1.0f` in `f` converted to an integer value:

```
float f = 1.0f;
int   i = (int)f;
```

Explicit casts between vector types are not legal. The following examples will generate a compilation error:

```
int4   i;
uint4  u = (uint4)i;   // compile error

float4 f;
int4   i = (int4)f;    // compile error

float4 f;
int8   i = (int8)f;    // compile error
```

Scalar to vector conversions are performed by casting the scalar to the desired vector data type. Type casting will also perform the appropriate arithmetic conversion. Conversions to built-in integer vector types are performed with the round-toward-zero rounding mode. Conversions to built-in floating-point vector types are performed with the round-to-nearest rounding mode. When casting a `bool` to a vector integer data type, the vector components will be set to `-1` (that is, all bits are set) if the `bool` value is `true` and `0` otherwise.

Here are some examples of explicit casts:

```
float4 f = 1.0f;
float4 va = (float4)f;  // va is a float4 vector
                        // with elements ( f, f, f, f )

uchar u = 0xFF;
float4 vb = (float4)u;  // vb is a float4 vector with elements
                        // ( (float)u, (float)u,
                        //   (float)u, (float)u )

float f = 2.0f;
int2 vc = (int2)f;      // vc is an int2 vector with elements
                        // ( (int)f, (int)f )
```

[4] Unless the half-precision extension (`cl_khr_fp16`) is supported.

```
uchar4 vtrue =(uchar4)true;  // vtrue is a uchar4 vector with
                             // elements(0xFF, 0xFF, 0xFF, 0xFF)
```

Explicit Conversions

In the preceding sections we learned that implicit conversions and explicit casts do not allow conversions between vector types. However, there are many cases where we need to convert a vector type to another type. In addition, it may be necessary to specify the rounding mode that should be used to perform the conversion and whether the results of the conversion are to be saturated. This is useful for both scalar and vector data types.

Consider the following example:

```
float x;
int   i = (int)x;
```

In this example the value in `x` is truncated to an integer value and stored in `i`; that is, the cast performs round-toward-zero rounding when converting the floating-point value to an integer value.

Sometimes we need to round the floating-point value to the nearest integer. The following example shows how this is typically done:

```
float x;
int   i = (int)(x + 0.5f);
```

This works correctly for most values of `x` except when `x` is `0.5f - 1 ulp`[5] or if `x` is a negative number. When `x` is `0.5f - 1 ulp`, `(int)(x + 0.5f)` returns `1`; that is, it rounds up instead of rounding down. When `x` is a negative number, `(int)(x + 0.5f)` rounds down instead of rounding up.

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

int
main(void)
{
    float a = 0.5f;
    float b = a - nextafterf(a, (float)-INFINITY); // a - 1 ulp
```

[5] `ulp(x)` is the gap between two finite floating-point numbers. A detailed description of `ulp(x)` is given in Chapter 5 in the section "Math Functions," subsection "Relative Error as `ulp`s."

```
    printf("a = %8x, b = %8x\n",
                    *(unsigned int *)&a, *(unsigned int *)&b);
    printf("(int)(a + 0.5f) = %d \n", (int)(a + 0.5f));
    printf("(int)(b + 0.5f) = %d \n", (int)(b + 0.5f));
}

The printed values are:

a = 3f000000, b = 3effffff  // where b = a - 1 ulp.
(int)(a + 0.5f) = 1,
(int)(b + 0.5f) = 1
```

We could fix these issues by adding appropriate checks to see what value `x` is and then perform the correct conversion, but there is hardware to do these conversions with rounding and saturation on most devices. It is important from a performance perspective that OpenCL C allows developers to perform these conversions using the appropriate hardware ISA as opposed to emulating in software. This is why OpenCL implements built-in functions that perform conversions from one type to another with options that select saturation and one of four rounding modes.

Explicit conversions may be performed using either of the following:

```
destType convert_destType<_sat><_roundingMode> (sourceType)
destType convert_destTypen<_sat><_roundingMode> (sourceTypen)
```

These provide a full set of type conversions for the following scalar types: `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`,[6] `half`,[7] and the built-in vector types derived therefrom. The operand and result type must have the same number of elements. The operand and result type may be the same type, in which case the conversion has no effect on the type or value.

In the following example, `convert_int4` converts a `uchar4` vector `u` to an `int4` vector `c`:

```
uchar4 u;
int4   c = convert_int4(u);
```

In the next example, `convert_int` converts a `float` scalar `f` to an `int` scalar `i`:

```
float f;
int   i = convert_int(f);
```

---

[6] Unless the double-precision extension (`cl_khr_fp64`) is supported.

[7] Unless the half-precision extension (`cl_khr_fp16`) is supported.

Table 4.7 Rounding Modes for Conversions

Rounding Mode Modifier	Rounding Mode Description
`_rte`	Round to nearest even.
`_rtz`	Round toward zero.
`_rtp`	Round toward positive infinity.
`_rtn`	Round toward negative infinity.
No modifier specified	Use the default rounding mode for this destination type: `_rtz` for conversion to integers or `_rte` for conversion to floating-point types.

The optional rounding mode modifier can be set to one of the values described in Table 4.7.

The optional saturation modifier (`_sat`) can be used to specify that the results of the conversion must be saturated to the result type. When the conversion operand is either greater than the greatest representable destination value or less than the least representable destination value, it is said to be out of range. When converting between integer types, the resulting value for out-of-range inputs will be equal to the set of least significant bits in the source operand element that fits in the corresponding destination element. When converting from a floating-point type to an integer type, the behavior is implementation-defined.

Conversions to integer type may opt to convert using the optional saturated mode by appending the `_sat` modifier to the conversion function name. When in saturated mode, values that are outside the representable range clamp to the nearest representable value in the destination format. (`NaN` should be converted to `0`.)

Conversions to a floating-point type conform to IEEE 754 rounding rules. The `_sat` modifier may not be used for conversions to floating-point formats.

Following are a few examples of using explicit conversion functions.

The next example shows a conversion of a `float4` to a `ushort4` with round-to-nearest rounding mode and saturation. Figure 4.2 describes the values in `f` and the result of conversion in `c`.

```
float4  f = (float4)(-5.0f, 254.5f, 254.6f, 1.2e9f);

ushort4 c = convert_uchar4_sat_rte(f);
```

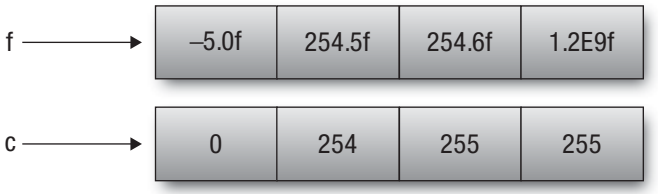

Figure 4.2 Converting a `float4` to a `ushort4` with round-to-nearest rounding and saturation

The next example describes the behavior of the saturation modifier when converting a signed value to an unsigned value or performing a down-conversion with integer types:

```
short4 s;

// negative values clamped to 0
ushort4 u = convert_ushort4_sat(s);

// values > CHAR_MAX converted to CHAR_MAX
// values < CHAR_MIN converted to CHAR_MIN
char4 c = convert_char4_sat(s);
```

The following example illustrates conversion from a floating-point to an integer with saturation and rounding mode modifiers:

```
float4 f;

// values implementation-defined for f > INT_MAX, f < INT_MAX, or
NaN
int4 i = convert_int4(f);

// values > INT_MAX clamp to INT_MAX,
// values < INT_MIN clamp to INT_MIN
// NaN should produce 0.
// The _rtz rounding mode is used to produce the integer values.
int4 i2 = convert_int4_sat(f);

// similar to convert_int4 except that floating-point values
// are rounded to the nearest integer instead of truncated
int4 i3 = convert_int4_rte(f);

// similar to convert_int4_sat except that floating-point values
// are rounded to the nearest integer instead of truncated
int4 i4 = convert_int4_sat_rte(f);
```

The final conversion example given here shows conversions from an integer to a floating-point value with and without the optional rounding mode modifier:

```
int4 i;

// convert ints to floats using the round-to-nearest rounding mode
float4 f = convert_float4(i);

// convert ints to floats; integer values that cannot be
// exactly represented as floats should round up to the next
// representable float
float4 f = convert_float4_rtp(i);
```

Reinterpreting Data as Another Type

Consider the case where you want to mask off the sign bit of a floating-point type. There are multiple ways to solve this in C—using pointer aliasing, unions, or `memcpy`. Of these, only `memcpy` is strictly correct in C99. Because OpenCL C does not support `memcpy`, we need a different method to perform this masking-off operation. The general capability we need is the ability to reinterpret bits in a data type as another data type. In the example where we want to mask off the sign bit of a floating-point type, we want to reinterpret these bits as an unsigned integer type and then mask off the sign bit. Other examples include using the result of a vector relational operator and extracting the exponent or mantissa bits of a floating-point type.

The `as_type` and `as_type`*n* built-in functions allow you to reinterpret bits of a data type as another data type of the same size. The `as_type` is used for scalar data types (except `bool` and `void`) and `as_type`*n* for vector data types. `double` and `half` are supported only if the appropriate extensions are supported by the implementation.

The following example describes how you would mask off the sign bit of a floating-point type using the `as_type` built-in function:

```
float f;
uint  u;

u = as_uint(f);
f = as_float(u & ~(1 << 31));
```

If the operand and result type contain the same number of elements, the bits in the operand are returned directly without modification as the new

type. If the operand and result type contain a different number of elements, two cases arise:

- The operand is a 4-component vector and the result is a 3-component vector. In this case, the `xyz` components of the operand and the result will have the same bits. The `w` component of the result is considered to be undefined.
- For all other cases, the behavior is implementation-defined.

We next describe a few examples that show how to use `as_type` and `as_type`*n*. The following example shows how to reinterpret an `int` as a `float`:

```
uint  u = 0x3f800000;
float f = as_float(u);
```

The variable `u`, which is declared as an unsigned integer, contains the value `0x3f800000`. This represents the single-precision floating-point value `1.0`. The variable `f` now contains the floating-point value `1.0`.

In the next example, we reinterpret a `float4` as an `int4`:

```
float4 f = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
int4 i = as_int4(f);
```

The variable `i`, defined to be of type `int4`, will have the following values in its `xyzw` components: `0x3f800000`, `0x40000000`, `0x40400000`, `0x40800000`.

The next example shows how we can perform the ternary selection operator (`?:`) for floating-point vector types using `as_type`*n*:

```
// Perform the operation f = f < g ? f : 0 for components of a
// vector
float4 f, g;
int4 is_less = f < g;

// Each component of the is_less vector will be 0 if result of <
// operation is false and will be -1 (i.e., all bits set) if
// the result of < operation is true.

f = as_float4(as_int4(f) & is_less);
// This basically selects f or 0 depending on the values in is_less.
```

The following example describes cases where the operand and result have a different number of results, in which case the behavior of `as_type` and `as_type`*n* is implementation-defined:

```
int i;
short2 j = as_short2(i); // Legal. Result is implementation-defined

int4 i;
short8 j = as_short8(i); // Legal. Result is implementation-defined

float4 f;
float3 g = as_float3(f); // Legal. g.xyz will have same values as
                         // f.xyz. g.w is undefined
```

This example describes reinterpreting a 4-component vector as a 3-component vector:

```
float4 f;
float3 g = as_float3(f); // Legal. g.xyz will have same values as
                         // f.xyz. g.w is undefined
```

The next example shows invalid ways of using `as_type` and `as_type`*n*, which should result in compilation errors:

```
float4 f;
double4 g = as_double4(f); // Error. Result and operand have
                           // different sizes.

float3 f;
float4 g = as_float4(f); // Error. Result and operand have
                         // different sizes
```

Vector Operators

Table 4.8 describes the list of operators that can be used with vector data types or a combination of vector and scalar data types.

Table 4.8 Operators That Can Be Used with Vector Data Types

Operator Category	Operator Symbols
Arithmetic operators	Add (+)
	Subtract (-)
	Multiply (*)
	Divide (/)
	Remainder (%)

continues

Table 4.8 Operators That Can Be Used with Vector Data Types (*Continued*)

Operator Category	Operator Symbols
Relational operators	Greater than (>) Less than (<) Greater than or equal (>=) Less than or equal (<=)
Equality operators	Equal (==) Not equal (!=)
Bitwise operators	And (&) Or (|) Exclusive or (^), not (~)
Logical operators	And (&&) Or (||)
Conditional operator	Ternary selection operator (?:)
Shift operators	Right shift (>>) Left shift (<<)
Unary operators	Arithmetic (+ or –) Post- and pre-increment (++) Post- and pre-decrement (--) `sizeof`, not (!) Comma operator (,) Address and indirection operators (&, *)
Assignment operators	=, *= , /= , += , -= , <<= , >>= , &= , ^= , |=

The behavior of these operators for scalar data types is as described by the C99 specification. The following sections discuss how each operator works with operands that are vector data types or vector and scalar data types.

Arithmetic Operators

The arithmetic operators—add (+), subtract (–), multiply (*), and divide (/)—operate on built-in integer and floating-point scalar and vector data types. The remainder operator (%) operates on built-in integer scalar and vector data types only. The following cases arise:

- The two operands are scalars. In this case, the operation is applied according to C99 rules.
- One operand is a scalar and the other is a vector. The scalar operand may be subject to the usual arithmetic conversion to the element type used by the vector operand and is then widened to a vector that has the same number of elements as the vector operand. The operation is applied component-wise, resulting in the same size vector.
- The two operands are vectors of the same type. In this case, the operation is applied component-wise, resulting in the same size vector.

For integer types, a divide by zero or a division that results in a value that is outside the range will not cause an exception but will result in an unspecified value. Division by zero for floating-point types will result in ±infinity or `NaN` as prescribed by the IEEE 754 standard.

A few examples will illustrate how the arithmetic operators work when one operand is a scalar and the other a vector, or when both operands are vectors.

The first example in Figure 4.3 shows two vectors being added:

```
int4 v_iA = (int4)(7, -3, -2, 5);
int4 v_iB = (int4)(1, 2, 3, 4);
int4 v_iC = v_iA + v_iB;
```

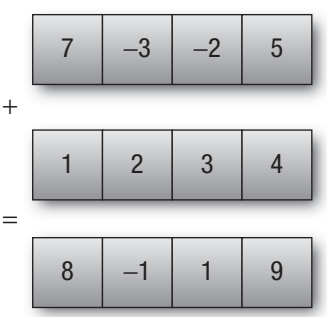

Figure 4.3 Adding two vectors

The result of the addition stored in vector `v_iC` is `(8, -1, 1, 9)`.

The next example in Figure 4.4 shows a multiplication operation where operands are a vector and a scalar. In this example, the scalar is just

widened to the size of the vector and the components of each vector are multiplied:

```
float4 vf = (float4)(3.0f, -1.0f, 1.0f, -2.0f);
float4 result = vf * 2.5f;
```

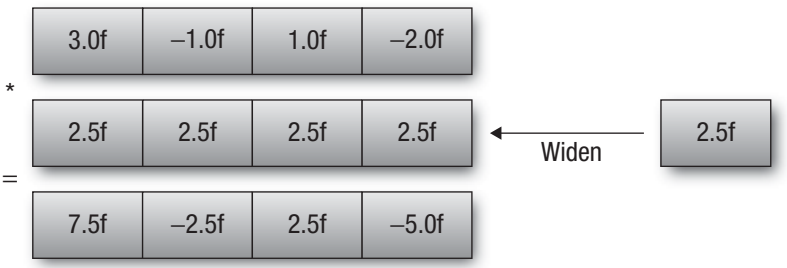

Figure 4.4 Multiplying a vector and a scalar with widening

The result of the multiplication stored in vector `result` is `(7.5f, -2.5f, 2.5f, -5.0f)`.

The next example in Figure 4.5 shows how we can multiply a vector and a scalar where the scalar is implicitly converted and widened:

```
float4 vf = (float4)(3.0f, -1.0f, 1.0f, -2.0f);
float4 result = vf * 2;
```

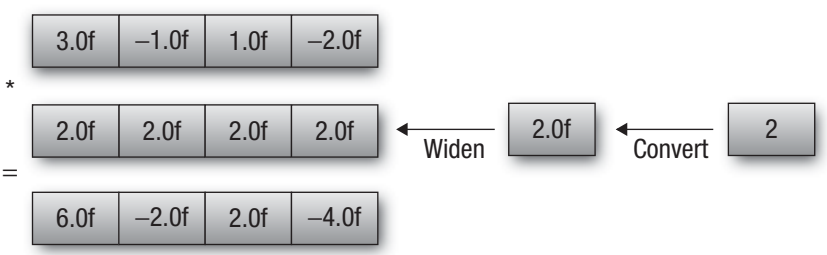

Figure 4.5 Multiplying a vector and a scalar with conversion and widening

The result of the multiplication stored in the vector `result` is `(6.0f, -2.0f, 2.0f, -4.0f)`.

Relational and Equality Operators

The relational operators—greater than (>), less than (<), greater than or equal (>=), and less than or equal (<=)—and equality operators—equal (==) and not equal (!=)—operate on built-in integer and floating-point scalar and vector data types. The result is an integer scalar or vector type. The following cases arise:

- The two operands are scalars. In this case, the operation is applied according to C99 rules.
- One operand is a scalar and the other is a vector. The scalar operand may be subject to the usual arithmetic conversion to the element type used by the vector operand and is then widened to a vector that has the same number of elements as the vector operand. The operation is applied component-wise, resulting in the same size vector.
- The two operands are vectors of the same type. In this case, the operation is applied component-wise, resulting in the same size vector.

The result is a scalar signed integer of type `int` if both source operands are scalar and a vector signed integer type of the same size as the vector source operand. The result is of type `charn` if the source operands are `charn` or `ucharn`; `shortn` if the source operands are `shortn`, `shortn`, or `halfn`; `intn` if the source operands are `intn`, `uintn`, or `floatn`; `longn` if the source operands are `longn`, `ulongn`, or `doublen`.

For scalar types, these operators return `0` if the specified relation is false and `1` if the specified relation is true. For vector types, these operators return `0` if the specified relation is false and `-1` (i.e., all bits set) if the specified relation is true. The relational operators always return `0` if one or both arguments are not a number (`NaN`). The equality operator equal (==) returns `0` if one or both arguments are not a number (`NaN`), and the equality operator not equal (!=) returns `1` (for scalar source operands) or `-1` (for vector source operands) if one or both arguments are not a number (`NaN`).

Bitwise Operators

The bitwise operators—and (&), or (|), exclusive or (^), and not (~)—operate on built-in integer scalar and vector data types. The result is an integer scalar or vector type. The following cases arise:

- The two operands are scalars. In this case, the operation is applied according to C99 rules.

- One operand is a scalar and the other is a vector. The scalar operand may be subject to the usual arithmetic conversion to the element type used by the vector operand and is then widened to a vector that has the same number of elements as the vector operand. The operation is applied component-wise, resulting in the same size vector.
- The two operands are vectors of the same type. In this case, the operation is applied component-wise, resulting in the same size vector.

Logical Operators

The logical operators—and (`&&`), or (`||`)—operate on built-in integer scalar and vector data types. The result is an integer scalar or vector type. The following cases arise:

- The two operands are scalars. In this case, the operation is applied according to C99 rules.
- One operand is a scalar and the other is a vector. The scalar operand may be subject to the usual arithmetic conversion to the element type used by the vector operand and is then widened to a vector that has the same number of elements as the vector operand. The operation is applied component-wise, resulting in the same size vector.
- The two operands are vectors of the same type. In this case, the operation is applied component-wise, resulting in the same size vector.

If both source operands are scalar, the logical operator and (`&&`) will evaluate the right-hand operand only if the left-hand operand compares unequal to `0`, and the logical operator or (`||`) will evaluate the right-hand operand only if the left-hand operand compares equal to `0`. If one or both source operands are vector types, both operands are evaluated.

The result is a scalar signed integer of type `int` if both source operands are scalar and a vector signed integer type of the same size as the vector source operand. The result is of type `charn` if the source operands are `charn` or `ucharn`; `shortn` if the source operands are `shortn` or `ushortn`; `intn` if the source operands are `intn` or `uintn`; or `longn` if the source operands are `longn` or `ulongn`.

For scalar types, these operators return `0` if the specified relation is false and `1` if the specified relation is true. For vector types, these operators return `0` if the specified relation is false and `-1` (i.e., all bits set) if the specified relation is true.

The logical exclusive operator (`^^`) is reserved for future use.

Conditional Operator

The ternary selection operator (`?:`) operates on three expressions (`expr1 ? expr2 : expr3`). This operator evaluates the first expression, `expr1`, which can be a scalar or vector type except the built-in floating-point types. If the result is a scalar value, the second expression, `expr2`, is evaluated if the result compares unequal to 0; otherwise the third expression, `expr3`, is evaluated. If the result is a vector value, then (`expr1 ? expr2 : expr3`) is applied component-wise and is equivalent to calling the built-in function `select(expr3, expr2, expr1)`. The second and third expressions can be any type as long as their types match or if an implicit conversion can be applied to one of the expressions to make their types match, or if one is a vector and the other is a scalar, in which case the usual arithmetic conversion followed by widening is applied to the scalar to match the vector operand type. This resulting matching type is the type of the entire expression.

A few examples will show how the ternary selection operator works with scalar and vector types:

```
int4   va, vb, vc, vd;
int    a, b, c, d;
float4 vf;

vc = d ? va : vb;  // vc = va if d is true, = vb if d is false

vc = vd ? va : vb; // vc.x = vd.x ? va.x : vb.x
                   // vc.y = vd.y ? va.y : vb.y
                   // vc.z = vd.z ? va.z : vb.z
                   // vc.w = vd.w ? va.w : vb.w

vc = vd ? a : vb;  // a is widened to an int4 first
                   // vc.x = vd.x ? a : vb.x
                   // vc.y = vd.y ? a : vb.y
                   // vc.z = vd.z ? a : vb.z
                   // vc.w = vd.w ? a : vb.w

vc = vd ? va : vf; // error – vector types va & vf do not match
```

Shift Operators

The shift operators—right shift (>>) and left shift (<<)—operate on built-in integer scalar and vector data types. The result is an integer scalar or vector type. The rightmost operand must be a scalar if the first operand is a scalar. For example:

```
uint  a, b, c;
uint2 r0, r1;

c = a << b;   // legal - both operands are scalars
r1 = a << r0; // illegal - first operand is a scalar and
              // therefore second operand (r0) must also be scalar.
c = b << r0;  // illegal - first operand is a scalar and
              // therefore second operand (r0) must also be scalar.
```

The rightmost operand can be a vector or scalar if the first operand is a vector. For vector types, the operators are applied component-wise.

If operands are scalar, the result of `E1 << E2` is `E1` left-shifted by `log2(N)` least significant bits in `E2`. The vacated bits are filled with zeros. If `E2` is negative or has a value that is greater than or equal to the width of `E1`, the C99 specification states that the behavior is undefined. Most implementations typically return `0`.

Consider the following example:

```
char x = 1;
char y = -2;
x = x << y;
```

When compiled using a C compiler such as GCC on an Intel x86 processor, `(x << y)` will return `0`. However, with OpenCL C, `(x << y)` is implemented as `(x << (y & 0x7))`, which returns `0x40`.

For vector types, `N` is the number of bits that can represent the type of elements in a vector type for `E1` used to perform the left shift. For example:

```
char2 x = (uchar2)(1, 2);
char  y = -9;

x = x << y;
```

Because components of vector `x` are an `unsigned char`, the vector shift operation is performed as `( (1 << (y & 0x7)), (2 << (y & 0x7)) )`.

Similarly, if operands are scalar, the result of `E1 >> E2` is `E1` right-shifted by `log2(N)` least significant bits in `E2`. If `E2` is negative or has a value that is greater than or equal to the width of `E1`, the C99 specification states that the behavior is undefined. For vector types, `N` is the number of bits that can represent the type of elements in a vector type for `E1` used to perform the right shift. The vacated bits are filled with zeros if `E1` is an unsigned type or is a signed type but is not a negative value. If `E1` is a signed type and a negative value, the vacated bits are filled with ones.

Unary Operators

The arithmetic unary operators (+ and –) operate on built-in scalar and vector types.

The arithmetic post- and pre- increment (++) and decrement (––) operators operate on built-in scalar and vector data types except the built-in scalar and vector floating-point data types. These operators work component-wise on their operands and result in the same type they operated on.

The logical unary operator not (!) operates on built-in scalar and vector data types except the built-in scalar and vector floating-point data types. These operators work component-wise on their operands. The result is a scalar signed integer of type `int` if both source operands are scalar and a vector signed integer type of the same size as the vector source operand. The result is of type `charn` if the source operands are `charn` or `ucharn`; `shortn` if the source operands are `shortn` or `ushortn` ; `intn` if the source operands are `intn` or `uintn`; or `longn` if the source operands are `longn` or `ulongn`.

For scalar types, these operators return `0` if the specified relation is false and `1` if the specified relation is true. For vector types, these operators return `0` if the specified relation is false and `–1` (i.e., all bits set) if the specified relation is true.

The comma operator (,) operates on expressions by returning the type and value of the rightmost expression in a comma-separated list of expressions. All expressions are evaluated, in order, from left to right. For example:

```
// comma acts as a separator not an operator.
int a = 1, b = 2, c = 3, x;

// comma acts as an operator
x = a += 2, a + b;      // a = 3, x = 5
x = (a, b, c);          // x = 3
```

The `sizeof` operator yields the size (in bytes) of its operand. The result is an integer value. The result is `1` if the operand is of type `char` or `uchar`; `2` if the operand is of type `short`, `ushort`, or `half`; `4` if the operand is of type `int`, `uint`, or `float`; and `8` if the operand is of type `long`, `ulong`, or `double`. The result is `number of components in vector * size of each scalar component` if the operand is a vector type except for 3-component vectors, which return `4 * size of each scalar component`. If the operand is an array type, the result is the total number of bytes in the array, and if the operand is a structure or union type, the

result is the total number of bytes in such an object, including any internal or trailing padding.

The behavior of applying the `sizeof` operator to the `image2d_t`, `image3d_t`, `sampler_t`, and `event_t` types is implementation-defined. For some implementations, `sizeof(sampler_t) = 4` and on some implementation this may result in a compile-time error. For portability across OpenCL implementations, it is recommended not to use the `sizeof` operator for these types.

The unary operator (`*`) denotes indirection. If the operand points to an object, the result is an l-value designating the object. If the operand has type "pointer to `type`," the result has type `type`. If an invalid value has been assigned to the pointer, the behavior of the indirection operator is undefined.

The unary operator (`&`) returns the address of its operand.

Assignment Operator

Assignments of values to variables names are done with the assignment operator (=), such as

```
lvalue = expression
```

The assignment operator stores the value of `expression` into `lvalue`. The following cases arise:

- The two operands are scalars. In this case, the operation is applied according to C99 rules.
- One operand is a scalar and the other is a vector. The scalar operand is *explicitly converted* to the element type used by the vector operand and is then widened to a vector that has the same number of elements as the vector operand. The operation is applied component-wise, resulting in the same size vector.
- The two operands are vectors of the same type. In this case, the operation is applied component-wise, resulting in the same size vector.

The following expressions are equivalent:

```
lvalue op= expression
lvalue = lvalue op expression
```

The `lvalue` and `expression` must satisfy the requirements for both operator `op` and assignment (=).

Qualifiers

OpenCL C supports four types of qualifiers: function qualifiers, address space qualifiers, access qualifiers, and type qualifiers.

Function Qualifiers

OpenCL C adds the `kernel` (or `__kernel`) function qualifier. This qualifier is used to specify that a function in the program source is a kernel function. The following example demonstrates the use of the kernel qualifier:

```
kernel void
parallel_add(global float *a, global float *b, global float *result)
{
    ...
}

// The following example is an example of an illegal kernel
// declaration and will result in a compile-time error.
// The kernel function has a return type of int instead of void.
kernel int
parallel_add(global float *a, global float *b, global float *result)
{
    ...
}
```

The following rules apply to kernel functions:

- The return type must be `void`. If the return type is not `void`, it will result in a compilation error.
- The function can be executed on a device by enqueuing a command to execute the kernel from the host.
- The function behaves as a regular function if it is called from a kernel function. The only restriction is that a kernel function with variables declared inside the function with the `local` qualifier cannot be called from another kernel function.

The following example shows a kernel function calling another kernel function that has variables declared with the `local` qualifier. The behavior is implementation-defined so it is not portable across implementations and should therefore be avoided.

```
kernel void
my_func_a(global float *src, global float *dst)
```

```
{
    local float l_var[32];

    ...
}

kernel void
my_func_b(global float * src, global float *dst)
{
    my_func_a(src, dst); // implementation-defined behavior
}
```

A better way to implement this example that is also portable is to pass the `local` variable as an argument to the `kernel`:

```
kernel void
my_func_a(global float *src, global float *dst, local float *l_var)
{

    ...
}

kernel void
my_func_b(global float * src, global float *dst, local float *l_var)
{
    my_func_a(src, dst, l_var);
}
```

Kernel Attribute Qualifiers

The `kernel` qualifier can be used with the keyword `__attribute__` to declare the following additional information about the kernel:

- `__attribute__((work_group_size_hint(X, Y, Z)))` is a hint to the compiler and is intended to specify the work-group size that will most likely be used, that is, the value specified in the `local_work_size` argument to `clEnqueueNDRangeKernel`.
- `__attribute__((reqd_work_group_size(X, Y, Z)))` is intended to specify the work-group size that will be used, that is, the value specified in the `local_work_size` argument to `clEnqueueNDRangeKernel`. This provides an opportunity for the compiler to perform specific optimizations that depend on knowing what the work-group size is.
- `__attribute__((vec_type_hint(<type>)))` is a hint to the compiler on the computational width of the `kernel`, that is, the size

of the data type the kernel is operating on. This serves as a hint to an auto-vectorizing compiler. The default value of <type> is int, indicating that the kernel is scalar in nature and the auto-vectorizer can therefore vectorize the code across the SIMD lanes of the vector unit for multiple work-items.

Address Space Qualifiers

Work-items executing a kernel have access to four distinct memory regions. These memory regions can be specified as a type qualifier. The type qualifier can be global (or __global), local (or __local), constant (or __constant), or private (or __private).

If the type of an object is qualified by an address space name, the object is allocated in the specified address space. If the address space name is not specified, then the object is allocated in the generic address space. The generic address space name (for arguments to functions in a program, or local variables in a function) is private.

A few examples that describe how to specify address space names follow:

```
// declares a pointer p in the private address space that points to
// a float object in address space global
global float *p;

// declares an array of integers in the private address space
int   f[4];

// for my_func_a function we have the following arguments:
//
//   src - declares a pointer in the private address space that
//         points to a float object in address space constant
//
//   v   - allocate in the private address space
//
int
my_func_a(constant float *src, int4 v)
{
    float temp;  // temp is allocated in the private address space.
}
```

Arguments to a kernel function that are declared to be a pointer of a type must point to one of the following address spaces only: global, local, or constant. Not specifying an address space name for such arguments will result in a compilation error. This limitation does not apply to non-kernel functions in a program.

A few examples of legal and illegal use cases are shown here:

```
kernel void my_func(int *p) // illegal because generic address space
                            // name for p is private.

kernel void
my_func(private int *p) // illegal because memory pointed to by
                        // p is allocated in private.

void
my_func(int *p) // generic address space name for p is private.
                // legal as my_func is not a kernel function

void
my_func(private int *p) // legal as my_func is not a kernel function
```

Global Address Space

This address space name is used to refer to memory objects (buffers and images) allocated from the global memory region. This memory region allows read/write access to all work-items in all work-groups executing a kernel. This address space is identified by the `global` qualifier.

A buffer object can be declared as a pointer to a scalar, vector, or user-defined struct. Some examples are:

```
global float4 *color;    // an array of float4 elements

typedef struct {
    float3 a;
    int2   b[2];
} foo_t;
global foo_t *my_info;    // an array of foo_t elements
```

The global address qualifier should not be used for image types.

Pointers to the global address space are allowed as arguments to functions (including kernel functions) and variables declared inside functions. Variables declared inside a function *cannot* be allocated in the global address space.

A few examples of legal and illegal use cases are shown here:

```
void
my_func(global float4 *vA, global float4 *vB)
{
    global float4 *p;   // legal
    global float4 a;    // illegal
}
```

Constant Address Space

This address space name is used to describe variables allocated in global memory that are accessed inside a kernel(s) as read-only variables. This memory region allows read-only access to all work-items in all work-groups executing a kernel. This address space is identified by the `constant` qualifier.

Image types cannot be allocated in the constant address space. The following example shows `imgA` allocated in the `constant` address space, which is illegal and will result in a compilation error:

```
kernel void
my_func(constant image2d_t imgA)
{
    ...
}
```

Pointers to the constant address space are allowed as arguments to functions (including kernel functions) and variables declared inside functions.

Variables in kernel function scope (i.e., the outermost scope of a kernel function) can be allocated in the `constant` address space. Variables in program scope (i.e., global variables in a program) can be allocated only in the `constant` address space. All such variables are required to be initialized, and the values used to initialize these variables must be compile-time constants. Writing to such a variable will result in a compile-time error.

Also, storage for all string literals declared in a program will be in the `constant` address space.

A few examples of legal and illegal use cases follow:

```
// legal - program scope variables can be allocated only
// in the constant address space
constant float wtsA[] = { 0, 1, 2, . . . };  // program scope

// illegal - program scope variables can be allocated only
// in the constant address space
global float wtsB[] = { 0, 1, 2, . . . };

kernel void
my_func(constant float4 *vA, constant float4 *vB)
{
    constant float4 *p = vA;  // legal
    constant float a;         // illegal – not initialized
    constant float b = 2.0f;  // legal – initialized with a compile-
                              //         time constant
```

```
    p[0] = (float4)(1.0f);    // illegal - p cannot be modified

    // the string "opencl version" is allocated in the
    // constant address space
    char *c = "opencl version";

}
```

Note The number of variables declared in the constant address space that can be used by a kernel is limited to `CL_DEVICE_MAX_CONSTANT_ARGS`. OpenCL 1.1 describes that the minimum value all implementations must support is eight. So up to eight variables declared in the constant address space can be used by a kernel and are guaranteed to work portably across all implementations. The size of these eight constant arguments is given by `CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE` and is set to 64KB. It is therefore possible that multiple constant declarations (especially those defined in the program scope) can be merged into one constant buffer as long as their total size is less than `CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE`. This aggregation of multiple variables declared to be in the constant address space is not a required behavior and so may not be implemented by all OpenCL implementations. For portable code, the developer should assume that these variables do not get aggregated into a single constant buffer.

Local Address Space

This address space name is used to describe variables that need to be allocated in local memory and are shared by all work-items of a work-group but not across work-groups executing a kernel. This memory region allows read/write access to all work-items in a work-group. This address space is identified by the `local` qualifier.

A good analogy for local memory is a user-managed cache. Local memory can significantly improve performance if a work-item or multiple work-items in a work-group are reading from the same location in global memory. For example, when applying a Gaussian filter to an image, multiple work-items read overlapping regions of the image. The overlap region size is determined by the width of the filter. Instead of reading multiple times from global memory (which is an order of magnitude slower), it is preferable to read the required data from global memory once into local memory and then have the work-items read multiple times from local memory.

Pointers to the local address space are allowed as arguments to functions (including kernel functions) and variables declared inside functions.

Variables declared inside a kernel function *can* be allocated in the local address space but with a few restrictions:

- These variable declarations must occur at kernel function scope.
- These variables cannot be initialized.

Note that variables in the local address space that are passed as pointer arguments to or declared inside a kernel function exist only for the lifetime of the work-group executing the kernel.

A few examples of legal and illegal use cases are shown here:

```
kernel void
my_func(global float4 *vA, local float4 *l)
{
    local  float4 *p;   // legal
    local  float4 a;    // legal
    a = 1;
    local  float4 b = (float4)(0); // illegal - b cannot be
                                   //           initialized

    if (...)
    {
        local float c;  // illegal - must be allocated at
                        // kernel function scope
        ...
    }
}
```

Private Address Space

This address space name is used to describe variables that are private to a work-item and cannot be shared between work-items in a work-group or across work-groups. This address space is identified by the `private` qualifier.

Variables inside a kernel function not declared with an address space qualifier, all variables declared inside non-kernel functions, and all function arguments are in the `private` address space.

Casting between Address Spaces

A pointer in an address space can be assigned to another pointer only in the same address space. Casting a pointer in one address space to a pointer in a different address space is illegal. For example:

```
kernel void
my_func(global float4 *particles)
{
    // legal - particle_ptr & particles are in the
    //         same address space
    global float *particle_ptr = (global float *)particles;

    // illegal - private_ptr and particle_ptr are in different
    //           address spaces
    float *private_ptr = (float *)particle_ptr;
}
```

Access Qualifiers

The access qualifiers can be specified with arguments that are an image type. These qualifiers specify whether the image is a read-only (`read_only` or `__read_only`) or write-only (`write_only` or `__write_only`) image. This is because of a limitation of current GPUs that do not allow reading and writing to the same image in a kernel. The reason for this is that image reads are cached in a texture cache, but writes to an image do not update the texture cache.

In the following example `imageA` is a read-only 2D image object and `imageB` is a write-only 2D image object:

```
kernel void
my_func(read_only image2d_t imageA, write_only image2d_t imageB)
{
    ...
}
```

Images declared with the `read_only` qualifier can be used with the built-in functions that read from an image. However, these images cannot be used with built-in functions that write to an image. Similarly, images declared with the `write_only` qualifier can be used only to write to an image and cannot be used to read from an image. The following examples demonstrate this:

```
kernel void
my_func(read_only image2d_t imageA,
        write_only image2d_t imageB,
        sampler_t sampler)
{
    float4 clr;
    float2 coords;

    clr = read_imagef(imageA, sampler, coords); // legal
    clr = read_imagef(imageB, sampler, coords); // illegal
```

```
    write_imagef(imageA, coords, &clr);          // illegal
    write_imagef(imageB, coords, &clr);          // legal
}
```

`imageA` is declared to be a `read_only` image so it cannot be passed as an argument to `write_imagef`. Similarly, `imageB` is declared to be a `write_only` image so it cannot be passed as an argument to `read_imagef`.

The read-write qualifier (`read_write` or `__read_write`) is reserved. Using this qualifier will result in a compile-time error.

Type Qualifiers

The type qualifiers `const`, `restrict`, and `volatile` as defined by the C99 specification are supported. These qualifiers cannot be used with the `image2d_t` and `image3d_t` type. Types other than pointer types cannot use the `restrict` qualifier.

Keywords

The following names are reserved for use as keywords in OpenCL C and cannot be used otherwise:

- Names already reserved as keywords by C99
- OpenCL C data types (defined in Tables 4.1, 4.2, and 4.6)
- Address space qualifiers: `__global`, `global`, `__local`, `local`, `__constant`, `constant`, `__private`, and `private`
- Function qualifiers: `__kernel` and `kernel`
- Access qualifiers: `__read_only`, `read_only`, `__write_only`, `write_only`, `__read_write`, and `read_write`

Preprocessor Directives and Macros

The preprocessing directives defined by the C99 specification are supported. These include

```
# non-directive
#if
#ifdef
```

```
#ifndef
#elif
#else
#endif
#include
#define
#undef
#line
#error
#pragma
```

The `defined` operator is also included.

The following example demonstrates the use of `#if`, `#elif`, `#else`, and `#endif` preprocessor macros. In this example, we use the preprocessor macros to determine which arithmetic operation to apply in the kernel. The kernel source is described here:

```
#define OP_ADD          1
#define OP_SUBTRACT     2
#define OP_MULTIPLY     3
#define OP_DIVIDE       4

kernel void
foo(global float *dst, global float *srcA, global float *srcB)
{
    size_t id = get_global_id(0);
#if OP_TYPE == OP_ADD
    dst[id] = srcA[id] + srcB[id];
#elif OP_TYPE == OP_SUBTRACT
    dst[id] = srcA[id] - srcB[id];
#elif OP_TYPE == OP_MULTIPLY
    dst[id] = srcA[id] * srcB[id];
#elif OP_TYPE == OP_DIVIDE
    dst[id] = srcA[id] / srcB[id];
#else
    dst[id] = NAN;
#endif
}
```

To build the program executable with the appropriate value for `OP_TYPE`, the application calls `clBuildProgram` as follows:

```
// build program so that kernel foo does an add operation
err = clBuildProgram(program, 0, NULL,
                                 "-DOP_TYPE=1", NULL, NULL);
```

Pragma Directives

The #pragma directive is described as

```
#pragma pp-tokensopt new-line
```

A #pragma directive where the preprocessing token OPENCL (used instead of STDC) does not immediately follow pragma in the directive (prior to any macro replacement) causes the implementation to behave in an implementation-defined manner. The behavior might cause translation to fail or cause the translator or the resulting program to behave in a nonconforming manner. Any such pragma that is not recognized by the implementation is ignored. If the preprocessing token OPENCL does immediately follow pragma in the directive (prior to any macro replacement), then no macro replacement is performed on the directive.

The following standard pragma directives are available.

Floating-Point Pragma

The FP_CONTRACT floating-point pragma can be used to allow (if the state is on) or disallow (if the state is off) the implementation to contract expressions. The FP_CONTRACT pragma definition is

```
#pragma OPENCL FP_CONTRACT on-off-switch
            on-off-switch: one of ON OFF DEFAULT
```

A detailed description of #pragma OPENCL FP_CONTRACT is found in Chapter 5 in the section "Floating-Point Pragmas."

Compiler Directives for Optional Extensions

The #pragma OPENCL EXTENSION directive controls the behavior of the OpenCL compiler with respect to language extensions. The #pragma OPENCL EXTENSION directive is defined as follows, where extension_name is the name of the extension:

```
#pragma OPENCL EXTENSION extension_name: behavior
#pragma OPENCL EXTENSION all : behavior
            behavior: enable or disable
```

The extension_name will have names of the form cl_khr_<*name*> for an extension (such as cl_khr_fp64) approved by the OpenCL working group and will have names of the form cl_<*vendor_name*>_<*name*> for vendor extensions. The token all means that the behavior applies to all extensions supported by the compiler. The behavior can be set to one of the values given in Table 4.9.

Table 4.9 Optional Extension Behavior Description

Behavior	Description
`enable`	Enable the extension `extension_name`. Report an error on the `#pragma OpenCL EXTENSION` if the `extension_name` is not supported, or if `all` is specified.
`disable`	Behave (including issuing errors and warnings) as if the extension `extension_name` is not part of the language definition. If `all` is specified, then behavior must revert back to that of the nonextended core version of the language being compiled to. Warn on the `#pragma OPENCL EXTENSION` if the extension `extension_name` is not supported.

The `#pragma OPENCL EXTENSION` directive is a simple, low-level mechanism to set the behavior for each language extension. It does not define policies such as which combinations are appropriate; these are defined elsewhere. The order of directives matters in setting the behavior for each extension. Directives that occur later override those seen earlier. The `all` variant sets the behavior for all extensions, overriding all previously issued extension directives, but only if the `behavior` is set to `disable`.

An extension needs to be enabled before any language feature (such as preprocessor macros, data types, or built-in functions) of this extension is used in the OpenCL program source. The following example shows how to enable the double-precision floating-point extension:

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
double x = 2.0;
```

If this extension is not supported, then a compilation error will be reported for `double x = 2.0`. If this extension is supported, this enables the use of double-precision floating-point extensions in the program source following this directive.

Similarly, the `cl_khr_3d_image_writes` extension adds new built-in functions that support writing to a 3D image:

```
#pragma OPENCL EXTENSION cl_khr_3d_image_writes : enable
kernel void my_func(write_only image3d_t img, ...)

{
    float4 coord, clr;
    ...
    write_imagef(img, coord, clr);
}
```

The built-in functions such as `write_imagef` with `image3d_t` in the preceding example can be called only if this extension is enabled; otherwise a compilation error will occur.

The initial state of the compiler is as if the following directive were issued, telling the compiler that all error and warning reporting must be done according to this specification, ignoring any extensions:

```
#pragma OPENCL EXTENSION all : disable
```

Every extension that affects the OpenCL language semantics or syntax or adds built-in functions to the language must also create a preprocessor `#define` that matches the extension name string. This `#define` would be available in the language if and only if the extension is supported on a given implementation. For example, an extension that adds the extension string `cl_khr_fp64` should also add a preprocessor `#define` called `cl_khr_fp64`. A kernel can now use this preprocessor `#define` to do something like this:

```
#ifdef cl_khr_fp64
    // do something using this extension
#else
    // do something else or #error
#endif
```

Macros

The following predefined macro names are available:

- `__FILE__` is the presumed name of the current source file (a character string literal).
- `__LINE__` is the presumed line number (within the current source file) of the current source line (an integer constant).
- `CL_VERSION_1_0` substitutes the integer `100`, reflecting the OpenCL 1.0 version.
- `CL_VERSION_1_1` substitutes the integer `110`, reflecting the OpenCL 1.1 version.
- `__OPENCL_VERSION__` substitutes an integer reflecting the version number of the OpenCL supported by the OpenCL device. This reflects both the language version supported and the device capabilities as given in Table 4.3 of the OpenCL 1.1 specification. The version of OpenCL described in this book will have `__OPENCL_VERSION__` substitute the integer `110`.

- `__ENDIAN_LITTLE__` is used to determine if the OpenCL device is a little endian architecture or a big endian architecture (an integer constant of `1` if the device is little endian and is undefined otherwise).
- `__kernel_exec(X, typen)` (and `kernel_exec(X, typen)`) is defined as
  ```
  __kernel __attribute__((work_group_size_hint(X, 1, 1))) \
      __attribute__((vec_type_hint(typen))).
  ```
- `__IMAGE_SUPPORT__` is used to determine if the OpenCL device supports images. This is an integer constant of `1` if images are supported and is undefined otherwise.
- `__FAST_RELAXED_MATH__` is used to determine if the `-cl-fast-relaxed-math` optimization option is specified in build options given to `clBuildProgram`. This is an integer constant of `1` if the `-cl-fast-relaxed-math` build option is specified and is undefined otherwise.

The macro names defined by the C99 specification but not currently supported by OpenCL are reserved for future use.

Restrictions

OpenCL C implements the following restrictions. Some of these restrictions have already been described in this chapter but are also included here to provide a single place where the language restrictions are described.

- Kernel functions have the following restrictions:
  - Arguments to kernel functions that are pointers must use the `global`, `constant`, or `local` qualifier.
  - An argument to a kernel function cannot be declared as a pointer to a pointer(s).
  - Arguments to kernel functions cannot be declared with the following built-in types: `bool`, `half`, `size_t`, `ptrdiff_t`, `intptr_t`, `uintptr_t`, or `event_t`.
  - The return type for a kernel function must be `void`.
  - Arguments to kernel functions that are declared to be a struct cannot pass OpenCL objects (such as buffers, images) as elements of the struct.

- Bit field struct members are not supported.
- Variable-length arrays and structures with flexible (or unsized) arrays are not supported.
- Variadic macros and functions are not supported.
- The `extern`, `static`, `auto`, and `register` storage class specifiers are not supported.
- Predefined identifiers such as `__func__` are not supported.
- Recursion is not supported.
- The library functions defined in the C99 standard headers—`assert.h`, `ctype.h`, `complex.h`, `errno.h`, `fenv.h`, `float.h`, `inttypes.h`, `limits.h`, `locale.h`, `setjmp.h`, `signal.h`, `stdarg.h`, `stdio.h`, `stdlib.h`, `string.h`, `tgmath.h`, `time.h`, `wchar.h`, and `wctype.h`—are not available and cannot be included by a program.
- The image types `image2d_t` and `image3d_t` can be specified only as the types of a function argument. They cannot be declared as local variables inside a function or as the return types of a function. An image function argument cannot be modified. An image type cannot be used with the `private`, `local`, and `constant` address space qualifiers. An image type cannot be used with the `read_write` access qualifier, which is reserved for future use. An image type cannot be used to declare a variable, a structure or union field, an array of images, a pointer to an image, or the return type of a function.
- The sampler type `sampler_t` can be specified only as the type of a function argument or a variable declared in the program scope or the outermost scope of a kernel function. The behavior of a sampler variable declared in a non-outermost scope of a kernel function is implementation-defined. A sampler argument or a variable cannot be modified. The sampler type cannot be used to declare a structure or union field, an array of samplers, a pointer to a sampler, or the return type of a function. The sampler type cannot be used with the `local` and `global` address space qualifiers.
- The event type `event_t` can be used as the type of a function argument except for kernel functions or a variable declared inside a function. The event type can be used to declare an array of events. The event type can be used to declare a pointer to an event, for example, `event_t *event_ptr`. An event argument or variable cannot be modified. The event type cannot be used to declare a structure or

union field, or for variables declared in the program scope. The event type cannot be used with the `local`, `constant`, and `global` address space qualifiers.

- The behavior of irreducible control flow in a kernel is implementation-defined. Irreducible control flow is typically encountered in code that uses `goto`s. An example of irreducible control flow is a `goto` jumping inside a nested loop or a Duff's device.

Index

B

C

D

E

F

G

H

I

K

N

O

P

Q

R

S

T

U

V

W

Z