Programming Guide

AMD Accelerated Parallel Processing TECHNOLOGY

AMD Accelerated Parallel Processing

OpenCL™

July 2012

rev2.3

© 2012 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo, AMD Accelerated Parallel Processing, the AMD Accelerated Parallel Processing logo, ATI, the ATI logo, Radeon, FireStream, FirePro, Catalyst, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Microsoft, Visual Studio, Windows, and Windows Vista are registered trademarks of Microsoft Corporation in the U.S. and/or other jurisdictions. Other names are for informational purposes only and may be trademarks of their respective owners. OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos.

The contents of this document are provided in connection with Advanced Micro Devices, Inc. ("AMD") products. AMD makes no representations or warranties with respect to the accuracy or completeness of the contents of this publication and reserves the right to make changes to specifications and product descriptions at any time without notice. The information contained herein may be of a preliminary or advance nature and is subject to change without notice. No license, whether express, implied, arising by estoppel or otherwise, to any intellectual property rights is granted by this publication. Except as set forth in AMD's Standard Terms and Conditions of Sale, AMD assumes no liability whatsoever, and disclaims any express or implied warranty, relating to its products including, but not limited to, the implied warranty of merchantability, fitness for a particular purpose, or infringement of any intellectual property right.

AMD's products are not designed, intended, authorized or warranted for use as components in systems intended for surgical implant into the body, or in other applications intended to support or sustain life, or in any other application in which the failure of AMD's product could create a situation where personal injury, death, or severe property or environmental damage may occur. AMD reserves the right to discontinue or make changes to its products at any time without notice.

Advanced Micro Devices, Inc.
One AMD Place
P.O. Box 3453
Sunnyvale, CA 94088-3453
www.amd.com

For AMD Accelerated Parallel Processing:

URL:	**developer.amd.com/appsdk**
Developing:	**developer.amd.com/**
Support:	**developer.amd.com/appsdksupport**
Forum:	**developer.amd.com/openclforum**

Preface

About This Document

This document provides a basic description of the AMD Accelerated Parallel Processing environment and components. It describes the basic architecture of stream processors and provides useful performance tips. This document also provides a guide for programmers who want to use AMD Accelerated Parallel Processing to accelerate their applications.

Audience

This document is intended for programmers. It assumes prior experience in writing code for CPUs and a basic understanding of threads (work-items). While a basic understanding of GPU architectures is useful, this document does not assume prior graphics knowledge. It further assumes an understanding of chapters 1, 2, and 3 of the *OpenCL Specification* (for the latest version, see http://www.khronos.org/registry/cl/).

Organization

This AMD Accelerated Parallel Processing document begins, in Chapter 1, with an overview of: the AMD Accelerated Parallel Processing programming models, OpenCL, the AMD Compute Abstraction Layer (CAL), the AMD APP Kernel Analyzer, and the AMD APP Profiler. Chapter 2 discusses the compiling and running of OpenCL programs. Chapter 3 describes using GNU debugger (GDB) to debug OpenCL programs. Chapter 4 is a discussion of general performance and optimization considerations when programming for AMD Accelerated Parallel Processing devices. Chapter 5 details performance and optimization considerations specifically for Southern Island devices. Chapter 6 details performance and optimization devices for Evergreen and Northern Islands devices. Appendix A describes the supported optional OpenCL extensions. Appendix B details the installable client driver (ICD) for OpenCL. Appendix C details the compute kernel and contrasts it with a pixel shader. Appendix D lists the device parameters. Appendix E describes the OpenCL binary image format (BIF). Appendix F describes the OpenVideo Decode API. Appendix G describes the interoperability between OpenCL and OpenGL. The last section of this book is a glossary of acronyms and terms, as well as an index.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Conventions

The following conventions are used in this document.

`mono-spaced font`	A filename, file path, or code.
*	Any number of alphanumeric characters in the name of a code format, parameter, or instruction.
[1,2)	A range that includes the left-most value (in this case, 1) but excludes the right-most value (in this case, 2).
[1,2]	A range that includes both the left-most and right-most values (in this case, 1 and 2).
{x | y}	One of the multiple options listed. In this case, x or y.
0.0f 0.0	A single-precision (32-bit) floating-point value. A double-precision (64-bit) floating-point value.
1011b	A binary value, in this example a 4-bit value.
7:4	A bit range, from bit 7 to 4, inclusive. The high-order bit is shown first.
italicized word or phrase	The first use of a term or concept basic to the understanding of stream computing.

Related Documents

- *The OpenCL Specification*, Version 1.1, Published by Khronos OpenCL Working Group, Aaftab Munshi (ed.), 2010.
- AMD, *R600 Technology, R600 Instruction Set Architecture*, Sunnyvale, CA, est. pub. date 2007. This document includes the RV670 GPU instruction details.
- ISO/IEC 9899:TC2 - *International Standard - Programming Languages - C*
- Kernighan Brian W., and Ritchie, Dennis M., *The C Programming Language*, Prentice-Hall, Inc., Upper Saddle River, NJ, 1978.
- I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, *“Brook for GPUs: stream computing on graphics hardware,”* ACM Trans. Graph., vol. 23, no. 3, pp. 777–786, 2004.
- *AMD Compute Abstraction Layer (CAL) Intermediate Language (IL) Reference Manual*. Published by AMD.
- Buck, Ian; Foley, Tim; Horn, Daniel; Sugerman, Jeremy; Hanrahan, Pat; Houston, Mike; Fatahalian, Kayvon. “BrookGPU” http://graphics.stanford.edu/projects/brookgpu/
- Buck, Ian. “Brook Spec v0.2”. October 31, 2003. http://merrimac.stanford.edu/brook/brookspec-05-20-03.pdf
- *OpenGL Programming Guide*, at http://www.glprogramming.com/red/
- *Microsoft DirectX Reference Website*, at http://msdn.microsoft.com/en-us/directx
- *GPGPU*: http://www.gpgpu.org, and Stanford BrookGPU discussion forum http://www.gpgpu.org/forums/

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Contact Information

URL: **developer.amd.com/appsdk**
Developing: **developer.amd.com/**
Support: **developer.amd.com/appsdksupport**
Forum: **developer.amd.com/openclforum**

REVISION HISTORY

Rev	Description
1.3 e	Deleted encryption reference.
1.3f	Added basic guidelines to CL-GL Interop appendix. Corrected code in two samples in Chpt. 4.
1.3g	Numerous changes to CL-GL Interop appendix. Added subsections to Additional Performance Guidance on CPU Programmers Using OpenCL to Program CPUs and Using Special CPU Instructions in the Optimizing Kernel Code subsection.
2.0	Added ELF Header section in Appendix E.
2.1	New Profiler and KernelAnalyzer sections in chapter 4. New AMD gDEBugger section in chapter 3. Added extensions to Appendix A. Numerous updates throughout for Southern Islands, especially in Chapters 1 and 5. Split original chapter 4 into three chapters. Now, chapter 4 is general considerations for Evergreen, Northern Islands, and Southern Islands; chapter 5 is specifically for Southern Islands devices; chapter 6 is for Evergreen and Northern Islands devices. Update of Device Parameters in Appendix D.
2.1a	Reinstated some supplemental compiler options in Section 2.1.4. Changes/additions to Table 4.3
2.1b	Minor change in Section 1.8.3, indicating that LDS model has not changed from previous GPU families.
2.2	Addition of channel mapping information (chpt 5). Minor corrections throughout. Deletion of duplicate material from chpt 6.
2.3	Inclusion of upgraded index. Minor rewording and corrections.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Contents

Preface

Contents

Chapter 1 OpenCL Architecture and AMD Accelerated Parallel Processing

1.1 Software Overview 1-1
1.1.1 Data-Parallel Programming Model 1-1
1.1.2 Task-Parallel Programming Model 1-1
1.1.3 Synchronization 1-1
1.2 Hardware Overview for Southern Islands Devices 1-2
1.3 Hardware Overview for Evergreen and Northern Islands Devices 1-4
1.4 The AMD Accelerated Parallel Processing Implementation of OpenCL 1-6
1.4.1 Work-Item Processing 1-9
1.4.2 Flow Control 1-10
1.4.3 Work-Item Creation 1-11
1.5 Memory Architecture and Access 1-11
1.5.1 Memory Access 1-13
1.5.2 Global Buffer 1-13
1.5.3 Image Read/Write 1-13
1.5.4 Memory Load/Store 1-14
1.6 Communication Between Host and GPU in a Compute Device 1-14
1.6.1 PCI Express Bus 1-14
1.6.2 Processing API Calls: The Command Processor 1-14
1.6.3 DMA Transfers 1-15
1.6.4 Masking Visible Devices 1-15
1.7 GPU Compute Device Scheduling 1-15
1.8 Terminology 1-17
1.8.1 Compute Kernel 1-17
1.8.2 Wavefronts and Work-groups 1-18
1.8.3 Local Data Store (LDS) 1-18
1.9 Programming Model 1-18
1.10 Example Programs 1-20
1.10.1 First Example: Simple Buffer Write 1-20
1.10.2 Example: Parallel Min() Function 1-23

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Chapter 2 Building and Running OpenCL Programs
2.1 Compiling the Program 2-2
2.1.1 Compiling on Windows 2-2
2.1.2 Compiling on Linux 2-3
2.1.3 Supported Standard OpenCL Compiler Options 2-4
2.1.4 AMD-Developed Supplemental Compiler Options 2-4
2.2 Running the Program 2-5
2.2.1 Running Code on Windows 2-6
2.2.2 Running Code on Linux 2-7
2.3 Calling Conventions 2-7

Chapter 3 Debugging OpenCL
3.1 AMD gDEBugger 3-1
3.2 Debugging CPU Kernels with GDB 3-2
3.2.1 Setting the Environment 3-2
3.2.2 Setting the Breakpoint in an OpenCL Kernel 3-2
3.2.3 Sample GDB Session 3-3
3.2.4 Notes 3-4

Chapter 4 OpenCL Performance and Optimization
4.1 AMD APP Profiler 4-1
4.1.1 Collecting OpenCL Application Trace 4-2
4.1.2 Collecting OpenCL GPU Kernel Performance Counters 4-5
4.1.3 OpenCL Kernel Occupancy Modeler 4-6
4.2 AMD APP KernelAnalyzer 4-8
4.3 Analyzing Processor Kernels 4-9
4.3.1 Intermediate Language and GPU Disassembly 4-9
4.3.2 Generating IL and ISA Code 4-10
4.4 Estimating Performance 4-10
4.4.1 Measuring Execution Time 4-10
4.4.2 Using the OpenCL timer with Other System Timers 4-11
4.4.3 Estimating Memory Bandwidth 4-12
4.5 OpenCL Memory Objects 4-13
4.5.1 Types of Memory Used by the Runtime 4-13
4.5.2 Placement 4-16
4.5.3 Memory Allocation 4-18
4.5.4 Mapping 4-18
4.5.5 Reading, Writing, and Copying 4-21
4.5.6 Command Queue 4-21
4.6 OpenCL Data Transfer Optimization 4-21
4.6.1 Definitions 4-22
4.6.2 Buffers 4-22

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

4.7 Using Multiple OpenCL Devices 4-29
4.7.1 CPU and GPU Devices 4-29
4.7.2 When to Use Multiple Devices 4-31
4.7.3 Partitioning Work for Multiple Devices 4-32
4.7.4 Synchronization Caveats 4-34
4.7.5 GPU and CPU Kernels 4-35
4.7.6 Contexts and Devices 4-36

Chapter 5 OpenCL Performance and Optimization for Southern Islands Devices

5.1 Global Memory Optimization 5-1
5.1.1 Channel Conflicts 5-2
5.1.2 Coalesced Writes 5-8
5.1.3 Hardware Variations 5-9
5.2 Local Memory (LDS) Optimization 5-9
5.3 Constant Memory Optimization 5-12
5.4 OpenCL Memory Resources: Capacity and Performance 5-14
5.5 Using LDS or L1 Cache 5-15
5.6 NDRange and Execution Range Optimization 5-16
5.6.1 Hiding ALU and Memory Latency 5-16
5.6.2 Resource Limits on Active Wavefronts 5-17
5.6.3 Partitioning the Work 5-20
5.6.4 Summary of NDRange Optimizations 5-22
5.7 Instruction Selection Optimizations 5-23
5.7.1 Instruction Bandwidths 5-23
5.7.2 AMD Media Instructions 5-24
5.7.3 Math Libraries 5-24
5.7.4 Compiler Optimizations 5-25
5.8 Additional Performance Guidance 5-25
5.8.1 Loop Unroll `pragma` 5-25
5.8.2 Memory Tiling 5-26
5.8.3 General Tips 5-27
5.8.4 Guidance for CUDA Programmers Using OpenCL 5-29
5.8.5 Guidance for CPU Programmers Using OpenCL to Program GPUs 5-29
5.8.6 Optimizing Kernel Code 5-30
5.8.7 Optimizing Kernels for Southern Island GPUs 5-31
5.9 Specific Guidelines for Southern Islands GPUs 5-32

Chapter 6 OpenCL Performance and Optimization for Evergreen and Northern Islands Devices

6.1 Global Memory Optimization 6-1
6.1.1 Two Memory Paths 6-3
6.1.2 Channel Conflicts 6-6
6.1.3 Float4 Or Float1 6-11

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

6.1.4 Coalesced Writes 6-12
6.1.5 Alignment..6-14
6.1.6 Summary of Copy Performance...6-16
6.1.7 Hardware Variations..6-16
6.2 Local Memory (LDS) Optimization 6-16
6.3 Constant Memory Optimization.. 6-19
6.4 OpenCL Memory Resources: Capacity and Performance 6-20
6.5 Using LDS or L1 Cache 6-22
6.6 NDRange and Execution Range Optimization.. 6-23
6.6.1 Hiding ALU and Memory Latency 6-23
6.6.2 Resource Limits on Active Wavefronts..6-24
6.6.3 Partitioning the Work..6-28
6.6.4 Optimizing for Cedar 6-32
6.6.5 Summary of NDRange Optimizations...6-32
6.7 Using Multiple OpenCL Devices 6-33
6.7.1 CPU and GPU Devices 6-33
6.7.2 When to Use Multiple Devices 6-35
6.7.3 Partitioning Work for Multiple Devices 6-35
6.7.4 Synchronization Caveats 6-37
6.7.5 GPU and CPU Kernels...6-39
6.7.6 Contexts and Devices...6-40
6.8 Instruction Selection Optimizations 6-41
6.8.1 Instruction Bandwidths 6-41
6.8.2 AMD Media Instructions..6-42
6.8.3 Math Libraries...6-42
6.8.4 VLIW and SSE Packing 6-43
6.8.5 Compiler Optimizations...6-45
6.9 Clause Boundaries 6-46
6.10 Additional Performance Guidance.. 6-48
6.10.1 Loop Unroll `pragma`..6-48
6.10.2 Memory Tiling...6-48
6.10.3 General Tips...6-49
6.10.4 Guidance for CUDA Programmers Using OpenCL 6-51
6.10.5 Guidance for CPU Programmers Using OpenCL to Program GPUs.........................6-52
6.10.6 Optimizing Kernel Code 6-52
6.10.7 Optimizing Kernels for Evergreen and 69XX-Series GPUs.......................................6-53

Chapter 7 OpenCL Static C++ Programming Language

7.1 Overview 7-1
7.1.1 Supported Features 7-1
7.1.2 Unsupported Features...7-2
7.1.3 Relations with ISO/IEC C++ 7-2

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

7.2 Additions and Changes to Section 5 - The OpenCL C Runtime 7-2
  7.2.1 Additions and Changes to Section 5.7.1 - Creating Kernel Objects 7-2
  7.2.2 Passing Classes between Host and Device 7-3
7.3 Additions and Changes to Section 6 - The OpenCL C Programming Language 7-3
  7.3.1 Building C++ Kernels 7-3
  7.3.2 Classes and Derived Classes 7-3
  7.3.3 Namespaces 7-4
  7.3.4 Overloading 7-4
  7.3.5 Templates 7-5
  7.3.6 Exceptions 7-6
  7.3.7 Libraries 7-6
  7.3.8 Dynamic Operation 7-6
7.4 Examples 7-6
  7.4.1 Passing a Class from the Host to the Device and Back 7-6
  7.4.2 Kernel Overloading 7-7
  7.4.3 Kernel Template 7-8

Appendix A OpenCL Optional Extensions

A.1 Extension Name Convention A-1
A.2 Querying Extensions for a Platform A-1
A.3 Querying Extensions for a Device A-2
A.4 Using Extensions in Kernel Programs A-2
A.5 Getting Extension Function Pointers A-3
A.6 List of Supported Extensions that are Khronos-Approved A-3
A.7 cl_ext Extensions A-4
A.8 AMD Vendor-Specific Extensions A-4
  A.8.1 cl_amd_fp64 .... A-4
  A.8.2 cl_amd_vec3 .... A-4
  A.8.3 cl_amd_device_persistent_memory A-4
  A.8.4 cl_amd_device_attribute_query A-5
  A.8.5 cl_amd_device_profiling_timer_offset A-5
  A.8.6 cl_amd_device_topology A-5
  A.8.7 cl_amd_device_board_name A-5
  A.8.8 cl_amd_compile_options A-6
  A.8.9 cl_amd_offline_devices A-6
  A.8.10 cl_amd_event_callback A-6
  A.8.11 cl_amd_popcnt A-7
  A.8.12 cl_amd_media_ops A-7
  A.8.13 cl_amd_media_ops2 .... A-9
  A.8.14 cl_amd_printf A-12
A.9 cl_amd_predefined_macros A-13
A.10 Supported Functions for cl_amd_fp64 / cl_khr_fp64 .... A-15

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

A.11 Extension Support by Device....A-15

Appendix B The OpenCL Installable Client Driver (ICD)

B.1 Overview....B-1

B.2 Using ICD....B-1

Appendix C Compute Kernel

C.1 Differences from a Pixel Shader....C-1

C.2 Indexing....C-1

C.3 Performance Comparison....C-2

C.4 Pixel Shader....C-2

C.5 Compute Kernel....C-3

C.6 LDS Matrix Transpose....C-4

C.7 Results Comparison....C-4

Appendix D Device Parameters

Appendix E OpenCL Binary Image Format (BIF) v2.0

E.1 Overview....E-1

E.1.1 Executable and Linkable Format (ELF) Header....E-2

E.1.2 Bitness....E-3

E.2 BIF Options....E-3

Appendix F Open Decode API Tutorial

F.1 Overview....F-1

F.2 Initializing....F-2

F.3 Creating the Context....F-2

F.4 Creating the Session....F-3

F.5 Decoding....F-3

F.6 Destroying Session and Context....F-4

Appendix G OpenCL-OpenGL Interoperability

G.1 Under Windows....G-1

G.1.1 Single GPU Environment....G-2

G.1.2 Multi-GPU Environment....G-4

G.1.3 Limitations....G-7

G.2 Linux Operating System....G-8

G.2.1 Single GPU Environment....G-8

G.2.2 Multi-GPU Configuration....G-11

Index

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Figures

1.1 Generalized AMD GPU Compute Device Structure for Southern Islands Devices 1-2
1.2 AMD Radeon™ HD 79XX Device Partial Block Diagram 1-3
1.3 Generalized AMD GPU Compute Device Structure...1-4
1.4 Simplified Block Diagram of and Evergreen-Family GPU 1-5
1.5 AMD Accelerated Parallel Processing Software Ecosystem 1-6
1.6 Simplified Mapping of OpenCL onto AMD Accelerated Parallel Processing for Evergreen and Northern Island Devices 1-8
1.7 Work-Item Grouping Into Work-Groups and Wavefronts 1-9
1.8 Interrelationship of Memory Domains for Southern Islands Devices 1-12
1.9 Dataflow between Host and GPU 1-12
1.10 Simplified Execution Of Work-Items On A Single Stream Core 1-16
1.11 Stream Core Stall Due to Data Dependency 1-17
1.12 OpenCL Programming Model...1-19
2.1 OpenCL Compiler Toolchain...2-1
2.2 Runtime Processing Structure 2-6
4.1 Timeline and API Trace View in Microsoft Visual Studio 2010 4-3
4.2 Context Summary Page View in Microsoft Visual Studio 2010..4-4
4.3 Warning(s) and Error(s) Page 4-5
4.4 Sample Session View in Microsoft Visual Studio 2010 4-6
4.5 Sample Kernel Occupancy Modeler Screen 4-7
4.6 AMD APP Kernel Analyzer..4-9
5.1 Memory System 5-1
5.2 Channel Remapping/Interleaving...5-4
5.3 Transformation to Staggered Offsets..5-7
5.4 One Example of a Tiled Layout Format...5-27
5.5 Northern Islands Compute Unit Arrangement..5-35
5.6 Southern Island Compute Unit Arrangement 5-35
6.1 Memory System 6-2
6.2 FastPath (blue) vs CompletePath (red) Using float1 6-3
6.3 Transformation to Staggered Offsets..6-9
6.4 Two Kernels: One Using float4 (blue), the Other float1 (red) 6-11
6.5 Effect of Varying Degrees of Coalescing - Coal (blue), NoCoal (red), Split (green) 6-13
6.6 Unaligned Access Using float1..6-15
6.7 Unmodified Loop 6-43
6.8 Kernel Unrolled 4X...6-44
6.9 Unrolled Loop with Stores Clustered...6-44
6.10 Unrolled Kernel Using float4 for Vectorization 6-45
6.11 One Example of a Tiled Layout Format...6-49
C.1 Pixel Shader Matrix Transpose C-2
C.2 Compute Kernel Matrix Transpose.. C-3
C.3 LDS Matrix Transpose C-4
F.1 Open Decode with Optional Post-Processing.. F-1

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Tables

4.1 Memory Bandwidth in GB/s (R = read, W = write) in GB/s 4-14
4.2 OpenCL Memory Object Properties 4-17
4.3 Transfer policy on `clEnqueueMapBuffer` / `clEnqueueMapImage` / `clEnqueueUnmapMemObject` for Copy Memory Objects 4-20
4.4 CPU and GPU Performance Characteristics 4-30
5.1 Hardware Performance Parameters 5-14
5.2 Effect of LDS Usage on Wavefronts/CU1 5-19
5.3 Instruction Throughput (Operations/Cycle for Each Stream Processor) 5-23
5.4 Resource Limits for Northern Islands and Southern Islands 5-34
6.1 Bandwidths for 1D Copies 6-4
6.2 Bandwidths for Different Launch Dimensions 6-8
6.3 Bandwidths Including float1 and float4 6-12
6.4 Bandwidths Including Coalesced Writes 6-14
6.5 Bandwidths Including Unaligned Access 6-15
6.6 Hardware Performance Parameters 6-21
6.7 Impact of Register Type on Wavefronts/CU 6-26
6.8 Effect of LDS Usage on Wavefronts/CU1 6-28
6.9 CPU and GPU Performance Characteristics 6-33
6.10 Instruction Throughput (Operations/Cycle for Each Stream Processor) 6-41
6.11 Native Speedup Factor 6-43
A.1 Extension Support for AMD GPU Devices 1 .... A-15
A.2 Extension Support for Older AMD GPUs and CPUs A-16
D.1 Parameters for 7xxx Devices D-2
D.2 Parameters for 68xx and 69xx Devices D-3
D.3 Parameters for 65xx, 66xx, and 67xx Devices D-4
D.4 Parameters for 64xx Devices D-5
D.5 Parameters for Zacate and Ontario Devices D-6
D.6 Parameters for 56xx, 57xx, 58xx, Eyfinity6, and 59xx Devices D-7
D.7 Parameters for Exxx, Cxx, 54xx, and 55xx Devices D-8
E.1 ELF Header Fields E-2

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Chapter 1 OpenCL Architecture and AMD Accelerated Parallel Processing

This chapter provides a general software and hardware overview of the AMD Accelerated Parallel Processing implementation of the OpenCL standard. It explains the memory structure and gives simple programming examples.

1.1 Software Overview

OpenCL supports data-parallel and task-parallel programming models, as well as hybrids of these models. Of the two, the primary one is the data-parallel model.

1.1.1 Data-Parallel Programming Model

In the data parallel programming model, a computation is defined in terms of a sequence of instructions that executes at each point in an N-dimensional index space. It is a common, though by not required, formulation of an algorithm that each computation index maps to an element in an input data set.

The OpenCL data-parallel programming model is hierarchical. The hierarchical subdivision can be specified in two ways:

- Explicitly - the developer defines the total number of work-items to execute in parallel, as well as the division of work-items into specific work-groups.
- Implicitly - the developer specifies the total number of work-items to execute in parallel, and OpenCL manages the division into work-groups.

1.1.2 Task-Parallel Programming Model

In this model, a kernel instance is executed independently of any index space. This is equivalent to executing a kernel on a compute device with a work-group and NDRange containing a single work-item. Parallelism is expressed using vector data types implemented by the device, enqueuing multiple tasks, and/or enqueuing native kernels developed using a programming model orthogonal to OpenCL.

1.1.3 Synchronization

The two domains of synchronization in OpenCL are work-items in a single work-group and command-queue(s) in a single context. Work-group barriers enable synchronization of work-items in a work-group. Each work-item in a work-group must first execute the barrier before executing any instruction beyond this barrier. Either all of, or none of, the work-items in a work-group must encounter the

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

barrier. A barrier or `mem_fence` operation does not have global scope, but is relevant only to the local workgroup on which they operate. However, atomic operations done on global memory do have a global scope, hence may provide a way to do global synchronization.

There are two types of synchronization between commands in a command-queue:

- command-queue barrier - enforces ordering within a single queue. Any resulting changes to memory are available to the following commands in the queue.
- events - enforces ordering between, or within, queues. Enqueued commands in OpenCL return an event identifying the command as well as the memory object updated by it. This ensures that following commands waiting on that event see the updated memory objects before they execute.

1.2 Hardware Overview for Southern Islands Devices

A general OpenCL device comprises compute units (CUs), each of which has sub-modules that ultimately have ALUs. A work-item (or SPMD kernel instance) executes on an ALU, as shown in Figure 1.1).

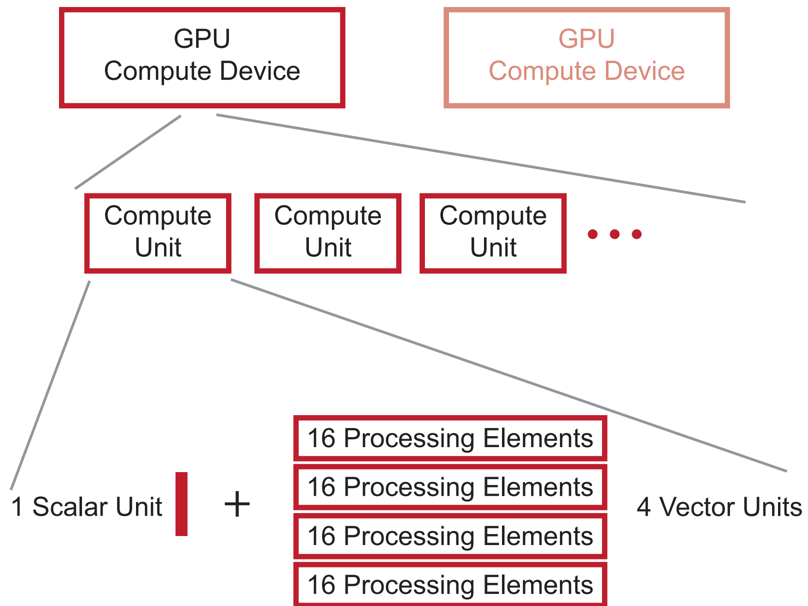

Figure 1.1 Generalized AMD GPU Compute Device Structure for Southern Islands Devices

For AMD Radeon™ HD 79XX devices, each of the 32 CUs has one Scalar Unit and four Vector Units, each of which contain an array of 16 PEs. Each PE consists of one ALU. Figure 1.2 shows only two compute units of the array that comprises the compute device of the AMD Radeon™ HD 7XXX family. The four Vector Units use SIMD execution of a scalar instruction. This makes it possible to run four separate instructions at once, but they are dynamically scheduled (as

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

opposed to those for the AMD Radeon™ HD 69XX devices, which are statically scheduled.

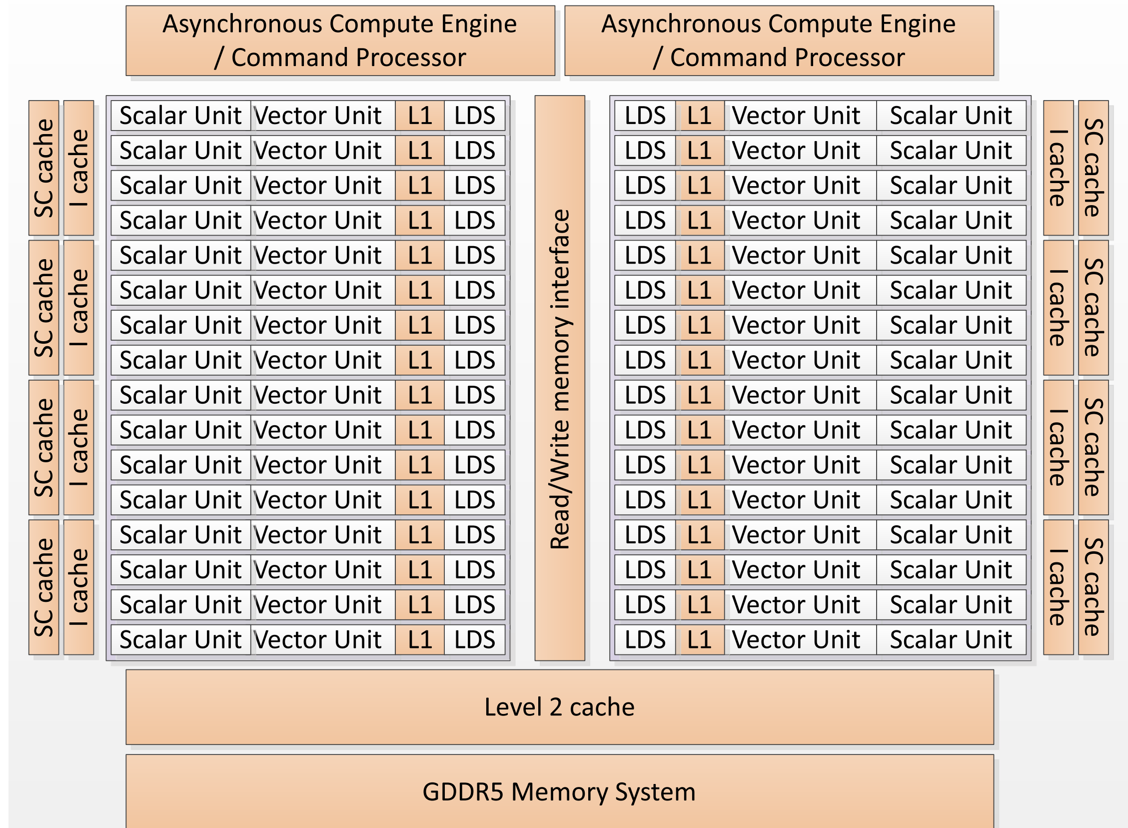

Figure 1.2 AMD Radeon™ HD 79XX Device Partial Block Diagram

In Figure 1.2, there are two command processors, which can process two command queues concurrently. The Scalar Unit, Vector Unit, Level 1 data cache (L1), and Local Data Share (LDS) are the components of one compute unit, of which there are 32. The SC cache is the scalar unit data cache, and the Level 2 cache consists of instructions and data.

As noted, the AMD Radeon™ HD 79XX devices also have a scalar unit, and the instruction stream contains both scalar and vector instructions. On each cycle, it selects a scalar instruction and a vector instruction (as well as a memory operation and a branch operation, if available); it issues one to the scalar unit, the other to the vector unit; this takes four cycles to issue over the four vector cores (the same four cycles over which the 16 units execute 64 work-items).

The number of compute units in an AMD GPU, and the way they are structured, varies with the device family, as well as device designations within a family. Each of these vector units possesses ALUs (processing elements). For devices in the Northern Islands (AMD Radeon™ HD 69XX) and Southern Islands (AMD Radeon™ HD 7XXX) families, these ALUs are arranged in four (in the Evergreen family, there are five) SIMD arrays consisting of 16 processing elements each. (See Section 1.3, "Hardware Overview for Evergreen and Northern Islands

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Devices.") Each of these arrays executes a single instruction across each lane for each of a block of 16 work-items. That instruction is repeated over four cycles to make the 64-element vector called a *wavefront*. On devices in the Southern Island family, the four stream cores execute code from four different wavefronts.

1.3 Hardware Overview for Evergreen and Northern Islands Devices

A general OpenCL device comprises compute units, each of which can have multiple processing elements. A work-item (or SPMD kernel instance) executes on a single processing element. The processing elements within a compute unit can execute in lock-step using SIMD execution. Compute units, however, execute independently (see Figure 1.3).

AMD GPUs consists of multiple compute units. The number of them and the way they are structured varies with the device family, as well as device designations within a family. Each of these processing elements possesses ALUs. For devices in the Northern Islands and Southern Islands families, these ALUs are arranged in four (in the Evergreen family, there are five) processing elements with arrays of 16 ALUs. Each of these arrays executes a single instruction across each lane for each of a block of 16 work-items. That instruction is repeated over four cycles to make the 64-element vector called a *wavefron*t. On devices in the Southern Island family, the four processing elements execute code from four different wavefronts. On Northern Islands and Evergreen family devices, the four arrays execute instructions from one wavefront, so that each work-item issues four (for Northern Islands) or five (for Evergreen) instructions at once in a very-long-instruction-word (VLIW) packet.

Figure 1.3 shows a simplified block diagram of a generalized AMD GPU compute device.

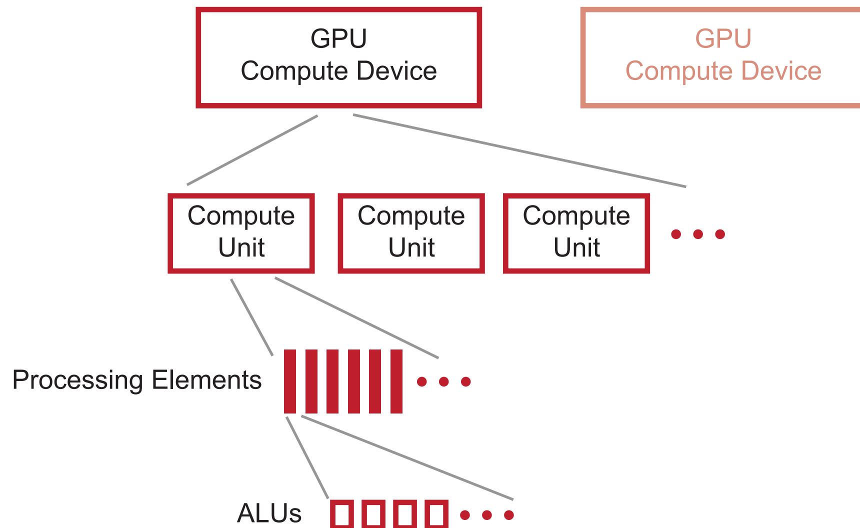

Figure 1.3 Generalized AMD GPU Compute Device Structure

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Figure 1.4 is a simplified diagram of an AMD GPU compute device. Different GPU compute devices have different characteristics (such as the number of compute units), but follow a similar design pattern.

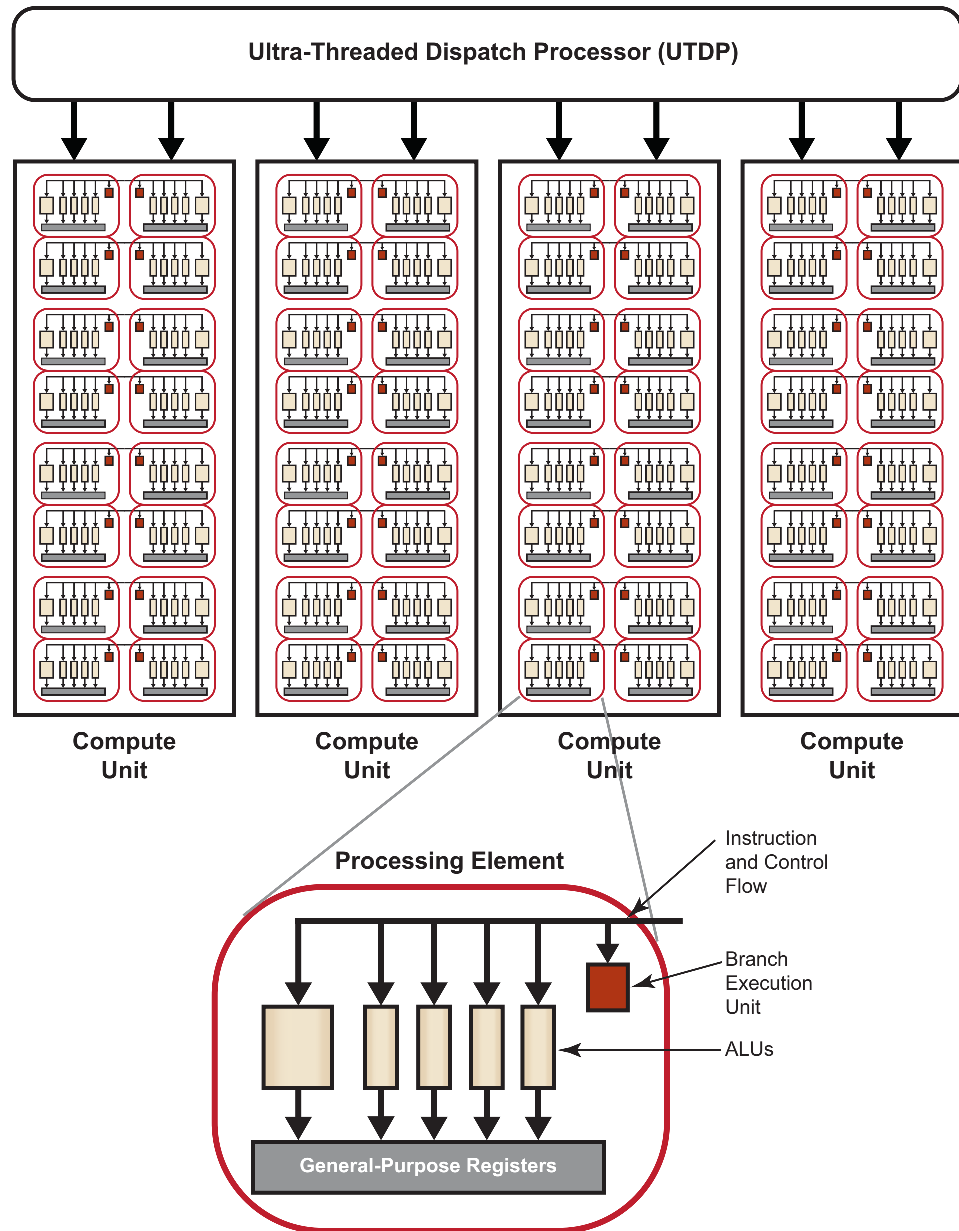

Figure 1.4 Simplified Block Diagram of and Evergreen-Family GPU[1]

GPU compute devices comprise groups of compute units. Each compute unit contains numerous processing elements, which are responsible for executing kernels, each operating on an independent data stream. Processing elements, in turn, contain numerous processing elements, which are the fundamental,

1. Much of this is transparent to the programmer.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

programmable ALUs that perform integer, single-precision floating-point, double-precision floating-point, and transcendental operations. All processing elements within a compute unit execute the same instruction sequence in lock-step for Evergreen and Northern Islands devices; different compute units can execute different instructions.

A processing element is arranged as a five-way or four-way (depending on the GPU type) very long instruction word (VLIW) processor (see bottom of Figure 1.4). Up to five scalar operations (or four, depending on the GPU type) can be co-issued in a VLIW instruction, each of which are executed on one of the corresponding five ALUs. ALUs can execute single-precision floating point or integer operations. One of the five ALUs also can perform transcendental operations (sine, cosine, logarithm, etc.). Double-precision floating point operations are processed (where supported) by connecting two or four of the ALUs (excluding the transcendental core) to perform a single double-precision operation. The processing element also contains one branch execution unit to handle branch instructions.

Different GPU compute devices have different numbers of processing elements. For example, the ATI Radeon™ HD 5870 GPU has 20 compute units, each with 16 processing elements, and each processing elements contains five ALUs; this yields 1600 physical ALUs.

1.4 The AMD Accelerated Parallel Processing Implementation of OpenCL

AMD Accelerated Parallel Processing harnesses the tremendous processing power of GPUs for high-performance, data-parallel computing in a wide range of applications. The AMD Accelerated Parallel Processing system includes a software stack and the AMD GPUs. Figure 1.5 illustrates the relationship of the AMD Accelerated Parallel Processing components.

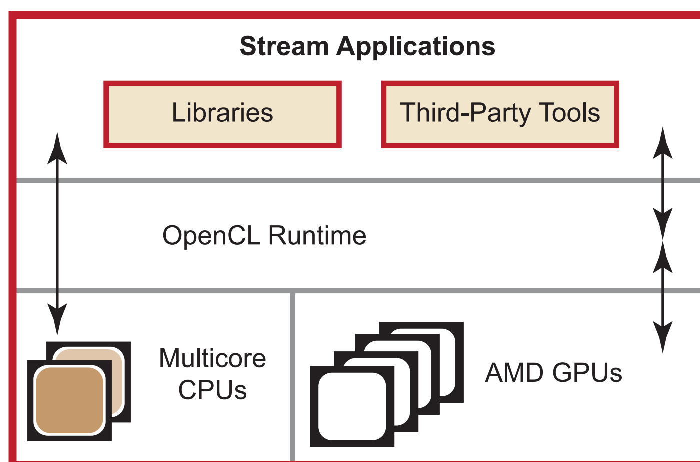

Figure 1.5 AMD Accelerated Parallel Processing Software Ecosystem

The AMD Accelerated Parallel Processing software stack provides end-users and developers with a complete, flexible suite of tools to leverage the processing

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

power in AMD GPUs. AMD Accelerated Parallel Processing software embraces open-systems, open-platform standards. The AMD Accelerated Parallel Processing open platform strategy enables AMD technology partners to develop and provide third-party development tools.

The software includes the following components:

- OpenCL compiler and runtime
- Performance Profiling Tools – AMD APP Profiler and AMD APP KernelAnalyzer.
- Performance Libraries – AMD Core Math Library (ACML) for optimized NDRange-specific algorithms.

The latest generations of AMD GPUs use unified shader architectures capable of running different kernel types interleaved on the same hardware. Programmable GPU compute devices execute various user-developed programs, known to graphics programmers as *shaders* and to compute programmers as *kernels*. These GPU compute devices can execute non-graphics functions using a data-parallel programming model that maps executions onto compute units. In this programming model, known as AMD Accelerated Parallel Processing, arrays of input data elements stored in memory are accessed by a number of compute units.

Each instance of a kernel running on a compute unit is called a work-item. A specified rectangular region of the output buffer to which work-items are mapped is known as the n-dimensional index space, called an *NDRange*.

The GPU schedules the range of work-items onto a group of processing elements, until all work-items have been processed. Subsequent kernels then can be executed, until the application completes. A simplified view of the AMD Accelerated Parallel Processing programming model and the mapping of work-items to processing elements is shown in Figure 1.6.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

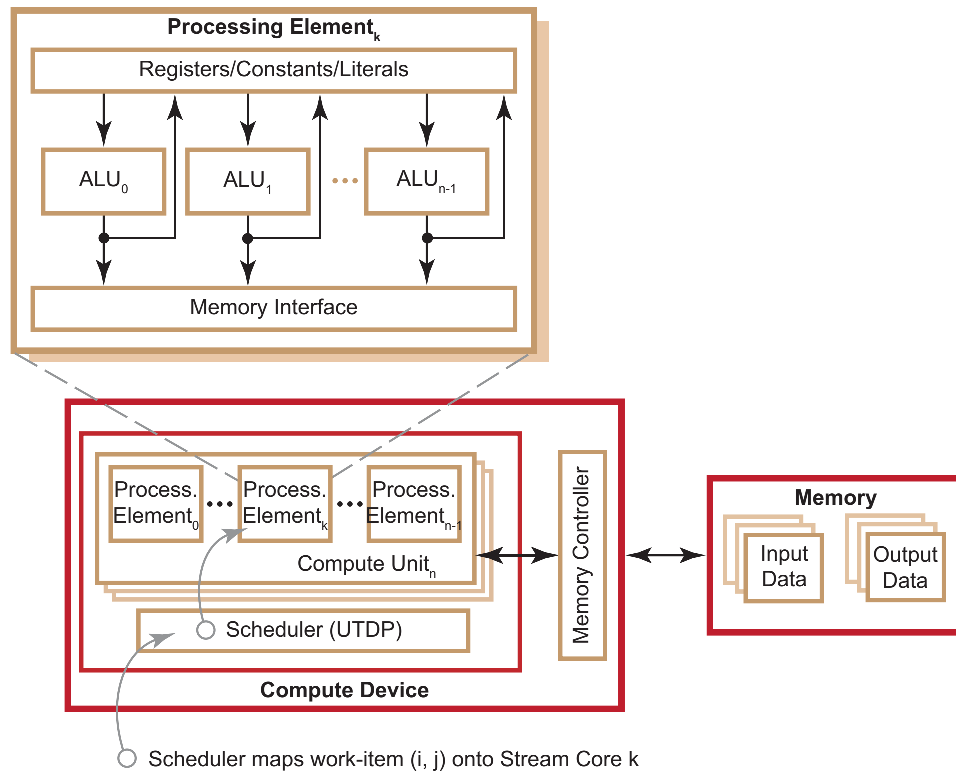

Figure 1.6 Simplified Mapping of OpenCL onto AMD Accelerated Parallel Processing for Evergreen and Northern Island Devices

OpenCL maps the total number of work-items to be launched onto an n-dimensional grid (ND-Range). The developer can specify how to divide these items into work-groups. AMD GPUs execute on wavefronts (groups of work-items executed in lock-step in a compute unit); there are an integer number of wavefronts in each work-group. Thus, as shown in Figure 1.7, hardware that schedules work-items for execution in the AMD Accelerated Parallel Processing environment includes the intermediate step of specifying wavefronts within a work-group. This permits achieving maximum performance from AMD GPUs. For a more detailed discussion of wavefronts, see Section 1.8.2, “Wavefronts and Work-groups,” page 1-18.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

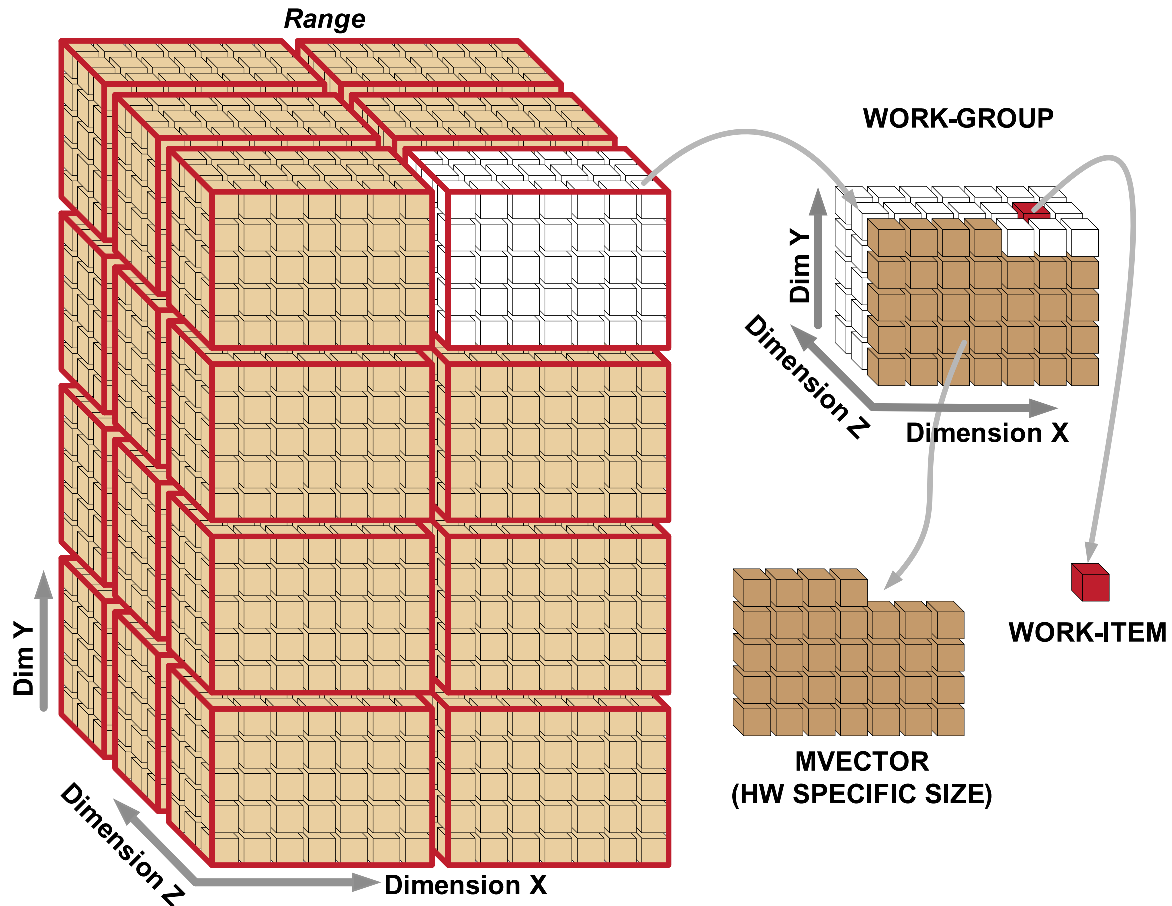

Figure 1.7 Work-Item Grouping Into Work-Groups and Wavefronts

1.4.1 Work-Item Processing

All stream cores within a compute unit execute the same instruction for each cycle. A work item can issue one VLIW instruction per clock cycle. The block of work-items that are executed together is called a *wavefront*. To hide latencies due to memory accesses and processing element operations, up to four work-items from the same wavefront are pipelined on the same stream core. For example, on the AMD Radeon™ HD 6970 GPU compute device, the 16 processing elements execute the same instructions for four cycles, which effectively appears as a 64-wide compute unit in execution width.

The size of wavefronts can differ on different GPU compute devices. For example, the AMD Radeon™ HD 54XX series graphics cards has a wavefront size of 32 work-items. Higher-end AMD GPUs have a wavefront size of 64 work-items.

Compute units operate independently of each other, so it is possible for different compute units to execute different instructions.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

1.4.2 Flow Control

Before discussing flow control, it is necessary to clarify the relationship of a wavefront to a work-group. If a user defines a work-group, it consists of one or more wavefronts. A wavefront is a hardware thread with its own program counter; it is capable of following control flow independently of other wavefronts. A wavefronts consists of 64 or fewer work-items, two wavefronts are between 65 to 128 work-items, etc., on a device with a wavefront size of 64. For optimum hardware usage, an integer multiple of 64 work-items is recommended.

Flow control, such as branching, is done by combining all necessary paths as a wavefront. If work-items within a wavefront diverge, all paths are executed serially. For example, if a work-item contains a branch with two paths, the wavefront first executes one path, then the second path. The total time to execute the branch is the sum of each path time. An important point is that even if only one work-item in a wavefront diverges, the rest of the work-items in the wavefront execute the branch. The number of work-items that must be executed during a branch is called the *branch granularity*. On AMD hardware, the branch granularity is the same as the wavefront granularity.

Masking of wavefronts is effected by constructs such as:

```
if(x)
  {
   .   //items within these braces = A
   .
   .
  }
else
  {
   .   //items within these braces = B
   .
   .
  }
```

The wavefront mask is set true for lanes (elements/items) in which `x` is true, then execute A. The mask then is inverted, and B is executed.

Example 1: If two branches, A and B, take the same amount of time *t* to execute over a wavefront, the total time of execution, if any work-item diverges, is *2t*.

Loops execute in a similar fashion, where the wavefront occupies a compute unit as long as there is at least one work-item in the wavefront still being processed. Thus, the total execution time for the wavefront is determined by the work-item with the longest execution time.

Example 2: If *t* is the time it takes to execute a single iteration of a loop; and within a wavefront all work-items execute the loop one time, except for a single work-item that executes the loop 100 times, the time it takes to execute that entire wavefront is *100t*.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

1.4.3 Work-Item Creation

For each work-group, the GPU compute device spawns the required number of wavefronts on a single compute unit. If there are non-active work-items within a wavefront, the stream cores that would have been mapped to those work-items are idle. An example is a work-group that is a non-multiple of a wavefront size (for example: if the work-group size is 32, the wavefront is half empty and unused).

1.5 Memory Architecture and Access

OpenCL has four memory domains: private, local, global, and constant; the AMD Accelerated Parallel Processing system also recognizes host (CPU) and PCI Express® (PCIe®) memory.

- private memory - specific to a work-item; it is not visible to other work-items.
- local memory - specific to a work-group; accessible only by work-items belonging to that work-group.
- global memory - accessible to all work-items executing in a context, as well as to the host (read, write, and map commands).
- constant memory - read-only region for host-allocated and -initialized objects that are not changed during kernel execution.
- host (CPU) memory - host-accessible region for an application's data structures and program data.
- PCIe memory - part of host (CPU) memory accessible from, and modifiable by, the host program and the GPU compute device. Modifying this memory requires synchronization between the GPU compute device and the CPU.

Figure 1.8 illustrates the interrelationship of the memories. (Note that the referenced color buffer is a write-only output buffer in a pixel shader that has a predetermined location based on the pixel location.)

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

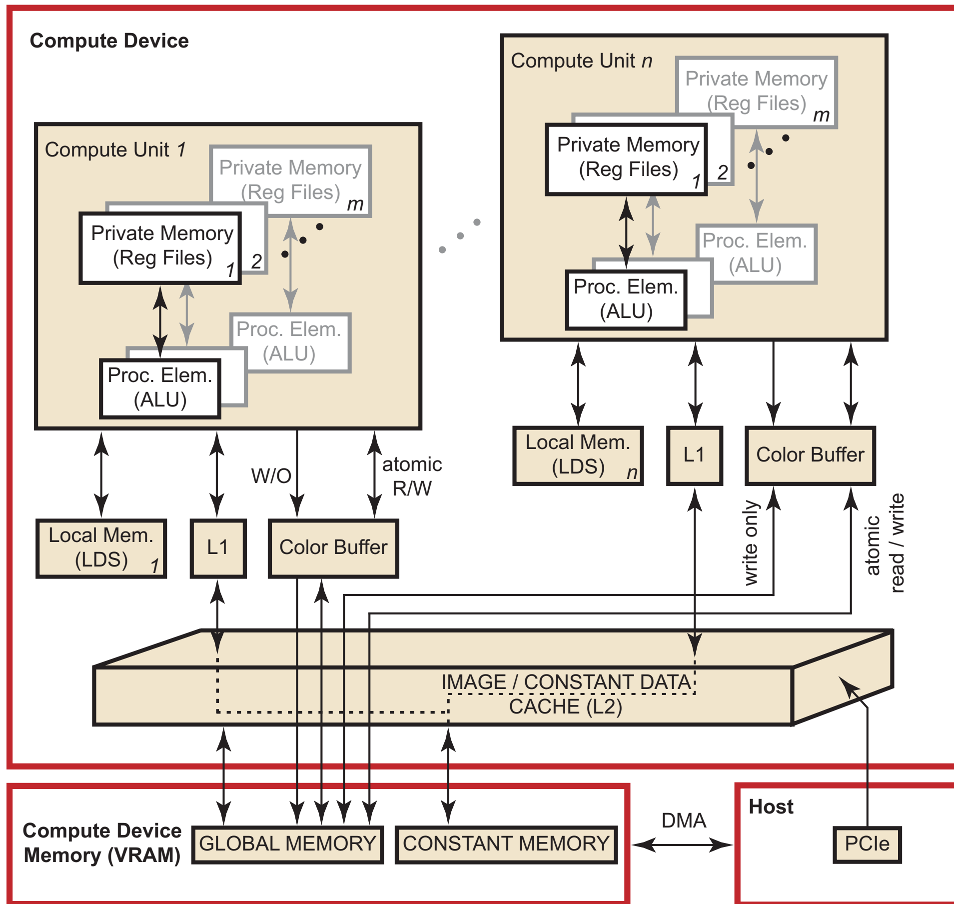

Figure 1.8 Interrelationship of Memory Domains for Southern Islands Devices

Figure 1.9 illustrates the standard dataflow between host (CPU) and GPU.

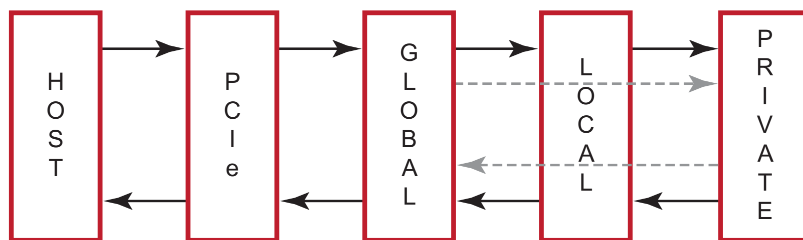

Figure 1.9 Dataflow between Host and GPU

There are two ways to copy data from the host to the GPU compute device memory:

- Implicitly by using `clEnqueueMapBuffer` and `clEnqueueUnMapMemObject`.
- Explicitly through `clEnqueueReadBuffer` and `clEnqueueWriteBuffer` (`clEnqueueReadImage`, `clEnqueueWriteImage`.).

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

When using these interfaces, it is important to consider the amount of copying involved. There is a two-copy processes: between host and PCIe, and between PCIe and GPU compute device. This is why there is a large performance difference between the system GFLOPS and the kernel GFLOPS.

With proper memory transfer management and the use of system pinned memory (host/CPU memory remapped to the PCIe memory space), copying between host (CPU) memory and PCIe memory can be skipped. Note that this is not an easy API call to use and comes with many constraints, such as page boundary and memory alignment.

Double copying lowers the overall system memory bandwidth. In GPU compute device programming, pipelining and other techniques help reduce these bottlenecks. See Chapter 4, Chapter 5, and Chapter 6 for more specifics about optimization techniques.

1.5.1 Memory Access

Using local memory (known as local data store, or LDS, as shown in Figure 1.8) typically is an order of magnitude faster than accessing host memory through global memory (VRAM), which is one order of magnitude faster again than PCIe. However, stream cores do not directly access memory; instead, they issue memory requests through dedicated hardware units. When a work-item tries to access memory, the work-item is transferred to the appropriate fetch unit. The work-item then is deactivated until the access unit finishes accessing memory. Meanwhile, other work-items can be active within the compute unit, contributing to better performance. The data fetch units handle three basic types of memory operations: loads, stores, and streaming stores. GPU compute devices can store writes to random memory locations using global buffers.

1.5.2 Global Buffer

The global buffer lets applications read from, and write to, arbitrary locations in memory. When using a global buffer, memory-read and memory-write operations from the stream kernel are done using regular GPU compute device instructions with the global buffer used as the source or destination for the instruction. The programming interface is similar to load/store operations used with CPU programs, where the relative address in the read/write buffer is specified.

1.5.3 Image Read/Write

Image reads are done by addressing the desired location in the input memory using the fetch unit. The fetch units can process either 1D or 2 D addresses. These addresses can be *normalized* or *un-normalized*. Normalized coordinates are between 0.0 and 1.0 (inclusive). For the fetch units to handle 2D addresses and normalized coordinates, pre-allocated memory segments must be bound to the fetch unit so that the correct memory address can be computed. For a single kernel invocation, up to 128 images can be bound at once for reading, and eight for writing. The maximum number of 2D addresses is 8192 x 8192.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Image reads are cached through the texture system (corresponding to the L2 and L1 caches).

1.5.4 Memory Load/Store

When using a global buffer, each work-item can write to an arbitrary location within the global buffer. Global buffers use a linear memory layout. If consecutive addresses are written, the compute unit issues a burst write for more efficient memory access. Only read-only buffers, such as constants, are cached.

1.6 Communication Between Host and GPU in a Compute Device

The following subsections discuss the communication between the host (CPU) and the GPU in a compute device. This includes an overview of the PCIe bus, processing API calls, and DMA transfers.

1.6.1 PCI Express Bus

Communication and data transfers between the system and the GPU compute device occur on the PCIe channel. AMD Accelerated Parallel Processing graphics cards use PCIe 2.0 x16 (second generation, 16 lanes). Generation 1 x16 has a theoretical maximum throughput of 4 GBps in each direction. Generation 2 x16 doubles the throughput to 8 GBps in each direction. Southern Islands AMD GPUs support PCIe 3.0 with a theoretical peak performance of 16 GBps. Actual transfer performance is CPU and chipset dependent.

Transfers from the system to the GPU compute device are done either by the *command processor* or by the *DMA engine*. The GPU compute device also can read and write system memory directly from the compute unit through kernel instructions over the PCIe bus.

1.6.2 Processing API Calls: The Command Processor

The host application does not interact with the GPU compute device directly. A driver layer translates and issues commands to the hardware on behalf of the application.

Most commands to the GPU compute device are buffered in a command queue on the host side. The command queue is sent to the GPU compute device, and the commands are processed by it. There is no guarantee as to when commands from the command queue are executed, only that they are executed in order. Unless the GPU compute device is busy, commands are executed immediately.

Command queue elements include:

- Kernel execution calls
- Kernels
- Constants
- Transfers between device and host

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

1.6.3 DMA Transfers

Direct Memory Access (DMA) memory transfers can be executed separately from the command queue using the DMA engine on the GPU compute device. DMA calls are executed immediately; and the order of DMA calls and command queue flushes is guaranteed.

DMA transfers can occur asynchronously. This means that a DMA transfer is executed concurrently with other system or GPU compute operations when there are no dependencies. However, data is not guaranteed to be ready until the DMA engine signals that the event or transfer is completed. The application can query the hardware for DMA event completion. If used carefully, DMA transfers are another source of parallelization.

1.6.4 Masking Visible Devices

By default, OpenCL applications are exposed to all GPUs installed in the system; this allows applications to use multiple GPUs to run the compute task.

In some cases, the user might want to mask the visibility of the GPUs seen by the OpenCL application. One example is to dedicate one GPU for regular graphics operations and the other three (in a four-GPU system) for Compute. To do that, set the `GPU_DEVICE_ORDINAL` environment parameter, which is a comma-separated list variable:

- Under Windows: set GPU_DEVICE_ORDINAL=1,2,3
- Under Linux: export GPU_DEVICE_ORDINAL=1,2,3

Another example is a system with eight GPUs, where two distinct OpenCL applications are running at the same time. The administrator might want to set `GPU_DEVICE_ORDINAL` to 0,1,2,3 for the first application, and 4,5,6,7 for the second application; thus, partitioning the available GPUs so that both applications can run at the same time.

1.7 GPU Compute Device Scheduling

GPU compute devices are very efficient at parallelizing large numbers of work-items in a manner transparent to the application. Each GPU compute device uses the large number of wavefronts to hide memory access latencies by having the resource scheduler switch the active wavefront in a given compute unit whenever the current wavefront is waiting for a memory access to complete. Hiding memory access latencies requires that each work-item contain a large number of ALU operations per memory load/store.

Figure 1.10 shows the timing of a simplified execution of work-items in a single stream core. At time 0, the work-items are queued and waiting for execution. In this example, only four work-items (T0…T3) are scheduled for the compute unit. The hardware limit for the number of active work-items is dependent on the resource usage (such as the number of active registers used) of the program

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

being executed. An optimally programmed GPU compute device typically has thousands of active work-items.

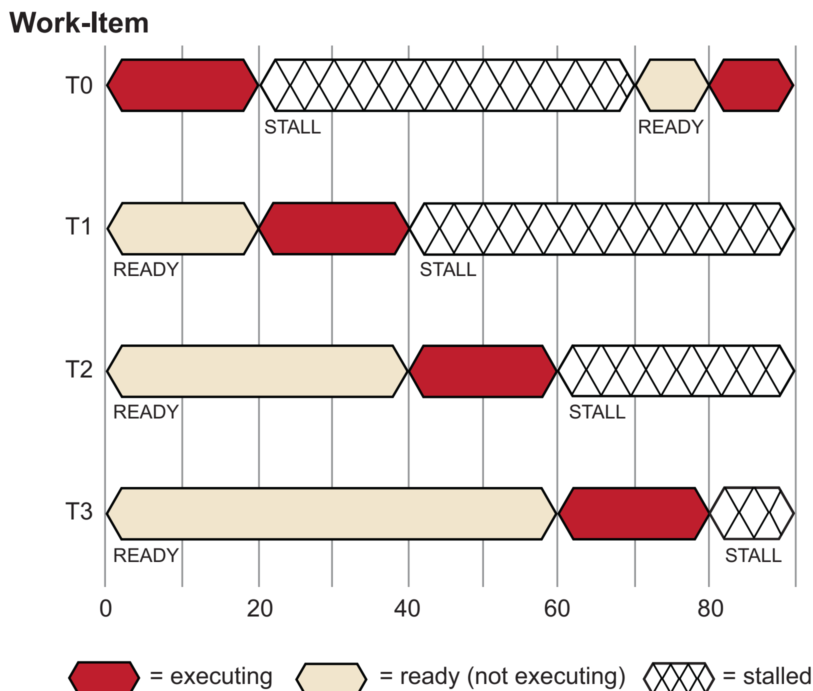

Figure 1.10 Simplified Execution Of Work-Items On A Single Stream Core

At runtime, work-item T0 executes until cycle 20; at this time, a stall occurs due to a memory fetch request. The scheduler then begins execution of the next work-item, T1. Work-item T1 executes until it stalls or completes. New work-items execute, and the process continues until the available number of active work-items is reached. The scheduler then returns to the first work-item, T0.

If the data work-item T0 is waiting for has returned from memory, T0 continues execution. In the example in Figure 1.10, the data is ready, so T0 continues. Since there were enough work-items and processing element operations to cover the long memory latencies, the stream core does not idle. This method of memory latency hiding helps the GPU compute device achieve maximum performance.

If none of T0 – T3 are runnable, the stream core waits (stalls) until one of T0 – T3 is ready to execute. In the example shown in Figure 1.11, T0 is the first to continue execution.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

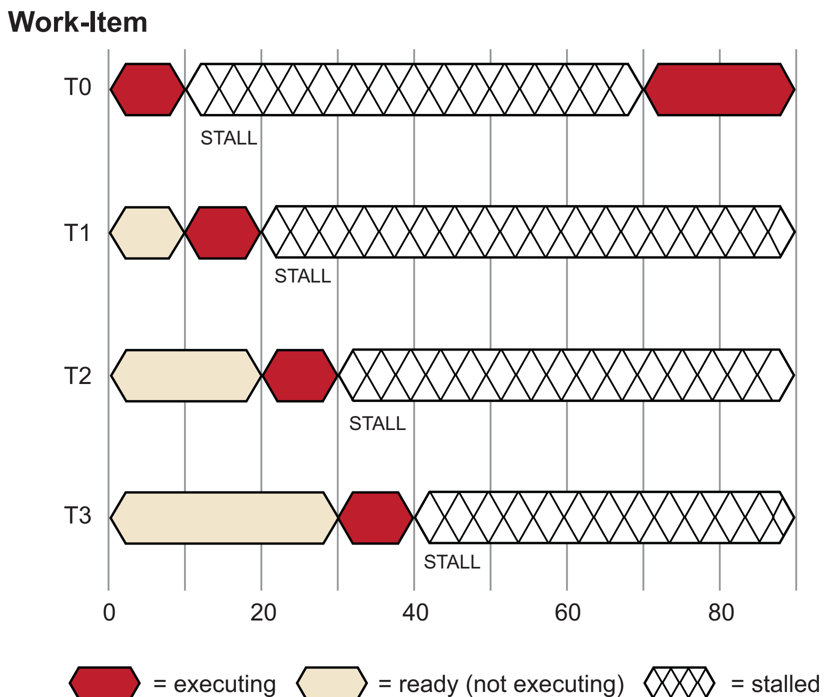

Figure 1.11 Stream Core Stall Due to Data Dependency

The causes for this situation are discussed in the following sections.

1.8 Terminology

1.8.1 Compute Kernel

To define a *compute kernel*, it is first necessary to define a kernel. A *kernel* is a small unit of execution that performs a clearly defined function and that can be executed in parallel. Such a kernel can be executed on each element of an input stream (called an NDRange), or simply at each point in an arbitrary index space. A kernel is analogous and, on some devices identical, to what graphics programmers call a shader program. This kernel is not to be confused with an OS kernel, which controls hardware. The most basic form of an NDRange is simply mapped over input data and produces one output item for each input tuple. Subsequent extensions of the basic model provide random-access functionality, variable output counts, and reduction/accumulation operations. Kernels are specified using the kernel keyword.

A compute kernel is a specific type of kernel that is not part of the traditional graphics pipeline. The compute kernel type can be used for graphics, but its strength lies in using it for non-graphics fields such as physics, AI, modeling, HPC, and various other computationally intensive applications.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

1.8.1.1 Work-Item Spawn Order

In a compute kernel, the work-item spawn order is sequential. This means that on a chip with N work-items per wavefront, the first N work-items go to wavefront 1, the second N work-items go to wavefront 2, etc. Thus, the work-item IDs for wavefront K are in the range (K•N) to ((K+1)•N) - 1.

1.8.2 Wavefronts and Work-groups

Wavefronts and work-groups are two concepts relating to compute kernels that provide data-parallel granularity. A wavefront executes a number of work-items in lock step relative to each other. Sixteen work-items are execute in parallel across the vector unit, and the whole wavefront is covered over four clock cycles. It is the lowest level that flow control can affect. This means that if two work-items inside of a wavefront go divergent paths of flow control, all work-items in the wavefront go to both paths of flow control.

Grouping is a higher-level granularity of data parallelism that is enforced in software, not hardware. Synchronization points in a kernel guarantee that all work-items in a work-group reach that point (barrier) in the code before the next statement is executed.

Work-groups are composed of wavefronts. Best performance is attained when the group size is an integer multiple of the wavefront size.

1.8.3 Local Data Store (LDS)

The LDS is a high-speed, low-latency memory private to each compute unit. It is a full gather/scatter model: a work-group can write anywhere in its allocated space. This model is unchanged for the AMD Radeon™ HD 7XXX series. The constraints of the current LDS model are:

1. The LDS size is allocated per work-group. Each work-group specifies how much of the LDS it requires. The hardware scheduler uses this information to determine which work groups can share a compute unit.
2. Data can only be shared within work-items in a work-group.
3. Memory accesses outside of the work-group result in undefined behavior.

1.9 Programming Model

The OpenCL programming model is based on the notion of a host device, supported by an application API, and a number of devices connected through a bus. These are programmed using OpenCL C. The host API is divided into platform and runtime layers. OpenCL C is a C-like language with extensions for parallel programming such as memory fence operations and barriers. Figure 1.12 illustrates this model with queues of commands, reading/writing data, and executing kernels for specific devices.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

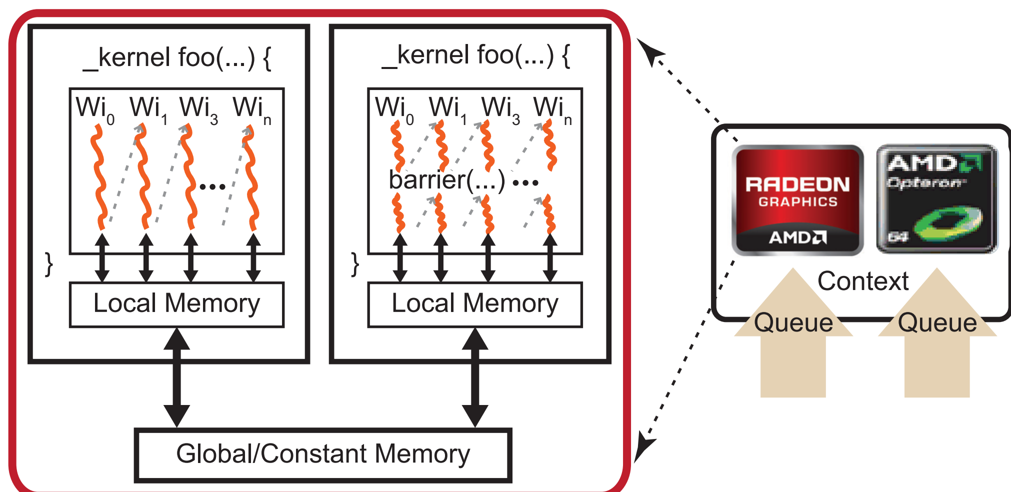

Figure 1.12 OpenCL Programming Model

The devices are capable of running data- and task-parallel work. A kernel can be executed as a function of multi-dimensional domains of indices. Each element is called a *work-item*; the total number of indices is defined as the *global work-size*. The global work-size can be divided into sub-domains, called work-groups, and individual work-items within a group can communicate through global or locally shared memory. Work-items are synchronized through barrier or fence operations. Figure 1.12 is a representation of the host/device architecture with a single platform, consisting of a GPU and a CPU.

An OpenCL application is built by first querying the runtime to determine which platforms are present. There can be any number of different OpenCL implementations installed on a single system. The desired OpenCL platform can be selected by matching the platform vendor string to the desired vendor name, such as "Advanced Micro Devices, Inc." The next step is to create a context. As shown in Figure 1.12, an OpenCL context has associated with it a number of compute devices (for example, CPU or GPU devices),. Within a context, OpenCL guarantees a relaxed consistency between these devices. This means that memory objects, such as buffers or images, are allocated per context; but changes made by one device are only guaranteed to be visible by another device at well-defined synchronization points. For this, OpenCL provides events, with the ability to synchronize on a given event to enforce the correct order of execution.

Many operations are performed with respect to a given context; there also are many operations that are specific to a device. For example, program compilation and kernel execution are done on a per-device basis. Performing work with a device, such as executing kernels or moving data to and from the device's local memory, is done using a corresponding command queue. A command queue is associated with a single device and a given context; all work for a specific device is done through this interface. Note that while a single command queue can be associated with only a single device, there is no limit to the number of command queues that can point to the same device. For example, it is possible to have

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

one command queue for executing kernels and a command queue for managing data transfers between the host and the device.

Most OpenCL programs follow the same pattern. Given a specific platform, select a device or devices to create a context, allocate memory, create device-specific command queues, and perform data transfers and computations. Generally, the platform is the gateway to accessing specific devices, given these devices and a corresponding context, the application is independent of the platform. Given a context, the application can:

- Create one or more command queues.
- Create programs to run on one or more associated devices.
- Create kernels within those programs.
- Allocate memory buffers or images, either on the host or on the device(s). (Memory can be copied between the host and device.)
- Write data to the device.
- Submit the kernel (with appropriate arguments) to the command queue for execution.
- Read data back to the host from the device.

The relationship between context(s), device(s), buffer(s), program(s), kernel(s), and command queue(s) is best seen by looking at sample code.

1.10 Example Programs

The following subsections provide simple programming examples with explanatory comments.

1.10.1 First Example: Simple Buffer Write

This sample shows a minimalist OpenCL C program that sets a given buffer to some value. It illustrates the basic programming steps with a minimum amount of code. This sample contains no error checks and the code is not generalized. Yet, many simple test programs might look very similar. The entire code for this sample is provided at the end of this section.

1. The host program must select a platform, which is an abstraction for a given OpenCL implementation. Implementations by multiple vendors can coexist on a host, and the sample uses the first one available.
2. A device id for a GPU device is requested. A CPU device could be requested by using `CL_DEVICE_TYPE_CPU` instead. The device can be a physical device, such as a given GPU, or an abstracted device, such as the collection of all CPU cores on the host.
3. On the selected device, an OpenCL context is created. A context ties together a device, memory buffers related to that device, OpenCL programs, and command queues. Note that buffers related to a device can reside on

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

either the host or the device. Many OpenCL programs have only a single context, program, and command queue.

4. Before an OpenCL kernel can be launched, its program source is compiled, and a handle to the kernel is created.
5. A memory buffer is allocated in the context.
6. The kernel is launched. While it is necessary to specify the global work size, OpenCL determines a good local work size for this device. Since the kernel was launch asynchronously, `clFinish()` is used to wait for completion.
7. The data is mapped to the host for examination. Calling `clEnqueueMapBuffer` ensures the visibility of the buffer on the host, which in this case probably includes a physical transfer. Alternatively, we could use `clEnqueueWriteBuffer()`, which requires a pre-allocated host-side buffer.

Example Code 1 –

```
//
// Copyright (c) 2010 Advanced Micro Devices, Inc. All rights reserved.
//

// A minimalist OpenCL program.

#include <CL/cl.h>
#include <stdio.h>

#define NWITEMS 512

// A simple memset kernel

const char *source =
"__kernel void memset( __global uint *dst )                                \n"
"{                                                                         \n"
"    dst[get_global_id(0)] = get_global_id(0);                             \n"
"}                                                                         \n";

int main(int argc, char ** argv)
{
   // 1. Get a platform.

   cl_platform_id platform;

   clGetPlatformIDs( 1, &platform, NULL );

    // 2. Find a gpu device.

   cl_device_id device;

   clGetDeviceIDs( platform, CL_DEVICE_TYPE_GPU,
                             1,
                             &device,
                             NULL);
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

```
    // 3. Create a context and command queue on that device.

    cl_context context = clCreateContext( NULL,
                                          1,
                                          &device,
                                          NULL, NULL, NULL);

    cl_command_queue queue = clCreateCommandQueue( context,
                                                   device,
                                                   0, NULL );

    // 4. Perform runtime source compilation, and obtain kernel entry point.

    cl_program program = clCreateProgramWithSource( context,
                                                    1,
                                                    &source,
                                                    NULL, NULL );

    clBuildProgram( program, 1, &device, NULL, NULL, NULL );

    cl_kernel kernel = clCreateKernel( program, "memset", NULL );

    // 5. Create a data buffer.

    cl_mem buffer = clCreateBuffer( context,
                                    CL_MEM_WRITE_ONLY,
                                    NWITEMS * sizeof(cl_uint),
                                    NULL, NULL );

    // 6. Launch the kernel. Let OpenCL pick the local work size.

    size_t global_work_size = NWITEMS;

    clSetKernelArg(kernel, 0, sizeof(buffer), (void*) &buffer);

    clEnqueueNDRangeKernel( queue,
                            kernel,
                            1,
                            NULL,
                            &global_work_size,
                            NULL, 0, NULL, NULL);

    clFinish( queue );

    // 7. Look at the results via synchronous buffer map.

    cl_uint *ptr;
    ptr = (cl_uint *) clEnqueueMapBuffer( queue,
                                          buffer,
                                          CL_TRUE,
                                          CL_MAP_READ,
                                          0,
                                          NWITEMS * sizeof(cl_uint),
                                          0, NULL, NULL, NULL );

    int i;

    for(i=0; i < NWITEMS; i++)
       printf("%d %d\n", i, ptr[i]);

    return 0;
}
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

1.10.2 Example: Parallel Min() Function

This medium-complexity sample shows how to implement an efficient parallel min() function.

The code is written so that it performs very well on either CPU or GPU. The number of threads launched depends on how many hardware processors are available. Each thread walks the source buffer, using a device-optimal access pattern selected at runtime. A multi-stage reduction using `__local` and `__global` atomics produces the single result value.

The sample includes a number of programming techniques useful for simple tests. Only minimal error checking and resource tear-down is used.

Runtime Code –

1. The source memory buffer is allocated, and initialized with a random pattern. Also, the actual min() value for this data set is serially computed, in order to later verify the parallel result.
2. The compiler is instructed to dump the intermediate IL and ISA files for further analysis.
3. The main section of the code, including device setup, CL data buffer creation, and code compilation, is executed for each device, in this case for CPU and GPU. Since the source memory buffer exists on the host, it is shared. All other resources are device-specific.
4. The global work size is computed for each device. A simple heuristic is used to ensure an optimal number of threads on each device. For the CPU, a given CL implementation can translate one work-item per CL compute unit into one thread per CPU core.

   On the GPU, an initial multiple of the wavefront size is used, which is adjusted to ensure even divisibility of the input data over all threads. The value of 7 is a minimum value to keep all independent hardware units of the compute units busy, and to provide a minimum amount of memory latency hiding for a kernel with little ALU activity.
5. After the kernels are built, the code prints errors that occurred during kernel compilation and linking.
6. The main loop is set up so that the measured timing reflects the actual kernel performance. If a sufficiently large NLOOPS is chosen, effects from kernel launch time and delayed buffer copies to the device by the CL runtime are minimized. Note that while only a single `clFinish()` is executed at the end of the timing run, the two kernels are always linked using an *event* to ensure serial execution.

   The bandwidth is expressed as “number of input bytes processed.” For high-end graphics cards, the bandwidth of this algorithm is about an order of magnitude higher than that of the CPU, due to the parallelized memory subsystem of the graphics card.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

7. The results then are checked against the comparison value. This also establishes that the result is the same on both CPU and GPU, which can serve as the first verification test for newly written kernel code.
8. Note the use of the debug buffer to obtain some runtime variables. Debug buffers also can be used to create short execution traces for each thread, assuming the device has enough memory.
9. You can use the `Timer.cpp` and `Timer.h` files from the TransferOverlap sample, which is in the SDK samples.

Kernel Code –

10. The code uses four-component vectors (uint4) so the compiler can identify concurrent execution paths as often as possible. On the GPU, this can be used to further optimize memory accesses and distribution across ALUs. On the CPU, it can be used to enable SSE-like execution.
11. The kernel sets up a memory access pattern based on the device. For the CPU, the source buffer is chopped into continuous buffers: one per thread. Each CPU thread serially walks through its buffer portion, which results in good cache and prefetch behavior for each core.

    On the GPU, each thread walks the source buffer using a stride of the total number of threads. As many threads are executed in parallel, the result is a maximally coalesced memory pattern requested from the memory back-end. For example, if each compute unit has 16 physical processors, 16 uint4 requests are produced in parallel, per clock, for a total of 256 bytes per clock.
12. The kernel code uses a reduction consisting of three stages: `__global` to `__private`, `__private` to `_local`, which is flushed to `__global`, and finally `__global` to `__global`. In the first loop, each thread walks `__global` memory, and reduces all values into a min value in `__private` memory (typically, a register). This is the bulk of the work, and is mainly bound by `__global` memory bandwidth. The subsequent reduction stages are brief in comparison.
13. Next, all per-thread minimum values inside the work-group are reduced to a `__local` value, using an atomic operation. Access to the `__local` value is serialized; however, the number of these operations is very small compared to the work of the previous reduction stage. The threads within a work-group are synchronized through a `local barrier()`. The reduced min value is stored in `__global` memory.
14. After all work-groups are finished, a second kernel reduces all work-group values into a single value in `__global` memory, using an atomic operation. This is a minor contributor to the overall runtime.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Example Code 3 –

```
//
// Copyright (c) 2010 Advanced Micro Devices, Inc. All rights reserved.
//

#include <CL/cl.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "Timer.h"

#define NDEVS       2

// A parallel min() kernel that works well on CPU and GPU

const char *kernel_source =
"                                                                            \n"
"#pragma OPENCL EXTENSION cl_khr_local_int32_extended_atomics : enable       \n"
"#pragma OPENCL EXTENSION cl_khr_global_int32_extended_atomics : enable      \n"
"                                                                            \n"
"  // 9. The source buffer is accessed as 4-vectors.                         \n"
"                                                                            \n"
"__kernel void minp( __global uint4 *src,                                    \n"
"                    __global uint  *gmin,                                   \n"
"                    __local  uint  *lmin,                                   \n"
"                    __global uint  *dbg,                                    \n"
"                    int           nitems,                                  \n"
"                    uint           dev )                                    \n"
"{                                                                           \n"
"   // 10. Set up __global memory access pattern.                            \n"
"                                                                            \n"
"   uint  count  = ( nitems / 4 ) / get_global_size(0);                      \n"
"   uint  idx    = (dev == 0) ? get_global_id(0) * count                     \n"
"                             : get_global_id(0);                            \n"
"   uint  stride = (dev == 0) ? 1 : get_global_size(0);                      \n"
"   uint  pmin   = (uint) -1;                                                \n"
"                                                                            \n"
"   // 11. First, compute private min, for this work-item.                   \n"
"                                                                            \n"
"   for( int n=0; n < count; n++, idx += stride )                            \n"
"   {                                                                        \n"
"      pmin = min( pmin, src[idx].x );                                       \n"
"      pmin = min( pmin, src[idx].y );                                       \n"
"      pmin = min( pmin, src[idx].z );                                       \n"
"      pmin = min( pmin, src[idx].w );                                       \n"
"   }                                                                        \n"
"                                                                            \n"
"   // 12. Reduce min values inside work-group.                              \n"
"                                                                            \n"
"   if( get_local_id(0) == 0 )                                               \n"
"      lmin[0] = (uint) -1;                                                  \n"
"                                                                            \n"
"   barrier( CLK_LOCAL_MEM_FENCE );                                          \n"
"                                                                            \n"
"   (void) atom_min( lmin, pmin );                                           \n"
"                                                                            \n"
"   barrier( CLK_LOCAL_MEM_FENCE );                                          \n"
"                                                                            \n"
"   // Write out to __global.                                                \n"
"                                                                            \n"
"   if( get_local_id(0) == 0 )                                               \n"
"      gmin[ get_group_id(0) ] = lmin[0];                                    \n"
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

```
"                                                                      \n"
"   // Dump some debug information.                                    \n"
"                                                                      \n"
"   if( get_global_id(0) == 0 )                                        \n"
"   {                                                                  \n"
"      dbg[0] = get_num_groups(0);                                     \n"
"      dbg[1] = get_global_size(0);                                    \n"
"      dbg[2] = count;                                                 \n"
"      dbg[3] = stride;                                                \n"
"   }                                                                  \n"
"}                                                                     \n"
"                                                                      \n"
"// 13. Reduce work-group min values from __global to __global.        \n"
"                                                                      \n"
"__kernel void reduce( __global uint4 *src,                            \n"
"                      __global uint  *gmin )                          \n"
"{                                                                     \n"
"   (void) atom_min( gmin, gmin[get_global_id(0)] ) ;                  \n"
"}                                                                    \n";


int main(int argc, char ** argv)
{
   cl_platform_id   platform;

   int              dev, nw;
   cl_device_type   devs[NDEVS] = { CL_DEVICE_TYPE_CPU,
                                    CL_DEVICE_TYPE_GPU };

   cl_uint          *src_ptr;
   unsigned int     num_src_items = 4096*4096;

   // 1. quick & dirty MWC random init of source buffer.

   // Random seed (portable).

   time_t ltime;
   time(&ltime);

   src_ptr = (cl_uint *) malloc( num_src_items * sizeof(cl_uint) );

   cl_uint a =   (cl_uint) ltime,
           b =   (cl_uint) ltime;
   cl_uint min = (cl_uint) -1;

   // Do serial computation of min() for result verification.

   for( int i=0; i < num_src_items; i++ )
   {
      src_ptr[i] = (cl_uint) (b = ( a * ( b & 65535 )) + (  b >> 16 ));
      min = src_ptr[i] < min ? src_ptr[i] : min;
   }

// Get a platform.

   clGetPlatformIDs( 1, &platform, NULL );

   // 3. Iterate over devices.

   for(dev=0; dev < NDEVS; dev++)
   {
      cl_device_id     device;
      cl_context       context;
      cl_command_queue queue;
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

```
cl_program       program;
cl_kernel        minp;
cl_kernel        reduce;

cl_mem           src_buf;
cl_mem           dst_buf;
cl_mem           dbg_buf;

cl_uint          *dst_ptr,
                 *dbg_ptr;

printf("\n%s: ", dev == 0 ? "CPU" : "GPU");

// Find the device.

clGetDeviceIDs( platform,
                devs[dev],
                1,
                &device,
                NULL);

// 4. Compute work sizes.

cl_uint compute_units;
size_t  global_work_size;
size_t  local_work_size;
size_t  num_groups;

clGetDeviceInfo( device,
                 CL_DEVICE_MAX_COMPUTE_UNITS,
                 sizeof(cl_uint),
                 &compute_units,
                 NULL);

if( devs[dev] == CL_DEVICE_TYPE_CPU )
{
   global_work_size = compute_units * 1;      // 1 thread per core
   local_work_size = 1;
}
else
{
    cl_uint ws = 64;

    global_work_size = compute_units * 7 * ws; // 7 wavefronts per SIMD

    while( (num_src_items / 4) % global_work_size != 0 )
       global_work_size += ws;

    local_work_size = ws;
}

num_groups = global_work_size / local_work_size;

// Create a context and command queue on that device.

context = clCreateContext( NULL,
                           1,
                           &device,
                           NULL, NULL, NULL);

queue = clCreateCommandQueue(context,
                             device,
                             0, NULL);
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

```
// Minimal error check.

if( queue == NULL )
{
    printf("Compute device setup failed\n");
    return(-1);
}

// Perform runtime source compilation, and obtain kernel entry point.

program = clCreateProgramWithSource( context,
                                     1,
                                     &kernel_source,
                                     NULL, NULL );

//Tell compiler to dump intermediate .il and .isa GPU files.

// 5. Print compiler error messages

if(ret != CL_SUCCESS)
{
   printf("clBuildProgram failed: %d\n", ret);

   char buf[0x10000];

   clGetProgramBuildInfo( program,
                          device,
                          CL_PROGRAM_BUILD_LOG,
                          0x10000,
                          buf,
                          NULL);
   printf("\n%s\n", buf);
   return(-1);
}

minp   = clCreateKernel( program, "minp", NULL );
reduce = clCreateKernel( program, "reduce", NULL );

// Create input, output and debug buffers.

src_buf = clCreateBuffer( context,
                          CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                          num_src_items * sizeof(cl_uint),
                          src_ptr,
                          NULL );

dst_buf = clCreateBuffer( context,
                          CL_MEM_READ_WRITE,
                          num_groups * sizeof(cl_uint),
                          NULL, NULL );

dbg_buf = clCreateBuffer( context,
                          CL_MEM_WRITE_ONLY,
                          global_work_size * sizeof(cl_uint),
                          NULL, NULL );

clSetKernelArg(minp, 0, sizeof(void *),         (void*) &src_buf);
clSetKernelArg(minp, 1, sizeof(void *),         (void*) &dst_buf);
clSetKernelArg(minp, 2, 1*sizeof(cl_uint),      (void*) NULL);
clSetKernelArg(minp, 3, sizeof(void *),         (void*) &dbg_buf);
clSetKernelArg(minp, 4, sizeof(num_src_items), (void*) &num_src_items);
clSetKernelArg(minp, 5, sizeof(dev),            (void*) &dev);
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

```
clSetKernelArg(reduce, 0, sizeof(void *),      (void*) &src_buf);
clSetKernelArg(reduce, 1, sizeof(void *),      (void*) &dst_buf);

CPerfCounter t;
t.Reset();
t.Start();

// 6. Main timing loop.

#define NLOOPS 500

cl_event ev;
int nloops = NLOOPS;

while(nloops--)
{

   clEnqueueNDRangeKernel( queue,
                           minp,
                           1,
                           NULL,
                           &global_work_size,
                           &local_work_size,
                           0, NULL, &ev);

   clEnqueueNDRangeKernel( queue,
                           reduce,
                           1,
                           NULL,
                           &num_groups,
                           NULL, 1, &ev, NULL);
}

clFinish( queue );
t.Stop();

printf("B/W %.2f GB/sec, ", ((float) num_src_items *
                             sizeof(cl_uint) * NLOOPS) /
                            t.GetElapsedTime() / 1e9 );

// 7. Look at the results via synchronous buffer map.

dst_ptr = (cl_uint *) clEnqueueMapBuffer( queue,
                                          dst_buf,
                                          CL_TRUE,
                                          CL_MAP_READ,
                                          0,
                                          num_groups * sizeof(cl_uint),
                                          0, NULL, NULL, NULL );

dbg_ptr = (cl_uint *) clEnqueueMapBuffer( queue,
                                          dbg_buf,
                                          CL_TRUE,
                                          CL_MAP_READ,
                                          0,
                                          global_work_size *
                                          sizeof(cl_uint),
                                          0, NULL, NULL, NULL );

// 8. Print some debug info.

printf("%d groups, %d threads, count %d, stride %d\n", dbg_ptr[0],
                                                      dbg_ptr[1],
                                                      dbg_ptr[2],
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

```
                                                    dbg_ptr[3] );

      if( dst_ptr[0] == min )
         printf("result correct\n");
      else
         printf("result INcorrect\n");

   }

   printf("\n");
   return 0;
}
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Chapter 2
Building and Running OpenCL Programs

The compiler tool-chain provides a common framework for both CPUs and GPUs, sharing the front-end and some high-level compiler transformations. The back-ends are optimized for the device type (CPU or GPU). Figure 2.1 is a high-level diagram showing the general compilation path of applications using OpenCL. Functions of an application that benefit from acceleration are re-written in OpenCL and become the OpenCL source. The code calling these functions are changed to use the OpenCL API. The rest of the application remains unchanged. The kernels are compiled by the OpenCL compiler to either CPU binaries or GPU binaries, depending on the target device.

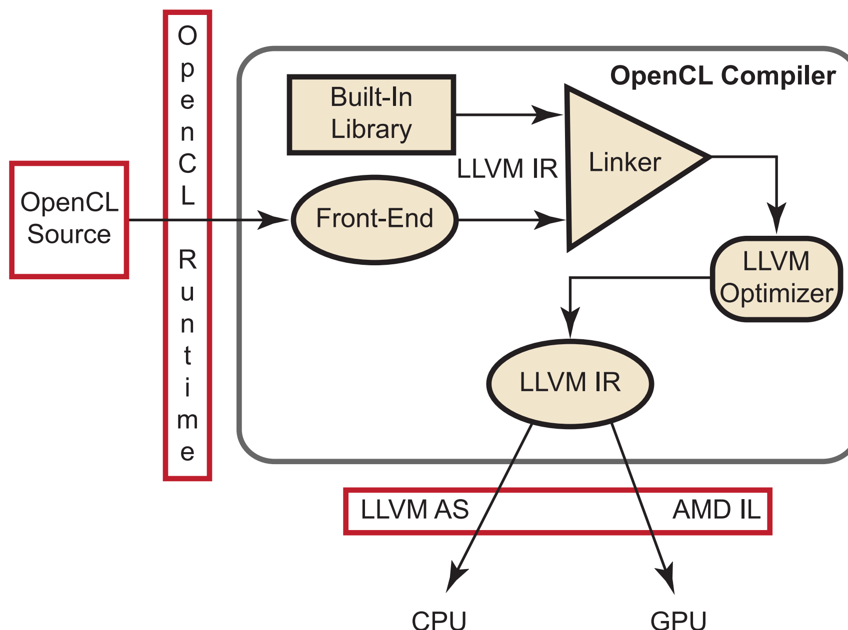

Figure 2.1 OpenCL Compiler Toolchain

For CPU processing, the OpenCL runtime uses the LLVM AS to generate x86 binaries. The OpenCL runtime automatically determines the number of processing elements, or cores, present in the CPU and distributes the OpenCL kernel between them.

For GPU processing, the OpenCL runtime post-processes the incomplete AMD IL from the OpenCL compiler and turns it into complete AMD IL. This adds macros (from a macro database, similar to the built-in library) specific to the

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

GPU. The OpenCL Runtime layer then removes unneeded functions and passes the complete IL to the CAL compiler for compilation to GPU-specific binaries.

2.1 Compiling the Program

An OpenCL application consists of a host program (C/C++) and an optional kernel program (`.cl`). To compile an OpenCL application, the host program must be compiled; this can be done using an off-the-shelf compiler such as g++ or MSVC++. The application kernels are compiled into device-specific binaries using the OpenCL compiler.

This compiler uses a standard C front-end, as well as the low-level virtual machine (LLVM) framework, with extensions for OpenCL. The compiler starts with the OpenCL source that the user program passes through the OpenCL runtime interface (Figure 2.1). The front-end translates the OpenCL source to LLVM IR. It keeps OpenCL-specific information as metadata structures. (For example, to debug kernels, the front end creates metadata structures to hold the debug information; also, a pass is inserted to translate this into LLVM debug nodes, which includes the line numbers and source code mapping.) The front-end supports additional data-types (int4, float8, etc.), additional keywords (kernel, global, etc.) and built-in functions (`get_global_id()`, `barrier()`, etc.). Also, it performs additional syntactic and semantic checks to ensure the kernels meet the OpenCL specification. The input to the LLVM linker is the output of the front-end and the library of built-in functions. This links in the built-in OpenCL functions required by the source and transfers the data to the optimizer, which outputs optimized LLVM IR.

For GPU processing, the LLVM IR-to-CAL IL module receives LLVM IR and generates optimized IL for a specific GPU type in an incomplete format, which is passed to the OpenCL runtime, along with some metadata for the runtime layer to finish processing.

For CPU processing, LLVM AS generates x86 binary.

2.1.1 Compiling on Windows

To compile OpenCL applications on Windows requires that Visual Studio 2008 Professional Edition (or later) or the Intel C (C++) compiler are installed. All C++ files must be added to the project, which must have the following settings.

- Project Properties → C/C++ → Additional Include Directories
  These must include `$(ATISTREAMSDKROOT)/include` for OpenCL headers. Optionally, they can include `$(ATISTREAMSDKSAMPLESROOT)/include` for SDKUtil headers.
- Project Properties → C/C++ → Preprocessor Definitions
  These must define `ATI_OS_WIN`.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

- Project Properties → Linker → Additional Library Directories
  These must include `$(ATISTREAMSDKROOT)/lib/x86` for OpenCL libraries. Optionally, they can include `$(ATISTREAMSDKSAMPLESROOT)/lib/x86` for SDKUtil libraries.
- Project Properties → Linker → Input → Additional Dependencies
  These must include `OpenCL.lib`. Optionally, they can include `SDKUtil.lib`.

2.1.2 Compiling on Linux

To compile OpenCL applications on Linux requires that the gcc or the Intel C compiler is installed. There are two major steps to do this: compiling and linking.

1. Compile all the C++ files (`Template.cpp`), and get the object files.
   For 32-bit object files on a 32-bit system, or 64-bit object files on 64-bit system:

```
g++ -o Template.o -DATI_OS_LINUX -c Template.cpp -I$ATISTREAMSDKROOT/include
```

   For building 32-bit object files on a 64-bit system:

```
g++ -o Template.o -DATI_OS_LINUX -c Template.cpp -I$ATISTREAMSDKROOT/include
```

2. Link all the object files generated in the previous step to the OpenCL library and create an executable.

   For linking to a 64-bit library:

```
g++ -o Template Template.o -lOpenCL -L$ATISTREAMSDKROOT/lib/x86_64
```

   For linking to a 32-bit library:

```
g++ -o Template Template.o -lOpenCL -L$ATISTREAMSDKROOT/lib/x86
```

The OpenCL samples in the SDK provided by AMD Accelerated Parallel Processing depend on the SDKUtil library. In Linux, the samples use the shipped `SDKUtil.lib`, whether or not the sample is built for release or debug. When compiling all samples from the `samples/opencl` folder, the `SDKUtil.lib` is created first; then, the samples use this generated library. When compiling the SDKUtil library, the created library replaces the shipped library.

The following are linking options if the samples depend on the SDKUtil Library (assuming the SDKUtil library is created in `$ATISTREAMSDKROOT/lib/x86_64` for 64-bit libraries, or `$ATISTREAMSDKROOT/lib/x86` for 32-bit libraries).

```
g++ -o Template Template.o -lSDKUtil -lOpenCL -L$ATISTREAMSDKROOT/lib/x86_64

g++ -o Template Template.o -lSDKUtil -lOpenCL -L$ATISTREAMSDKROOT/lib/x86
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

2.1.3 Supported Standard OpenCL Compiler Options

The currently supported options are:

- `-I dir` — Add the directory `dir` to the list of directories to be searched for header files. When parsing `#include` directives, the OpenCL compiler resolves relative paths using the current working directory of the application.
- `-D name` — Predefine *name* as a macro, with definition = 1. For `-D name=definition`, the contents of *definition* are tokenized and processed as if they appeared during the translation phase three in a `#define` directive. In particular, the definition is truncated by embedded newline characters. `-D` options are processed in the order they are given in the *options* argument to `clBuildProgram`.

2.1.4 AMD-Developed Supplemental Compiler Options

The following supported options are not part of the OpenCL specification:

- `-g` — This is an experimental feature that lets you use the GNU project debugger, GDB, to debug kernels on x86 CPUs running Linux or cygwin/minGW under Windows. For more details, see Chapter 3, "Debugging OpenCL." This option does not affect the default optimization of the OpenCL code.
- `-O0` — Specifies to the compiler not to optimize. This is equivalent to the OpenCL standard option `-cl-opt-disable`.
- `-f[no-]bin-source` — Does [not] generate OpenCL source in the `.source` section. For more information, see Appendix E, "OpenCL Binary Image Format (BIF) v2.0."
- `-f[no-]bin-llvmir` — Does [not] generate LLVM IR in the `.llvmir` section. For more information, see Appendix E, "OpenCL Binary Image Format (BIF) v2.0."
- `-f[no-]bin-amdil` — Does [not] generate AMD IL in the `.amdil` section. For more information, see Appendix E, "OpenCL Binary Image Format (BIF) v2.0."
- `-f[no-]bin-exe` — Does [not] generate the executable (ISA) in `.text` section. For more information, see Appendix E, "OpenCL Binary Image Format (BIF) v2.0."
- `-save-temps[=<prefix>]` — This option dumps intermediate temporary files, such as IL and ISA code, for each OpenCL kernel. If `<prefix>` is not given, temporary files are saved in the default temporary directory (the current directory for Linux, `C:\Users\<user>\AppData\Local` for Windows). If `<prefix>` is given, those temporary files are saved with the given `<prefix>`. If `<prefix>` is an absolute path prefix, such as `C:\your\work\dir\mydumpprefix`, those temporaries are saved under `C:\your\work\dir`, with `mydumpprefix` as prefix to all temporary names. For example,

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

`-save-temps`
under the default directory
`_temp_nn_xxx_yyy.il, _temp_nn_xxx_yyy.isa`

`-save-temps=aaa`
under the default directory
`aaa_nn_xxx_yyy.il, aaa_nn_xxx_yyy.isa`

`-save-temps=C:\you\dir\bbb`
under `C:\you\dir`
`bbb_nn_xxx_yyy.il, bbb_nn_xxx_yyy.isa`

where `xxx` and `yyy` are the device name and kernel name for this build, respectively, and `nn` is an internal number to identify a build to avoid overriding temporary files. Note that this naming convention is subject to change.

To avoid source changes, there are two environment variables that can be used to change CL options during the runtime.

- `AMD_OCL_BUILD_OPTIONS` — Overrides the CL options specified in `clBuildProgram()`.
- `AMD_OCL_BUILD_OPTIONS_APPEND` — Appends options to those specified in `clBuildProgram()`.

2.2 Running the Program

The runtime system assigns the work in the command queues to the underlying devices. Commands are placed into the queue using the `clEnqueue` commands shown in the listing below.

OpenCL API Function	Description
clCreateCommandQueue()	Create a command queue for a specific device (CPU, GPU).
clCreateProgramWithSource() clCreateProgramWithBinary()	Create a program object using the source code of the application kernels.
clBuildProgram()	Compile and link to create a program executable from the program source or binary.
clCreateKernel()	Creates a kernel object from the program object.
clCreateBuffer()	Creates a buffer object for use via OpenCL kernels.
clSetKernelArg() clEnqueueNDRangeKernel()	Set the kernel arguments, and enqueue the kernel in a command queue.
clEnqueueReadBuffer(), clEnqueueWriteBuffer()	Enqueue a command in a command queue to read from a buffer object to host memory, or write to the buffer object from host memory.
clEnqueueWaitForEvents()	Wait for the specified events to complete.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

The commands can be broadly classified into three categories.

- Kernel commands (for example, `clEnqueueNDRangeKernel()`, etc.),
- Memory commands (for example, `clEnqueueReadBuffer()`, etc.), and
- Event commands (for example, `clEnqueueWaitForEvents()`, etc.

As illustrated in Figure 2.2, the application can create multiple command queues (some in libraries, for different components of the application, etc.). These queues are muxed into one queue per device type. The figure shows command queues 1 and 3 merged into one CPU device queue (blue arrows); command queue 2 (and possibly others) are merged into the GPU device queue (red arrow). The device queue then schedules work onto the multiple compute resources present in the device. Here, K = kernel commands, M = memory commands, and E = event commands.

2.2.1 Running Code on Windows

The following steps ensure the execution of OpenCL applications on Windows.

1. The path to `OpenCL.lib` (`$ATISTREAMSDKROOT/lib/x86`) must be included in path environment variable.
2. Generally, the path to the kernel file (`Template_Kernel.cl`) specified in the host program is relative to the executable. Unless an absolute path is specified, the kernel file must be in the same directory as the executable.

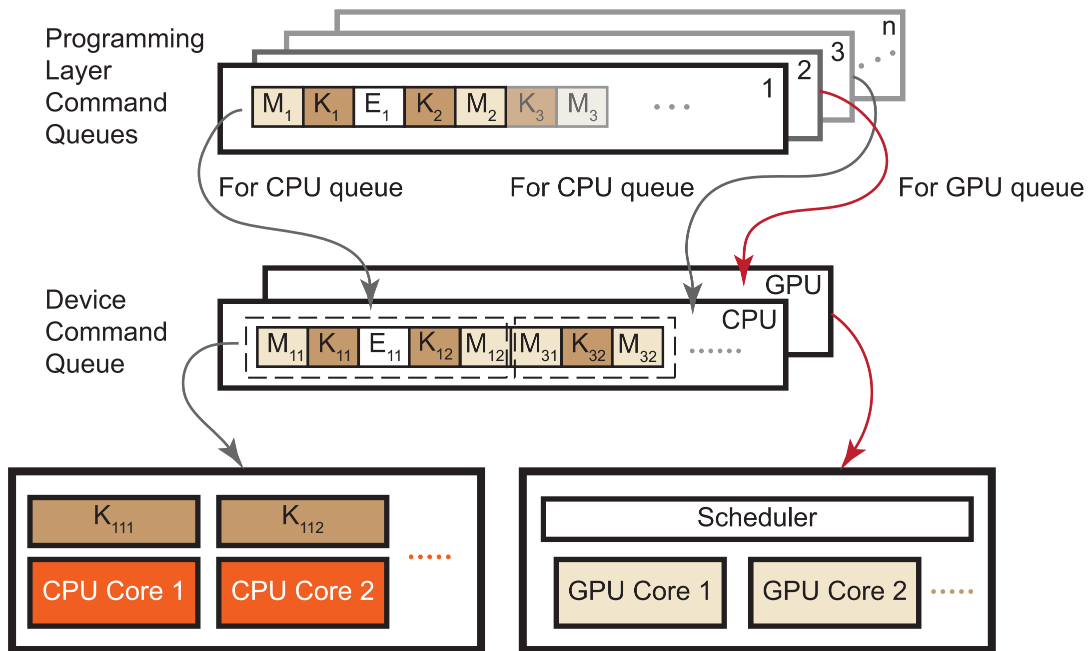

Figure 2.2 Runtime Processing Structure

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

2.2.2 Running Code on Linux

The following steps ensure the execution of OpenCL applications on Linux.

1. The path to `libOpenCL.so` (`$ATISTREAMSDKROOT/lib/x86`) must be included in `$LD_LIBRARY_PATH`.
2. `/usr/lib/OpenCL/vendors/` must have `libatiocl32.so` and/or `libatiocl64.so`.
3. Generally, the path to the kernel file (`Template_Kernel.cl`) specified in the host program is relative to the executable. Unless an absolute path is specified, the kernel file must be in the same directory as the executable.

2.3 Calling Conventions

For all Windows platforms, the `__stdcall` calling convention is used. Function names are undecorated.

For Linux, the calling convention is `__cdecl`.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Chapter 3 Debugging OpenCL

This chapter discusses how to debug OpenCL programs running on AMD Accelerated Parallel Processing GPU and CPU compute devices. The first, preferred, method is to debug with the AMD gDEBugger, as described in Section 3.1, "AMD gDEBugger." The second method, described in Section 3.2, "Debugging CPU Kernels with GDB," is to use experimental features provided by AMD Accelerated Parallel Processing (GNU project debugger, GDB) to debug kernels on x86 CPUs running Linux or cygwin/minGW under Windows.

3.1 AMD gDEBugger

gDEBugger 6.2 is available as an extension to Microsoft® Visual Studio®, a stand-alone version for Windows and a stand alone version for Linux. gDEBugger offers real-time OpenCL kernel debugging and memory analysis on GPU devices, allowing developers to access the kernel execution directly from the API call that issues it, debug inside the kernel, and view all variable values across the different work-groups and work-items. For Microsoft® Visual Studio®, it also provides OpenGL debugging and memory analysis. For information on downloading and installing gDEBugger, see:

http://developer.amd.com/tools/gDEBugger/Pages/default.aspx

After installing gDEBugger for Visual Studio, launch Visual Studio, and open the solution to be worked on. In the Visual Studio menu bar, note the new *gDEBugger* menu, which contains all the required controls.

Select a Visual C/C++ project, and set its debugging properties as normal. To add a breakpoint, either select *New gDEBugger Breakpoint* from the *gDEBugger* menu, or navigate to a kernel file used in the application and set a breakpoint on the desired source line. Then, select the *Launch OpenCL/OpenGL Debugging* from the *gDEBugger* menu to start debugging.

gDEBugger currently supports only API-level debugging and OpenCL kernel debugging; stepping through C/C++ code is not yet possible. However, the C/C++ call stack can be seen in the Visual Studio call stack view, which shows what led to the API function call.

To start kernel debugging, you can choose one of several options; one of these is to Step Into (F11) the appropriate `clEnqueueNDRangeKernel` function call. Once the kernel starts executing, debug it like C/C++ code, stepping into, out of, or over function calls in the kernel, setting source breakpoints, and inspecting the locals, autos, watch, and call stack views.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

To view OpenCL and OpenGL objects and their information, use the gDEBugger Explorer and gDEBugger Properties view. Additional views and features provide more detailed and advanced information on the OpenCL and OpenGL runtimes, their states, and the objects created within them.

For further information and more detailed usage instructions, see the *gDEBugger User Guide*:

http://developer.amd.com/tools/gDEBugger/webhelp/index.html

or the online help provided with gDEBugger.

3.2 Debugging CPU Kernels with GDB

This section describes an experimental feature for using the GNU project debugger, GDB, to debug kernels on x86 CPUs running Linux or cygwin/minGW under Windows.

3.2.1 Setting the Environment

The OpenCL program to be debugged first is compiled by passing the `"-g -O0"` (or `"-g -cl-opt-disable"`) option to the compiler through the options string to `clBuildProgram`. For example, using the C++ API:

```
err = program.build(devices,"-g -O0");
```

To avoid source changes, set the environment variable as follows:

`AMD_OCL_BUILD_OPTIONS_APPEND="-g -O0"` or
`AMD_OCL_BUILD_OPTIONS="-g -O0"`

Below is a sample debugging session of a program with a simple `hello world` kernel. The following GDB session shows how to debug this kernel. Ensure that the program is configured to be executed on the CPU. It is important to set `CPU_MAX_COMPUTE_UNITS=1`. This ensures that the program is executed deterministically.

3.2.2 Setting the Breakpoint in an OpenCL Kernel

To set a breakpoint, use:

`b [N | function | kernel_name]`

where *N* is the line number in the source code, *function* is the function name, and *kernel_name* is constructed as follows: if the name of the kernel is bitonicSort, the *kernel_name* is `__OpenCL_bitonicSort_kernel`.

Note that if no breakpoint is set, the program does not stop until execution is complete.

Also note that OpenCL kernel symbols are not visible in the debugger until the kernel is loaded. A simple way to check for known OpenCL symbols is to set a

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

breakpoint in the host code at `clEnqueueNDRangeKernel`, and to use the GDB `info functions __OpenCL` command, as shown in the example below.

3.2.3 Sample GDB Session

The following is a sample debugging session. Note that two separate breakpoints are set. The first is set in the host code, at `clEnqueueNDRangeKernel()`. The second breakpoint is set at the actual CL kernel function.

```
$ export AMD_OCL_BUILD_OPTIONS_APPEND="-g -O0"
$ export CPU_MAX_COMPUTE_UNITS=1
$ gdb BitonicSort
GNU gdb (GDB) 7.1-ubuntu
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/himanshu/Desktop/ati-stream-sdk-v2.3-
lnx64/samples/opencl/bin/x86_64/BitonicSort...done.
(gdb) b clEnqueueNDRangeKernel
Breakpoint 1 at 0x403228
(gdb) r --device cpu
Starting program: /home/himanshu/Desktop/ati-stream-sdk-v2.3-
lnx64/samples/opencl/bin/x86_64/BitonicSort --device cpu
[Thread debugging using libthread_db enabled]

Unsorted Input
53 5 199 15 120 9 71 107 71 242 84 150 134 180 26 128 196 9 98 4 102 65
206 35 224 2 52 172 160 94 2 214 99 .....


Platform Vendor : Advanced Micro Devices, Inc.
Device 0 : AMD Athlon(tm) II X4 630 Processor
[New Thread 0x7ffff7e6b700 (LWP 1894)]
[New Thread 0x7ffff2fcc700 (LWP 1895)]
Executing kernel for 1 iterations
-------------------------------------------

Breakpoint 1, 0x00007ffff77b9b20 in clEnqueueNDRangeKernel () from
/home/himanshu/Desktop/ati-stream-sdk-v2.3-lnx64/lib/x86_64/libOpenCL.so
(gdb) info functions __OpenCL
All functions matching regular expression "__OpenCL":

File OCLm2oVFr.cl:
void __OpenCL_bitonicSort_kernel(uint *, const uint, const uint, const
uint, const uint);

Non-debugging symbols:
0x00007ffff23c2dc0  __OpenCL_bitonicSort_kernel@plt
0x00007ffff23c2f40  __OpenCL_bitonicSort_stub
(gdb) b __OpenCL_bitonicSort_kernel
Breakpoint 2 at 0x7ffff23c2de9: file OCLm2oVFr.cl, line 32.
(gdb) c
Continuing.
[Switching to Thread 0x7ffff2fcc700 (LWP 1895)]

Breakpoint 2, __OpenCL_bitonicSort_kernel (theArray=0x615ba0, stage=0,
passOfStage=0, width=1024, direction=0) at OCLm2oVFr.cl:32
32      uint sortIncreasing = direction;
(gdb) p get_global_id(0)
$1 = 0
(gdb) c
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

```
Continuing.

Breakpoint 2, __OpenCL_bitonicSort_kernel (theArray=0x615ba0, stage=0,
passOfStage=0, width=1024, direction=0) at OCLm2oVFr.cl:32
32      uint sortIncreasing = direction;
(gdb) p get_global_id(0)
$2 = 1
(gdb)
```

3.2.4 Notes

1. To make a breakpoint in a working thread with some particular ID in dimension N, one technique is to set a conditional breakpoint when the `get_global_id(N) == ID`. To do this, use:

   `b [ N | function | kernel_name ] if (get_global_id(N)==ID)`

   where `N` can be 0, 1, or 2.

2. For complete GDB documentation, see http://www.gnu.org/software/gdb/documentation/ .

3. For debugging OpenCL kernels in Windows, a developer can use GDB running in cygwin or minGW. It is done in the same way as described in sections 3.1 and 3.2.

   Notes:

   – Only OpenCL kernels are visible to GDB when running cygwin or minGW. GDB under cygwin/minGW currently does not support host code debugging.

   – It is not possible to use two debuggers attached to the same process. Do not try to attach Visual Studio to a process, and concurrently GDB to the kernels of that process.

   – Continue to develop the application code using Visual Studio. Currently, gcc running in cygwin or minGW is not supported.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Chapter 4 OpenCL Performance and Optimization

This chapter discusses performance and optimization when programming for AMD Accelerated Parallel Processing (APP) GPU compute devices, as well as CPUs and multiple devices. Details specific to the Southern Islands series of GPUs is at the end of the chapter.

4.1 AMD APP Profiler

The AMD APP Profiler (hereafter Profiler) is a performance analysis tool that gathers data from the OpenCL run-time and AMD Radeon™ GPUs during the execution of an OpenCL application. This information is used to discover bottlenecks in the application and find ways to optimize the application's performance for AMD platforms. The Profiler can be installed as part of the AMD APP SDK installation, or separately using its own installer package. It is downloadable from:
http://developer.amd.com/tools/AMDAPPProfiler/Pages/default.aspx.

This section describes the major features of Profiler version 2.4. Because the Profiler is still being developed, please see the documentation for the latest features of the tool at the same URL provided above.

The Profiler supports two usage models.

- Plug-in for Microsoft Visual Studio 2008 or 2010 (recommended). This lets you visualize and analyze the results in multiple ways.
- Command-line utility tool for both Windows and Linux platforms. This is a way to collect data for applications without source code access. The results can be analyzed directly in the text format or visualized in the Visual Studio plug-in.

The Profiler supports two modes of operations.

- Collecting OpenCL application traces.
- Collecting OpenCL kernel GPU performance counters.

These are described in the following subsections.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

4.1.1 Collecting OpenCL Application Trace

The OpenCL application trace lists all the OpenCL API calls made by the application. For each of the API calls, the input parameters and output results are recorded, in addition to the CPU timestamps for the host code and device timestamps retrieved from the OpenCL run-time. The output data is recorded in an AMD custom application trace profile (`*.atp`) file format. See the Profiler documentation for more information.

This mode is especially useful for investigating the high-level structure of a complex application.

From the OpenCL application trace data, it is possible to:

- Reveal the high-level structure of the application with the Timeline view. This lets you investigate the number of OpenCL contexts and command queues created, as well as the relationships of these items in the application. The timeline shows the host code, kernel, and data transfer execution. See Section 4.1.1.1, “Timeline View,” page 4-2.
- Identify whether the application is bound by kernel execution or data transfer time; find the top ten most expensive kernels and data transfers; find the API hot spots (most frequently called or expensive API call) in the application with the Summary Pages view. See Section 4.1.1.2, “Summary Pages View,” page 4-4).
- View and debug the input parameters and output results for all API calls in the application with the API Trace view. See Section 4.1.1.3, “API Trace View,” page 4-5.
- An OpenCL Performance Marker (CLPerfMarker) library is also provided for visualizing and analyzing non-OpenCL host code on the Timeline. Users can instrument their code with calls to `clBeginPerfMarkerAMD()` and `clEndPerfMarkerAMD()`. These calls are then used by the Profiler to annotate the host-code timeline hierarchically. For more information, see the `CLPerfMarkerAMD.pdf` in the `CLPerfMarker/Doc` subdirectory under the Profiler installation directory, typically `$AMDAPPSDKROOT/Tools/AMDAPPProfiler-vx.x/`.

4.1.1.1 Timeline View

The timeline view (the top half of Figure 4.1) provides a visual representation of the execution of the application.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

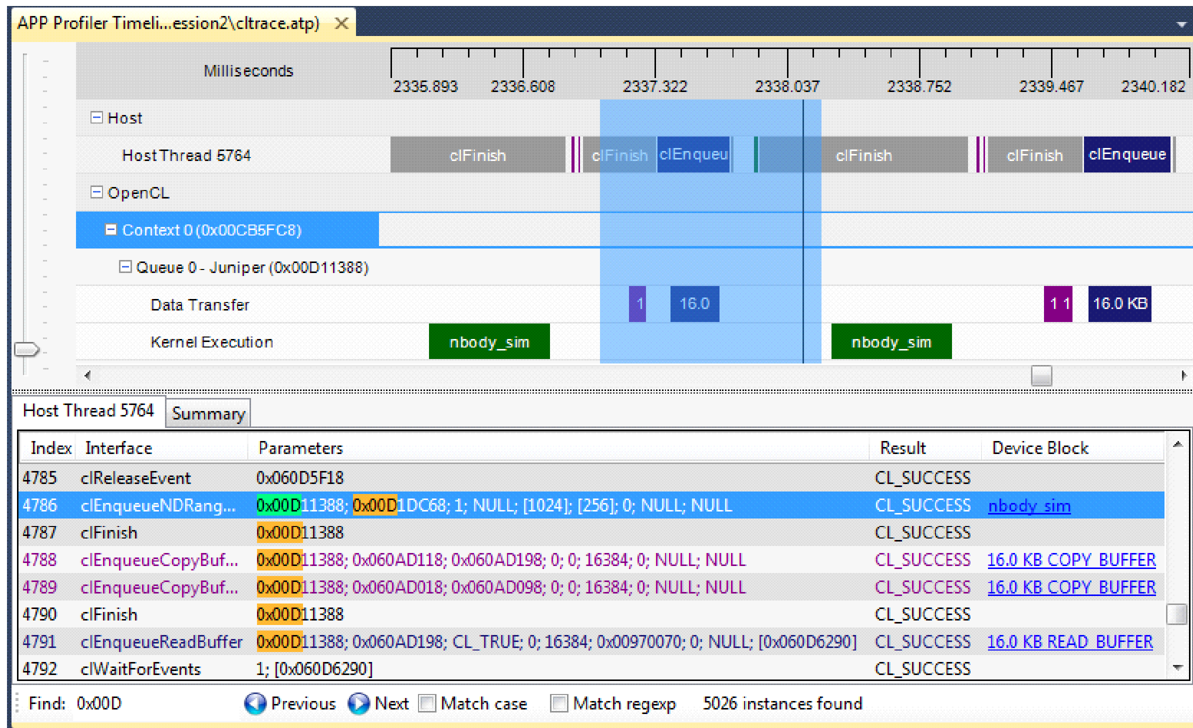

Figure 4.1 Timeline and API Trace View in Microsoft Visual Studio 2010

Along the top of the timeline is the time grid, which shows the total elapsed time of the application, in milliseconds. Timing begins when the first OpenCL call is made by the application, and ends when the final OpenCL is made. Directly below the time grid, each host (OS) thread that made at least one OpenCL call is listed. For each host thread, the OpenCL API calls are plotted along the time grid, showing the start time and duration of each call. Below the host threads, an OpenCL tree shows all contexts and queues created by the application, along with data transfer operations and kernel execution operations for each queue. The Timeline View can be navigated by zooming, panning, collapsing/expanding, and selecting an interest region. From the Timeline View, we can also navigate to the corresponding API call in the API Trace View and vice versa.

The Timeline View can be useful for debugging your OpenCL application. Some examples are:

- Easily confirm that the high-level structure of the algorithm is correct (the number of queues and contexts created match your expectation).
- Confirm that synchronization has been performed properly in the application. For example, if kernel A execution is dependent on a buffer write or copy and/or outputs from kernel B execution, then, if the synchronization has been set up correctly, kernel A execution appears after the completion of the buffer execution and kernel B execution in the timeline grid. It can be hard to find this type of synchronization error using traditional debugging techniques.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

- Confirm that the kernel and data transfer execution from all the queues have been performed efficiently. This is easily verified by ensuring that non-dependent kernel and data transfer execution happens concurrently in the timeline grid.

4.1.1.2 Summary Pages View

The Summary Pages View (Figure 4.2) shows the statistics of your OpenCL application. It can provide a general idea of the location of the program's bottlenecks. It also provides useful information such as the number of buffers and number of images created on each context, most expensive kernel call, etc.

Host Thread 1968 | Host Thread 2892 | Host Thread 3496 | Summary

Previous Next Context Summary

Context ID	# of Buffers	# of Images	# of Kernel Dispatch	Total Kernel Time(ms)	# of Memory Transfer	Total Memory Time(ms)	# of Read	Total Read Time (ms)	Size of Read	# of Write	Total Write Time(ms)	Size of Write	# of Map	Total Map Time (ms)	Size of Map	# of Copy	Total Copy Time (ms)	Size of Copy
0	2	0	2	73.17441	2	1.08503	0	0	0 Byte	2	1.08503	512.00 KB	0	0	0 Byte	0	0	0 Byte
1	2	0	1	35.53000	1	0.21951	0	0	0 Byte	1	0.21951	256.00 KB	0	0	0 Byte	0	0	0 Byte
2	2	0	1	3.41856	1	0.68449	0	0	0 Byte	1	0.68449	256.00 KB	0	0	0 Byte	0	0	0 Byte
3	2	0	1	35.73143	1	0.17307	0	0	0 Byte	1	0.17307	256.00 KB	0	0	0 Byte	0	0	0 Byte

Find: Previous Next Match case Match regexp

Figure 4.2 Context Summary Page View in Microsoft Visual Studio 2010

The Summary Pages View consists of the following pages.

1. API Summary shows the useful statistics for all OpenCL API calls made in the application for API hot spots identification.
2. Context Summary shows the timing information for all the kernel dispatches and data transfers for each context. This permits identifying whether the application is bound by the kernel execution or data transfer. If the application is bound by the data transfers, this page permits finding the most expensive data transfer type (read, write, copy, or map) in the application.
3. Kernel Summary lists all the kernels that are created in the application. If the application is bound by the kernel execution, it is possible to find the device causing the bottleneck. If the kernel execution on the GPU device is the bottleneck, use the GPU performance counters (see Section 4.1.2, “Collecting OpenCL GPU Kernel Performance Counters,” page 4-5) to investigate the bottleneck inside the kernel.
4. Top 10 Data Transfer Summary shows the top ten most expensive individual data transfers.
5. Top 10 Kernel Summary shows the top ten most expensive individual kernel executions.
6. Warning(s) and Error(s) shows potential problems in your OpenCL application.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

In order to minimize expensive data transfers, the algorithm/application may have be modified. With the help from the timeline view, we can investigate whether the data transfer execution has been most efficient (occurs concurrently with a kernel execution).

The Warning(s) and Error(s) page (Figure 4.3) shows potential problems and optimization hints in your OpenCL application, including unreleased OpenCL resources, OpenCL API failures, non-optimized work size, non-optimized data transfer, and excessive synchronization; it also provides suggestions to achieve better performance. Clicking on a hyperlink takes you to the corresponding OpenCL API that generated the message.

Index	Call Index	Thread ID	Type	Message
0	542	2268	Warning	Memory leak detected [Ref = 1, Handle = 0x0B1730B0]: Object created by clEnqueueNDRangeKernel
216	208	2268	Best Practices	clEnqueueNDRangeKernel: Work group size is too small - [1,1,1]. Recommended Work group size is a multiple of 64.
270	319	2268	Best Practices	clEnqueueNDRangeKernel: Global work size is too small - [111], resulting in low GPU utilization.
144	482	1932	Error	clEnqueueNDRangeKernel returns CL_INVALID_KERNEL_ARGS

Figure 4.3 Warning(s) and Error(s) Page

4.1.1.3 API Trace View

The API Trace View (the bottom half in Figure 4.1) lists all the OpenCL API calls made by the application. Each host thread that makes at least one OpenCL call is listed in a separate tab. Each tab contains a list of all the API calls made by that particular thread. For each call, it shows the index of the call (representing execution order), the name of the API function, a semicolon-delimited list of parameters passed to the function, and the value returned by the function. Double-clicking an item in the API Trace view displays and zooms into that API call in the Host Thread row in the Timeline View. If stack trace is enabled while collecting the API trace, and the application contains debug information, it is possible to navigate from the API trace to source code.

The lets you analyze and debug the input parameters and output results for each API call. For example it is easy to check that all the API calls are returning `CL_SUCCESS`, all the buffers are created with the correct flags, as well as to identify redundant API calls. The API Trace shows additional information about data transfers using `clEnqueueMapBuffer`/`clEnqueueMapImage`; this includes the source, destination, and transfer type of the map operation. Some devices can take advantage of zero copy to save on data transfer time.

4.1.2 Collecting OpenCL GPU Kernel Performance Counters

The GPU kernel performance counters can be used to find the possible bottlenecks in the kernel execution. You can find the list of performance counters supported by AMD Radeon™ GPUs in the Profiler documentation.

After determining the most expensive kernel to be optimized using the trace data, collect the GPU performance counters to drill down to the kernel execution on the GPU devices. Using the performance counters, it is possible to:

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

- Find the number of resources (VGPR, SGPR [if applicable], and Local Memory size) allocated for the kernel. These resources affect the possible number of in-flight wavefronts in the GPU (higher number is required to hide the data latency). The Occupancy Modeler identifies the limiting factor for achieving a higher count of in-flight wavefronts.
- Identify the number of ALU, global, and local memory instructions executed in the GPU.
- Identify the number of bytes fetched from and written to the global memory.
- View use of the SIMD engine and memory units in the system.
- View the efficiency of the Shader Compiler to pack ALU instructions into the VLIW instructions in AMD GPUs.
- View the local memory (LDS) bank conflict.

The Session view (Figure 4.4) shows the resulting performance counters for a Profiler session. The output data is recorded in a csv format.

APP Profiler Sessio...ssion1\Session1.csv)

Launch CSV

View Options: Show Kernel Dispatch | Show Data Transfer | Show Zero Column

Method	Execution(	GlobalWorkSize	GroupWorkSize	Time	LocalMemSiz	DataTransfer	GPRs
WriteBuffer	1			2.58932		16	
WriteBuffer	2			0.08525		16	
nbody_sim__k1_Juniper1	3	{ 1024 1 1}	{ 256 1 1}	0.65119	4096		12
CopyBuffer	4			0.18184		16	
CopyBuffer	5			0.10060		16	
ReadBuffer	6			0.13456		16	

Figure 4.4 Sample Session View in Microsoft Visual Studio 2010

4.1.3 OpenCL Kernel Occupancy Modeler

Figure 4.5 shows a sample screen shot of the OpenCL Kernel Occupancy Modeler.

The top part of the page shows four graphs (only three on non-GCN devices) that provide a visual indication of how kernel resources affect the theoretical number of in-flight wavefronts on a compute unit. The graph representing the limiting resource has its title displayed in red text. If there is more than one limiting resource, more than one graph can have a red title. In each graph, the actual usage of the particular resource being graphed is highlighted with an orange square. Hovering the mouse over a point in the graph causes a popup hint to be displayed; this shows the current X and Y values at that location.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Variable	Value	Device Limit
Device Info		
Device name	Tahiti	
Number of compute units	32	
Max number of waves per compute unit	40	
Max number of work-groups per compute unit	40	
Wavefront size	64	
Kernel Info		
Kernel name	mmmKernel_local	
Vector GPR usage per work-item	79	256
Scalar GPR usage per work-item	32	512
LDS usage per work-group	4096	32768
Flattened work-group size	64	256
Flattened global work size	65536	16777216
Number of waves per work-group	1	4
Kernel Occupancy		
Number of waves limited by Vector GPR and Work-group size	12	40
Number of waves limited by Scalar GPR and Work-group size	40	40
Number of waves limited by LDS and Work-group size	8	40
Number of waves limited by Work-group size	40	40
Limiting factor(s)	LDS	
Estimated occupancy	20%	

Figure 4.5 Sample Kernel Occupancy Modeler Screen

The first graph, titled **Number of waves limited by Work-group size**, shows how the number of active wavefronts is affected by the size of the work-group for the dispatched kernel. In the above screen shot, note that the highest number of wavefronts is achieved when the work-group size is in the range of 64 to 128.

The second graph, titled **Number of waves limited by VGPRs**, shows how the number of active wavefronts is affected by the number of vector GPRs used by the dispatched kernel. In the above screen shot, note that as the number of VGPRs used increases, the number active wavefronts decreases in steps. Note that this graph shows that more than 62 GPRs can be allocated, although 62 is the maximum number, since the Shader Compiler assumes the work-group size is 256 items by default (the largest possible work-group size). For the Shader Compiler to allocate more than 62 GPRs, the kernel source code must be marked with the `reqd_work_group_size` kernel attribute. This attribute can tell the Shader Compiler that the kernel is to be launched with a work-group size smaller than the maximum, allowing it to allocate more GPRs. Thus, for X-axis values greater than 62, the GPR graph shows the theoretical number of wavefronts that can be launched if the kernel specified a smaller work-group size using that attribute.

If running on and AMD Radeon™ HD 7XXX series (GCN) device, the third graph, titled **Number of waves limited by SGPR**, shows how the number of active wavefronts is affected by the number of scalar GPRs used by the dispatched kernel. In the above screen shot, note that as the number of used SGPRs increases, the number active wavefronts decreases in steps.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

The fourth graph, titled **Number of waves limited by LDS**, shows how the number of active wavefronts is affected by the amount of LDS used by the dispatched kernel. In the above screen shot, note that as the amount of LDS usage increases, the number active wavefronts decreases in steps.

Below the graphs is a table that provides information about the device, the kernel, and the kernel occupancy. In the **Kernel Occupancy** section, note the limits imposed by each kernel resource, as well as which resource is currently limiting the number of waves for the kernel dispatch. This also displays the kernel occupancy percentage.

4.2 AMD APP KernelAnalyzer

The AMD APP KernelAnalyzer is a static analysis tool to compile, analyze, and disassemble an OpenCL kernel for AMD Radeon GPUs. It is commonly used for rapid prototyping of OpenCL kernels since it quickly compiles the kernel for multiple GPU device targets.

It can be used as a GUI tool for interactive tunings of an OpenCL kernel or in command-line mode to generate detailed reports.

The KernelAnalyzer can be installed as part of the AMD APP SDK installation, or individually using its own installer package. The KernelAnalyzer package can be downloaded from:
http://developer.amd.com/tools/AMDAPPKernelAnalyzer/Pages/default.aspx.

To use the KernelAnalyzer, the AMD OpenCL run-time is required to be installed in the system; however, no GPU is required in the system.

To compile an OpenCL kernel in the KernelAnalyzer, drop the OpenCL source kernel to the source code panel in the GUI (Figure 4.6). The entire OpenCL application is not required to compile or analyze the OpenCL kernel.

With the KernelAnalyzer, it is possible to:

- Compile and disassemble the OpenCL kernel for multiple Catalyst driver versions and GPU device targets.
- View the OpenCL kernel compilation's error messages from the OpenCL run-time.
- View the AMD Intermediate Language (IL) code generated by the OpenCL run-time.
- View the ISA code generated by the AMD Shader Compiler. Typically, hard core kernel optimizations are performed by analyzing the ISA code.
- View the statistics generated by analyzing the ISA code.
- View General-Purpose Register usage and spill registers allocated for the kernel.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

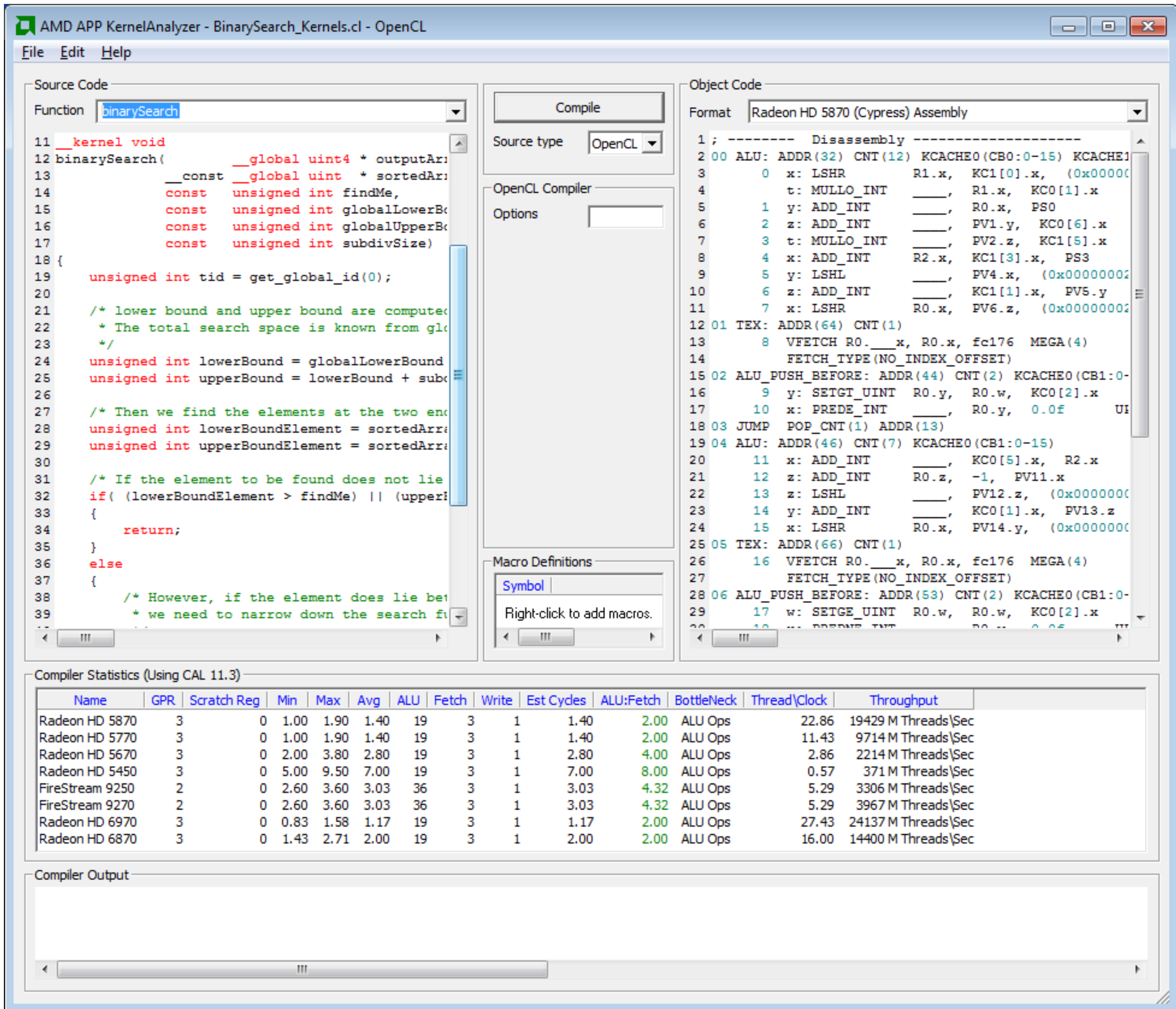

Figure 4.6 AMD APP Kernel Analyzer

4.3 Analyzing Processor Kernels

4.3.1 Intermediate Language and GPU Disassembly

The AMD Accelerated Parallel Processing software exposes the Intermediate Language (IL) and instruction set architecture (ISA) code generated for OpenCL™ kernels through the compiler options `-save-temps[=prefix]`.

The AMD Intermediate Language (IL) is an abstract representation for hardware vertex, pixel, and geometry shaders, as well as compute kernels that can be taken as input by other modules implementing the IL. An IL compiler uses an IL shader or kernel in conjunction with driver state information to translate these shaders into hardware instructions or a software emulation layer. For a complete description of IL, see the *AMD Intermediate Language (IL) Specification v2*.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

The instruction set architecture (ISA) defines the instructions and formats accessible to programmers and compilers for the AMD GPUs. The Northern Islands-family ISA instructions and microcode are documented in the *AMD Northern Islands-Family ISA Instructions and Microcode*.

4.3.2 Generating IL and ISA Code

In Microsoft Visual Studio, the AMD APP Profiler provides an integrated tool to view IL and ISA code. (The AMD APP KernelAnalyzer also can show the IL and ISA code.) After running the Profiler, single-click the name of the kernel for detailed programming and disassembly information. The associated ISA disassembly is shown in a new tab. A drop-down menu provides the option to view the IL, ISA, or source OpenCL for the selected kernel.

Developers also can generate IL and ISA code from their OpenCL kernel by setting the environment variable `AMD_OCL_BUILD_OPTIONS_APPEND=-save-temps` (see Section 2.1.4, “AMD-Developed Supplemental Compiler Options,” page 2-4).

4.4 Estimating Performance

4.4.1 Measuring Execution Time

The OpenCL runtime provides a built-in mechanism for timing the execution of kernels by setting the `CL_QUEUE_PROFILING_ENABLE` flag when the queue is created. Once profiling is enabled, the OpenCL runtime automatically records timestamp information for every kernel and memory operation submitted to the queue.

OpenCL provides four timestamps:

- `CL_PROFILING_COMMAND_QUEUED` - Indicates when the command is enqueued into a command-queue on the host. This is set by the OpenCL runtime when the user calls an `clEnqueue*` function.
- `CL_PROFILING_COMMAND_SUBMIT` - Indicates when the command is submitted to the device. For AMD GPU devices, this time is only approximately defined and is not detailed in this section.
- `CL_PROFILING_COMMAND_START` - Indicates when the command starts execution on the requested device.
- `CL_PROFILING_COMMAND_END` - Indicates when the command finishes execution on the requested device.

The sample code below shows how to compute the kernel execution time (End-Start):

```
cl_event myEvent;
cl_ulong startTime, endTime;

clCreateCommandQueue (…, CL_QUEUE_PROFILING_ENABLE, NULL);
clEnqueueNDRangeKernel(…, &myEvent);
clFinish(myCommandQ); // wait for all events to finish
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

```
clGetEventProfilingInfo(myEvent, CL_PROFILING_COMMAND_START,
    sizeof(cl_ulong), &startTime, NULL);
clGetEventProfilingInfo(myEvent,  CL_PROFILING_COMMAND_END,
    sizeof(cl_ulong), &endTimeNs, NULL);
cl_ulong kernelExecTimeNs = endTime-startTime;
```

The AMD APP Profiler also can record the execution time for a kernel automatically. The Kernel Time metric reported in the Profiler output uses the built-in OpenCL timing capability and reports the same result as the `kernelExecTimeNs` calculation shown above.

Another interesting metric to track is the kernel launch time (Start – Queue). The kernel launch time includes both the time spent in the user application (after enqueuing the command, but before it is submitted to the device), as well as the time spent in the runtime to launch the kernel. For CPU devices, the kernel launch time is fast (tens of µs), but for discrete GPU devices it can be several hundred µs. Enabling profiling on a command queue adds approximately 10 µs to 40 µs overhead to all clEnqueue calls. Much of the profiling overhead affects the start time; thus, it is visible in the launch time. Be careful when interpreting this metric. To reduce the launch overhead, the AMD OpenCL runtime combines several command submissions into a batch. Commands submitted as batch report similar start times and the same end time.

4.4.2 Using the OpenCL timer with Other System Timers

The resolution of the timer, given in ns, can be obtained from:

```
clGetDeviceInfo(…,CL_DEVICE_PROFILING_TIMER_RESOLUTION…);
```

AMD CPUs and GPUs report a timer resolution of 1 ns. AMD OpenCL devices are required to correctly track time across changes in frequency and power states. Also, the AMD OpenCL SDK uses the same time-domain for all devices in the platform; thus, the profiling timestamps can be directly compared across the CPU and GPU devices.

The sample code below can be used to read the current value of the OpenCL timer clock. The clock is the same routine used by the AMD OpenCL runtime to generate the profiling timestamps. This function is useful for correlating other program events with the OpenCL profiling timestamps.

```
uint64_t
timeNanos()
{
#ifdef linux
  struct timespec tp;
  clock_gettime(CLOCK_MONOTONIC, &tp);
  return (unsigned long long) tp.tv_sec * (1000ULL * 1000ULL * 1000ULL) +
      (unsigned long long) tp.tv_nsec;
#else
  LARGE_INTEGER current;
  QueryPerformanceCounter(&current);
  return (unsigned long long)((double)current.QuadPart / m_ticksPerSec * 1e9);
#endif
}
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Normal CPU time-of-day routines can provide a rough measure of the elapsed time of a GPU kernel. GPU kernel execution is non-blocking, that is, calls to `enqueue*Kernel` return to the CPU before the work on the GPU is finished. For an accurate time value, ensure that the GPU is finished. In OpenCL, you can force the CPU to wait for the GPU to become idle by inserting calls to `clFinish()` before and after the sequence you want to time; this increases the timing accuracy of the CPU routines. The routine `clFinish()` blocks the CPU until all previously enqueued OpenCL commands have finished.

For more information, see section 5.9, “Profiling Operations on Memory Objects and Kernels,” of the *OpenCL 1.0 Specification*.

4.4.3 Estimating Memory Bandwidth

The memory bandwidth required by a kernel is perhaps the most important performance consideration. To calculate this:

Effective Bandwidth = $(B_r + B_w)/T$

where:
B_r = total number of bytes read from global memory.
B_w = total number of bytes written to global memory.
T = time required to run kernel, specified in nanoseconds.

If B_r and B_w are specified in bytes, and T in ns, the resulting effective bandwidth is measured in GB/s, which is appropriate for current CPUs and GPUs for which the peak bandwidth range is 20-260 GB/s. Computing B_r and B_w requires a thorough understanding of the kernel algorithm; it also can be a highly effective way to optimize performance. For illustration purposes, consider a simple matrix addition: each element in the two source arrays is read once, added together, then stored to a third array. The effective bandwidth for a 1024x1024 matrix addition is calculated as:

B_r = 2 x (1024 x 1024 x 4 bytes) **=** 8388608 bytes ;; 2 arrays, 1024x1024, each element 4-byte float

B_w = 1 x (1024 x 1024 x 4 bytes) = 4194304 bytes ;; 1 array, 1024x1024, each element 4-byte float.

If the elapsed time for this copy as reported by the profiling timers is 1000000 ns (1 million ns, or .001 sec), the effective bandwidth is:

$(B_r+B_w)/T$ = (8388608+4194304)/1000000 = 12.6GB/s

The AMD APP Profiler can report the number of dynamic instructions per thread that access global memory through the FetchInsts and WriteInsts counters. The Fetch and Write reports average the per-thread counts; these can be fractions if the threads diverge. The Profiler also reports the dimensions of the global NDRange for the kernel in the `GlobalWorkSize` field. The total number of threads can be determined by multiplying together the three components of the range. If all (or most) global accesses are the same size, the counts from the Profiler and the approximate size can be used to estimate B_r and B_w:

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

B_r = Fetch * GlobalWorkitems * Size

B_w = Write * GlobalWorkitems * Element Size

where GlobalWorkitems is the dispatch size.

An example Profiler output and bandwidth calculation:

Method	GlobalWorkSize	Time	Fetch	Write
runKernel_Cypress	{192; 144; 1}	0.9522	70.8	0.5

WaveFrontSize = 192*144*1 = 27648 global work items.

In this example, assume we know that all accesses in the kernel are four bytes; then, the bandwidth can be calculated as:

Br = 70.8 * 27648 * 4 = 7829914 bytes

Bw = 0.5 * 27648 * 4 = 55296 bytes

The bandwidth then can be calculated as:

(Br + Bw)/T = (7829914 bytes + 55296 bytes) / .9522 ms / 1000000
= 8.2 GB/s

4.5 OpenCL Memory Objects

This section explains the AMD OpenCL runtime policy for memory objects. It also recommends best practices for best performance.

OpenCL uses memory objects to pass data to kernels. These can be either buffers or images. Space for these is managed by the runtime, which uses several types of memory, each with different performance characteristics. Each type of memory is suitable for a different usage pattern. The following subsections describe:

- the memory types used by the runtime;
- how to control which memory kind is used for a memory object;
- how the runtime maps memory objects for host access;
- how the runtime performs memory object reading, writing and copying;
- how best to use command queues; and
- some recommended usage patterns.

4.5.1 Types of Memory Used by the Runtime

Memory is used to store memory objects that are accessed by kernels executing on the device, as well as to hold memory object data when they are mapped for access by the host application. This section describes the different memory kinds used by the runtime. Table 4.1 lists the performance of each memory type given a PCIe3-capable platform and a high-end AMD Radeon™ 7XXX discrete GPU. In

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Table 4.1, when host memory is accessed by the GPU shader, it is of type CL_MEM_ALLOC_HOST_PTR. When GPU memory is accessed by the CPU, it is of type CL_MEM_PERSISTENT_MEM_AMD.

Table 4.1 Memory Bandwidth in GB/s (R = read, W = write) in GB/s

	CPU R	GPU R	GPU Shader R	GPU Shader W	GPU DMA R	GPU DMA W
Host Memory	10 - 20	10 - 20	9 - 10	2.5	11 - 12	11 - 12
GPU Memory	.01	9 - 10	230	120 -150	n/a	n/a

Host memory and device memory in the above table consists of one of the subtypes given below.

4.5.1.1 Host Memory

This regular CPU memory can be access by the CPU at full memory bandwidth; however, it is not directly accessible by the GPU. For the GPU to transfer host memory to device memory (for example, as a parameter to clEnqueueReadBuffer or clEnqueueWriteBuffer), it first must be pinned (see section 4.5.1.2). Pinning takes time, so avoid incurring pinning costs where CPU overhead must be avoided.

When host memory is copied to device memory, the OpenCL runtime uses the following transfer methods.

- <=32 kB: For transfers from the host to device, the data is copied by the CPU to a runtime pinned host memory buffer, and the DMA engine transfers the data to device memory. The opposite is done for transfers from the device to the host.
- >32 kB and <=16 MB: The host memory physical pages containing the data are pinned, the GPU DMA engine is used, and the pages then are unpinned.
- >16 MB: Runtime pinned host memory staging buffers are used. The CPU copies the data in pieces, which then are transferred to the device using the GPU DMA engine. Double buffering is used to overlap the CPU copies with the DMA.

Due to the cost of copying to staging buffers, or pinning/unpinning host memory, host memory does not offer the best transfer performance.

4.5.1.2 Pinned Host Memory

This is host memory that the operating system has bound to a fixed physical address and that the operating system ensures is resident. The CPU can access pinned host memory at full memory bandwidth. The runtime limits the total amount of pinned host memory that can be used for memory objects. (See Section 4.5.2, "Placement," page 4-16, for information about pinning memory.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

If the runtime knows the data is in pinned host memory, it can be transferred to, and from, device memory without requiring staging buffers or having to perform pinning/unpinning on each transfer. This offers improved transfer performance.

Currently, the runtime recognizes only data that is in pinned host memory for operation arguments that are memory objects it has allocated in pinned host memory. For example, the `buffer` argument of `clEnqueueReadBuffer`/`clEnqueueWriteBuffer` and `image` argument of `clEnqueueReadImage`/`clEnqueueWriteImage`. It does not detect that the ptr arguments of these operations addresses pinned host memory, even if they are the result of `clEnqueueMapBuffer`/`clEnqueueMapImage` on a memory object that is in pinned host memory.

The runtime can make pinned host memory directly accessible from the GPU. Like regular host memory, the CPU uses caching when accessing pinned host memory. Thus, GPU accesses must use the CPU cache coherency protocol when accessing. For discrete devices, the GPU access to this memory is through the PCIe bus, which also limits bandwidth. For fusion devices that do not have the PCIe overhead, GPU access is significantly slower than accessing device-visible host memory (see section 4.5.1.3), which does not use the cache coherency protocol.

4.5.1.3 Device-Visible Host Memory

The runtime allocates a limited amount of pinned host memory that is accessible by the GPU without using the CPU cache coherency protocol. This allows the GPU to access the memory at a higher bandwidth than regular pinned host memory.

A portion of this memory is also configured to be accessible by the CPU as uncached memory. Thus, reads by the CPU are significantly slower than those from regular host memory. However, these pages are also configured to use the memory system write combining buffers. The size, alignment, and number of write combining buffers is chip-set dependent. Typically, there are 4 to 8 buffers of 64 bytes, each aligned to start on a 64-byte memory address. These allow writes to adjacent memory locations to be combined into a single memory access. This allows CPU streaming writes to perform reasonably well. Scattered writes that do not fill the write combining buffers before they have to be flushed do not perform as well.

Fusion devices have no device memory and use device-visible host memory for their global device memory.

4.5.1.4 Device Memory

Discrete GPU devices have their own dedicated memory, which provides the highest bandwidth for GPU access. The CPU cannot directly access device memory on a discrete GPU (except for the host-visible device memory portion described in section 4.5.1.5).

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

On an APU, the system memory is shared between the GPU and the CPU; it is visible by either the CPU or the GPU at any given time. A significant benefit of this is that buffers can be zero copied between the devices by using map/unmap operations to logically move the buffer between the CPU and the GPU address space. See Section 4.5.4, “Mapping,” page 4-18, for more information on zero copy.

4.5.1.5 Host-Visible Device Memory

A limited portion of discrete GPU device memory is configured to be directly accessible by the CPU. It can be accessed by the GPU at full bandwidth, but CPU access is over the PCIe bus; thus, it is much slower that host memory bandwidth. The memory is mapped into the CPU address space as uncached, but using the memory system write combining buffers. This results in slow CPU reads and scattered writes, but streaming CPU writes perform much better because they reduce PCIe overhead.

4.5.2 Placement

Every OpenCL memory object has a location that is defined by the flags passed to `clCreateBuffer`/`clCreateImage`. A memory object can be located either on a device, or (as of SDK 2.4) it can be located on the host and accessed directly by all the devices. The Location column of Table 4.2 gives the memory type used for each of the allocation flag values for different kinds of devices. When a device kernel is executed, it accesses the contents of memory objects from this location. The performance of these accesses is determined by the memory kind used.

An OpenCL context can have multiple devices, and a memory object that is located on a device has a location on each device. To avoid over-allocating device memory for memory objects that are never used on that device, space is not allocated until first used on a device-by-device basis. For this reason, the first use of a memory object after it is created can be slower than subsequent uses.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Table 4.2 OpenCL Memory Object Properties

clCreateBuffer/ clCreateImage Flags Argument	Device Type	Location	clEnqueueMapBuffer/ clEnqueueMapImage/ clEnqueueUnmapMemObject	
			Map Mode	**Map Location**
Default (none of the following flags)	Discrete GPU	Device memory	Copy	Mapped data size: • <=32MiB: Pinned host memory • >32MiB: Host memory (different memory area can be used on each map)
	Fusion APU	Device-visible host memory		
	CPU	Use *Map Location* directly	Zero copy	
`CL_MEM_ALLOC_HOST_PTR` (`clCreateBuffer` on Windows 7 and Vista for Evergreen, Northern Islands, and Southern Islands; Southern Islands devices have this functionality on Linux as well.)	Discrete GPU	Pinned host memory shared by all devices in context (unless only device in context is CPU; then, host memory)	Zero copy	Use Location directly (same memory area is used on each map)
	Fusion APU			
	CPU			
`CL_MEM_ALLOC_HOST_PTR` (`clCreateImage` on Windows 7, Vista and Linux; `clCreateBuffer` on Linux)	Discrete GPU	Device memory	Copy	Pinned host memory, unless only device in context is CPU; then, host memory (same memory area is used on each map)
	Fusion APU	Device-visible memory		
	CPU		Zero copy	
`CL_MEM_USE_HOST_PTR`	Discrete GPU	Device memory	Copy	Pinned host memory, unless only device in context is CPU; then, host memory (same memory area is used on each map)
	Fusion APU	Device-visible host memory		
	CPU	Use *Map Location* directly	Zero copy	
`CL_MEM_USE_PERSISTENT_MEM_AMD` on Windows 7 and Vista for Evergreen, Northern Islands, and Southern Islands; Southern Islands devices have this functionality on Linux as well.)	Discrete GPU	Host-visible device memory	Zero copy	Use *Location* directly (different memory area can be used on each map)
	Fusion APU	Device-visible host memory		
	CPU	Host memory		
`CL_MEM_USE_PERSISTENT_MEM_AMD` (Linux for Evergreen and Northern Islands)	Same as default.			

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

4.5.3 Memory Allocation

4.5.3.1 Using the CPU

Create memory objects with `CL_MEM_ALLOC_HOST_PTR`, and use `map`/`unmap`; do not use `read`/`write`. The reason for this is that if the object is created with `CL_MEM_USE_HOST_PTR` the CPU is running the kernel on the buffer provided by the application (a hack that all vendors use). This results in zero copy between the CPU and the application buffer; the kernel updates the application buffer, and in this case a `map`/`unmap` is actually a no-op. Also, when allocating the buffer on the host, ensure that it is created with the correct alignment. For example, a buffer to be used as float4* must be 128-bit aligned.

4.5.3.2 Using Both CPU and GPU Devices, or using an APU Device

When creating memory objects, create them with `CL_MEM_USE_PERSISTENT_MEM_AMD`. This enables the zero copy feature, as explained in Section 4.5.3.1, "Using the CPU.".

4.5.3.3 Buffers vs Images

Unlike GPUs, CPUs do not contain dedicated hardware (samplers) for accessing images. Instead, image access is emulated in software. Thus, a developer may prefer using buffers instead of images if no sampling operation is needed.

4.5.3.4 Choosing Execution Dimensions

Note the following guidelines.

- Make the number of work-groups a multiple of the number of logical CPU cores (device compute units) for maximum use.
- When work-groups number exceed the number of CPU cores, the CPU cores execute the work-groups sequentially.

4.5.4 Mapping

The host application can use `clEnqueueMapBuffer`/`clEnqueueMapImage` to obtain a pointer that can be used to access the memory object data. When finished accessing, `clEnqueueUnmapMemObject` must be used to make the data available to device kernel access. When a memory object is located on a device, the data either can be transferred to, and from, the host, or (as of SDK 2.4) be accessed directly from the host. Memory objects that are located on the host, or located on the device but accessed directly by the host, are termed zero copy memory objects. The data is never transferred, but is accessed directly by both the host and device. Memory objects that are located on the device and transferred to, and from, the device when mapped and unmapped are termed copy memory objects. The Map Mode column of Table 4.2 specifies the transfer mode used for each kind of memory object, and the Map Location column indicates the kind of memory referenced by the pointer returned by the map operations.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

4.5.4.1 Zero Copy Memory Objects

CL_MEM_USE_PERSISTENT_MEM_AMD, CL_MEM_USE_HOST_PTR, and CL_MEM_ALLOC_HOST_PTR support zero copy memory objects. The first provides device-resident zero copy memory objects; the other two provide host-resident zero copy memory objects.

Zero copy memory objects can be used by an application to optimize data movement. When clEnqueueMapBuffer / clEnqueueMapImage / clEnqueueUnmapMemObject are used, no runtime transfers are performed, and the operations are very fast; however, the runtime can return a different pointer value each time a zero copy memory object is mapped. Note that only images created with CL_MEM_USE_PERSISTENT_MEM_AMD can be zero copy.

Southern Island devices support zero copy memory objects under Linux; however, only images created with CL_MEM_USE_PERSISTENT_MEM_AMD can be zero copy.

Zero copy host resident memory objects can boost performance when host memory is accessed by the device in a sparse manner or when a large host memory buffer is shared between multiple devices and the copies are too expensive. When choosing this, the cost of the transfer must be greater than the extra cost of the slower accesses.

Streaming writes by the host to zero copy device resident memory objects are about as fast as the transfer rates, so this can be a good choice when the host does not read the memory object to avoid the host having to make a copy of the data to transfer. Memory objects requiring partial updates between kernel executions can also benefit. If the contents of the memory object must be read by the host, use clEnqueueCopyBuffer to transfer the data to a separate CL_MEM_ALLOC_HOST_PTR buffer.

4.5.4.2 Copy Memory Objects

For memory objects with copy map mode, the memory object location is on the device, and it is transferred to, and from, the host when clEnqueueMapBuffer / clEnqueueMapImage / clEnqueueUnmapMemObject are called. Table 4.3 shows how the map_flags argument affects transfers. The runtime transfers only the portion of the memory object requested in the offset and cb arguments. When accessing only a portion of a memory object, only map that portion for improved performance.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Table 4.3 Transfer policy on clEnqueueMapBuffer / clEnqueueMapImage / clEnqueueUnmapMemObject for Copy Memory Objects

clEnqueueMapBuffer / clEnqueueMapImage map_flags argument	Transfer on clEnqueueMapBuffer / clEnqueueMapImage	Transfer on clEnqueueUnmapMemObject
CL_MAP_READ	Device to host, if map location is not current.	None.
CL_MAP_WRITE	Device to host, if map location is not current.	Host to device.
CL_MAP_READ CL_MAP_WRITE	Device to host if map location is not current.	Host to device.
CL_MAP_WRITE_INVALIDATE_REGION	None.	Host to device.

For default memory objects, the pointer returned by clEnqueueMapBuffer / clEnqueueMapImage may not be to the same memory area each time because different runtime buffers may be used.

For CL_MEM_USE_HOST_PTR and CL_MEM_ALLOC_HOST_PTR the same map location is used for all maps; thus, the pointer returned is always in the same memory area. For other copy memory objects, the pointer returned may not always be to the same memory region.

For CL_MEM_USE_HOST_PTR and the CL_MEM_ALLOC_HOST_PTR cases that use copy map mode, the runtime tracks if the map location contains an up-to-date copy of the memory object contents and avoids doing a transfer from the device when mapping as CL_MAP_READ. This determination is based on whether an operation such as clEnqueueWriteBuffer/clEnqueueCopyBuffer or a kernel execution has modified the memory object. If a memory object is created with CL_MEM_READ_ONLY, then a kernel execution with the memory object as an argument is not considered as modifying the memory object. Default memory objects cannot be tracked because the map location changes between map calls; thus, they are always transferred on the map.

For CL_MEM_USE_HOST_PTR, clCreateBuffer/clCreateImage pins the host memory passed to the host_ptr argument. It is unpinned when the memory object is deleted. To minimize pinning costs, align the memory to 4KiB. This avoids the runtime having to pin/unpin on every map/unmap transfer, but does add to the total amount of pinned memory.

For CL_MEM_USE_HOST_PTR, the host memory passed as the ptr argument of clCreateBuffer/clCreateImage is used as the map location. As mentioned in section 4.5.1.1, host memory transfers incur considerable cost in pinning/unpinning on every transfer. If used, minimize the pinning cost by ensuring the memory is 4 kB aligned. If host memory that is updated once is required, use CL_MEM_ALLOC_HOST_PTR with the CL_MEM_COPY_HOST_PTR flag instead. If device memory is needed, use CL_MEM_USE_PERSISTENT_MEM_AMD and clEnqueueWriteBuffer.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

If `CL_MEM_COPY_HOST_PTR` is specified with `CL_MEM_ALLOC_HOST_PTR` when creating a memory object, the memory is allocated in pinned host memory and initialized with the passed data. For other kinds of memory objects, the deferred allocation means the memory is not yet allocated on a device, so the runtime has to copy the data into a temporary runtime buffer. The memory is allocated on the device when the device first accesses the resource. At that time, any data that must be transferred to the resource is copied. For example, this would apply when a buffer was allocated with the flag `CL_MEM_COPY_HOST_PTR`. Using `CL_MEM_COPY_HOST_PTR` for these buffers is not recommended because of the extra copy. Instead, create the buffer without `CL_MEM_COPY_HOST_PTR`, and initialize with `clEnqueueWriteBuffer`/`clEnqueueWriteImage`.

When images are transferred, additional costs are involved because the image must be converted to, and from, linear address mode for host access. The runtime does this by executing kernels on the device.

4.5.5 Reading, Writing, and Copying

There are numerous OpenCL commands to read, write, and copy buffers and images. The runtime performs transfers depending on the memory kind of the source and destination. When transferring between host memory and device memory the methods described in section Section 4.5.1.1, “Host Memory,” page 4-14, are used. `Memcpy` is used to transferring between the various kinds of host memory, this may be slow if reading from device visible host memory, as described in section Section 4.5.1.3, “Device-Visible Host Memory,” page 4-15. Finally, device kernels are used to copy between device memory. For images, device kernels are used to convert to and from the linear address mode when necessary.

4.5.6 Command Queue

It is best to use non-blocking commands to allow multiple commands to be queued before the command queue is flushed to the GPU. This sends larger batches of commands, which amortizes the cost of preparing and submitting work to the GPU. Use event tracking to specify the dependence between operations. It is recommended to queue operations that do not depend of the results of previous copy and map operations. This can help keep the GPU busy with kernel execution and DMA transfers. Note that if a non-blocking copy or map is queued, it does not start until the command queue is flushed. Use `clFlush` if necessary, but avoid unnecessary flushes because they cause small command batching.

4.6 OpenCL Data Transfer Optimization

The AMD OpenCL implementation offers several optimized paths for data transfer to, and from, the device. The following chapters describe buffer and image paths, as well as how they map to common application scenarios. To find out where the application’s buffers are stored (and understand how the data

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

transfer behaves), use the APP Profiler's API Trace View, and look at the tool tips of the `clEnqueueMapBuffer` calls.

4.6.1 Definitions

- *Deferred allocation* — The CL runtime attempts to minimize resource consumption by delaying buffer allocation until first use. As a side effect, the first accesses to a buffer may be more expensive than subsequent accesses.
- *Peak interconnect bandwidth* — As used in the text below, this is the transfer bandwidth between host and device that is available under optimal conditions at the application level. It is dependent on the type of interconnect, the chipset, and the graphics chip. As an example, a high-performance PC with a PCIe 3.0 16x bus and a GCN architecture (AMD Radeon™ HD 7XXX series) graphics card has a nominal interconnect bandwidth of 16 GB/s.
- *Pinning* — When a range of host memory is prepared for transfer to the GPU, its pages are locked into system memory. This operation is called pinning; it can impose a high cost, proportional to the size of the memory range. One of the goals of optimizing data transfer is to use pre-pinned buffers whenever possible. However, if pre-pinned buffers are used excessively, it can reduce the available system memory and result in excessive swapping. Host side zero copy buffers provide easy access to pre-pinned memory.
- *WC* — Write Combine is a feature of the CPU write path to a select region of the address space. Multiple adjacent writes are combined into cache lines (for example, 64 bytes) before being sent to the external bus. This path typically provides fast streamed writes, but slower scattered writes. Depending on the chip set, scattered writes across a graphics interconnect can be very slow. Also, some platforms require multi-core CPU writes to saturate the WC path over an interconnect.
- *Uncached accesses* — Host memory and I/O regions can be configured as uncached. CPU read accesses are typically very slow; for example: uncached CPU reads of graphics memory over an interconnect.
- *USWC* — Host memory from the Uncached Speculative Write Combine heap can be accessed by the GPU without causing CPU cache coherency traffic. Due to the uncached WC access path, CPU streamed writes are fast, while CPU reads are very slow. On Fusion devices, this memory provides the fastest possible route for CPU writes followed by GPU reads.

4.6.2 Buffers

OpenCL buffers currently offer the widest variety of specialized buffer types and optimized paths, as well as slightly higher transfer performance.

4.6.2.1 Regular Device Buffers

Buffers allocated using the flags `CL_MEM_READ_ONLY`, `CL_MEM_WRITE_ONLY`, or `CL_MEM_READ_WRITE` are placed on the GPU device. These buffers can be accessed by a GPU kernel at very high bandwidths. For example, on a high-end

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

graphics card, the OpenCL kernel read/write performance is significantly higher than 100 GB/s. When device buffers are accessed by the host through any of the OpenCL read/write/copy and map/unmap API calls, the result is an explicit transfer across the hardware interconnect.

4.6.2.2 Zero Copy Buffers

AMD APP SDK 2.4 on Windows 7 and Vista introduces a new feature called *zero copy buffers*.

If a buffer is of the zero copy type, the runtime tries to leave its content in place, unless the application explicitly triggers a transfer (for example, through `clEnqueueCopyBuffer()`). Depending on its type, a zero copy buffer resides on the host or the device. Independent of its location, it can be accessed directly by the host CPU or a GPU device kernel, at a bandwidth determined by the capabilities of the hardware interconnect.

Calling `clEnqueueMapBuffer()` and `clEnqueueUnmapMemObject()` on a zero copy buffer is typically a low-cost operation.

Since not all possible read and write paths perform equally, check the application scenarios below for recommended usage. To assess performance on a given platform, use the BufferBandwidth sample.

If a given platform supports the zero copy feature, the following buffer types are available:

- The `CL_MEM_ALLOC_HOST_PTR` and `CL_MEM_USE_HOST_PTR` buffers are:
  - zero copy buffers that resides on the host.
  - directly accessible by the host at host memory bandwidth.
  - directly accessible by the device across the interconnect.
  - a pre-pinned sources or destinations for CL read, write, and copy commands into device memory at peak interconnect bandwidth.

  Note that buffers created with the flag `CL_MEM_ALLOC_HOST_PTR` together with `CL_MEM_READ_ONLY` may reside in uncached write-combined memory. As a result, CPU can have high streamed write bandwidth, but low read and potentially low write scatter bandwidth, due to the uncached WC path.
- The `CL_MEM_USE_PERSISTENT_MEM_AMD` buffer is
  - a zero copy buffer that resides on the GPU device.
  - directly accessible by the GPU device at GPU memory bandwidth.
  - directly accessible by the host across the interconnect (typically with high streamed write bandwidth, but low read and potentially low write scatter bandwidth, due to the uncached WC path).
  - copyable to, and from, the device at peak interconnect bandwidth using CL read, write, and copy commands.

  There is a limit on the maximum size per buffer, as well as on the total size of all buffers. This is platform-dependent, limited in size for each buffer, and

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

also for the total size of all buffers of that type (a good working assumption is 64 MB for the per-buffer limit, and 128 MB for the total).

Zero copy buffers work well on Fusion APU devices. SDK 2.5 introduced an optimization that is of particular benefit on Fusion APUs. The runtime uses USWC memory for buffers allocated as `CL_MEM_ALLOC_HOST_PTR | CL_MEM_READ_ONLY`. On Fusion systems, this type of zero copy buffer can be written to by the CPU at very high data rates, then handed over to the GPU at minimal cost for equally high GPU read-data rates over the Radeon memory bus. This path provides the highest data transfer rate for the CPU-to-GPU path. The use of multiple CPU cores may be necessary to achieve peak write performance.

1. buffer = `clCreateBuffer(CL_MEM_ALLOC_HOST_PTR | CL_MEM_READ_ONLY)`
2. address = `clMapBuffer`(buffer)
3. `memset`(address) or `memcpy`(address) (if possible, using multiple CPU cores)
4. `clEnqueueUnmapMemObject`(buffer)
5. `clEnqueueNDRangeKernel`(buffer)

As this memory is not cacheable, CPU read operations are very slow. This type of buffer also exists on discrete platforms, but transfer performance typically is limited by PCIe bandwidth.

Zero copy buffers can provide low latency for small transfers, depending on the transfer path. For small buffers, the combined latency of map/CPU memory access/unmap can be smaller than the corresponding DMA latency.

4.6.2.3 Pre-pinned Buffers

AMD APP SDK 2.5 introduces a new feature called pre-pinned buffers. This feature is supported on Windows 7, Windows Vista, and Linux.

Buffers of type `CL_MEM_ALLOC_HOST_PTR` or `CL_MEM_USE_HOST_PTR` are pinned at creation time. These buffers can be used directly as a source or destination for `clEnqueueCopyBuffer` to achieve peak interconnect bandwidth. Mapped buffers also can be used as a source or destination for `clEnqueueRead/WriteBuffer` calls, again achieving peak interconnect bandwidth. Note that using `CL_MEM_USE_HOST_PTR` permits turning an existing user memory region into pre-pinned memory. However, in order to stay on the fast path, that memory must be aligned to 256 bytes. Buffers of type `CL_MEM_USE_HOST_PTR` remain pre-pinned as long as they are used only for data transfer, but not as kernel arguments. If the buffer is used in a kernel, the runtime creates a cached copy on the device, and subsequent copies are not on the fast path. The same restriction applies to `CL_MEM_ALLOC_HOST_PTR` allocations under Linux.

See usage examples described for various options below.

The pre-pinned path is supported for the following calls.

- `clEnqueueRead/WriteBuffer`

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

- clEnqueueRead/WriteImage
- clEnqueueRead/WriteBufferRect (Windows only)

Offsets into mapped buffer addresses are supported, too.

Note that the CL image calls must use pre-pinned mapped buffers on the host side, and not pre-pinned images.

4.6.2.4 Application Scenarios and Recommended OpenCL Paths

The following section describes various application scenarios, and the corresponding paths in the OpenCL API that are known to work well on AMD platforms. The various cases are listed, ordered from generic to more specialized.

From an application point of view, two fundamental use cases exist, and they can be linked to the various options, described below.

- An application wants to transfer a buffer that was already allocated through `malloc()` or `mmap()`. In this case, options 2), 3) and 4) below always consist of a `memcpy()` plus a device transfer. Option 1) does not require a `memcpy()`.
- If an application is able to let OpenCL allocate the buffer, options 2) and 4) below can be used to avoid the extra `memcpy()`. In the case of option 5), `memcpy()` and transfer are identical.

Note that the OpenCL runtime uses deferred allocation to maximize memory resources. This means that a complete roundtrip chain, including data transfer and kernel compute, might take one or two iterations to reach peak performance.

A code sample named BufferBandwidth can be used to investigate and benchmark the various transfer options in combination with different buffer types.

Option 1 - `clEnqueueWriteBuffer()` and `clEnqueueReadBuffer()`

This option is the easiest to use on the application side. `CL_MEM_USE_HOST_PTR` is an ideal choice if the application wants to transfer a buffer that has already been allocated through `malloc()` or `mmap()`.

There are two ways to use this option. The first uses `clEnqueueRead/WriteBuffer` on a pre-pinned, mapped host-side buffer:

a. pinnedBuffer = `clCreateBuffer( CL_MEM_ALLOC_HOST_PTR or CL_MEM_USE_HOST_PTR )`
b. deviceBuffer = `clCreateBuffer()`
c. void *pinnedMemory = `clEnqueueMapBuffer( pinnedBuffer )`
d. `clEnqueueRead/WriteBuffer`(deviceBuffer, pinnedMemory)
e. `clEnqueueUnmapMemObject`(pinnedBuffer, pinnedMemory)

The pinning cost is incurred at step a. Step d does not incur any pinning cost. Typically, an application performs steps a, b, c, and e once. It then repeatedly reads or modifies the data in pinnedMemory, followed by step d.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

For the second way to use this option, `clEnqueueRead/WriteBuffer` is used directly on a user memory buffer. The standard `clEnqueueRead/Write` calls require to pin (lock in memory) memory pages before they can be copied (by the DMA engine). This creates a performance penalty that is proportional to the buffer size. The performance of this path is currently about two-thirds of peak interconnect bandwidth.

Option 2 - `clEnqueueCopyBuffer()` on a pre-pinned host buffer (requires pre-pinned buffer support)

This is analogous to Option 1. Performing a CL copy of a pre-pinned buffer to a device buffer (or vice versa) runs at peak interconnect bandwidth.

a. pinnedBuffer = `clCreateBuffer( CL_MEM_ALLOC_HOST_PTR or CL_MEM_USE_HOST_PTR )`
b. deviceBuffer = `clCreateBuffer()`

This is followed either by:

c. void *memory = `clEnqueueMapBuffer`(pinnedBuffer)
d. Application writes or modifies memory.
e. `clEnqueueUnmapMemObject`(pinnedBuffer, memory)
f. `clEnqueueCopyBuffer`(pinnedBuffer, deviceBuffer)

or by:

g. `clEnqueueCopyBuffer`(deviceBuffer, pinnedBuffer)
h. void *memory = `clEnqueueMapBuffer`(pinnedBuffer)
i. Application reads memory.
j. `clEnqueueUnmapMemObject`(pinnedBuffer, memory)

Since the pinnedBuffer resides in host memory, the `clMap()` and `clUnmap()` calls do not result in data transfers, and they are of very low latency. Sparse or dense memory operations by the application take place at host memory bandwidth.

Option 3 - `clEnqueueMapBuffer()` and `clEnqueueUnmapMemObject() of a Device Buffer`

This is a good choice if the application fills in the data on the fly, or requires a pointer for calls to other library functions (such as `fread()` or `fwrite()`). An optimized path exists for regular device buffers; this path provides peak interconnect bandwidth at map/unmap time.

For buffers already allocated through `malloc()` or `mmap()`, the total transfer cost includes a `memcpy()` into the mapped device buffer, in addition to the interconnect transfer. Typically, this is slower than option 1), above.

The transfer sequence is as follows:

a. Data transfer from host to device buffer.
   1. `ptr = clEnqueueMapBuffer( .., buf, .., CL_MAP_WRITE, .. )`

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Since the buffer is mapped write-only, no data is transferred from device buffer to host. The map operation is very low cost. A pointer to a pinned host buffer is returned.

2. The application fills in the host buffer through `memset( ptr )`, `memcpy ( ptr, srcptr )`, `fread( ptr )`, or direct CPU writes. This happens at host memory bandwidth.
3. `clEnqueueUnmapMemObject( .., buf, ptr, .. )`

   The pre-pinned buffer is transferred to the GPU device, at peak interconnect bandwidth.

b. Data transfer from device buffer to host.

1. `ptr = clEnqueueMapBuffer(.., buf, .., CL_MAP_READ, .. )`

   This command triggers a transfer from the device to host memory, into a pre-pinned temporary buffer, at peak interconnect bandwidth. A pointer to the pinned memory is returned.
2. The application reads and processes the data, or executes a `memcpy( dstptr, ptr )`, `fwrite (ptr)`, or similar function. Since the buffer resides in host memory, this happens at host memory bandwidth.
3. `clEnqueueUnmapMemObject( .., buf, ptr, .. )`

   Since the buffer was mapped as read-only, no transfer takes place, and the unmap operation is very low cost.

Option 4 - Direct host access to a zero copy device buffer (requires zero copy support)

This option allows overlapping of data transfers and GPU compute. It is also useful for sparse write updates under certain constraints.

a. A zero copy buffer on the device is created using the following command:

```
buf = clCreateBuffer ( .., CL_MEM_USE_PERSISTENT_MEM_AMD, .. )
```

This buffer can be directly accessed by the host CPU, using the uncached WC path. This can take place at the same time the GPU executes a compute kernel. A common double buffering scheme has the kernel process data from one buffer while the CPU fills a second buffer. See the TransferOverlap code sample.

A zero copy device buffer can also be used to for sparse updates, such as assembling sub-rows of a larger matrix into a smaller, contiguous block for GPU processing. Due to the WC path, it is a good design choice to try to align writes to the cache line size, and to pick the write block size as large as possible.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

b. Transfer from the host to the device.

   1. `ptr = clEnqueueMapBuffer( .., buf, .., CL_MAP_WRITE, .. )`

      This operation is low cost because the zero copy device buffer is directly mapped into the host address space.

   2. The application transfers data via `memset( ptr )`, `memcpy( ptr, srcptr )`, or direct CPU writes.

      The CPU writes directly across the interconnect into the zero copy device buffer. Depending on the chipset, the bandwidth can be of the same order of magnitude as the interconnect bandwidth, although it typically is lower than peak.

   3. `clEnqueueUnmapMemObject( .., buf, ptr, .. )`

      As with the preceding map, this operation is low cost because the buffer continues to reside on the device.

c. If the buffer content must be read back later, use

   `clEnqueueReadBuffer( .., buf, ..)` or
   `clEnqueueCopyBuffer( .., buf, zero copy host buffer, .. ).`

   This bypasses slow host reads through the uncached path.

Option 5 - Direct GPU access to a zero copy host buffer (requires zero copy support)

This option allows direct reads or writes of host memory by the GPU. A GPU kernel can import data from the host without explicit transfer, and write data directly back to host memory. An ideal use is to perform small I/Os straight from the kernel, or to integrate the transfer latency directly into the kernel execution time.

a. The application creates a zero copy host buffer.

   `buf = clCreateBuffer( .., CL_MEM_ALLOC_HOST_PTR, .. )`

b. Next, the application modifies or reads the zero copy host buffer.

   1. `ptr = clEnqueueMapBuffer( .., buf, .., CL_MAP_READ | CL_MAP_WRITE, .. )`

      This operation is very low cost because it is a map of a buffer already residing in host memory.

   2. The application modifies the data through `memset( ptr )`, `memcpy(`in either direction`)`, sparse or dense CPU reads or writes. Since the application is modifying a host buffer, these operations take place at host memory bandwidth.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

3. `clEnqueueUnmapMemObject( .., buf, ptr, .. )`

   As with the preceding map, this operation is very low cost because the buffer continues to reside in host memory.

c. The application runs `clEnqueueNDRangeKernel()`, using buffers of this type as input or output. GPU kernel reads and writes go across the interconnect to host memory, and the data transfer becomes part of the kernel execution.

   The achievable bandwidth depends on the platform and chipset, but can be of the same order of magnitude as the peak interconnect bandwidth. For discrete graphics cards, it is important to note that resulting GPU kernel bandwidth is an order of magnitude lower compared to a kernel accessing a regular device buffer located on the device.

d. Following kernel execution, the application can access data in the host buffer in the same manner as described above.

4.7 Using Multiple OpenCL Devices

The AMD OpenCL runtime supports both CPU and GPU devices. This section introduces techniques for appropriately partitioning the workload and balancing it across the devices in the system.

4.7.1 CPU and GPU Devices

Table 4.4 lists some key performance characteristics of two exemplary CPU and GPU devices: a quad-core AMD Phenom II X4 processor running at 2.8 GHz, and a mid-range AMD Radeon™ HD 7770 GPU running at 1 GHz. The "best" device in each characteristic is highlighted, and the ratio of the best/other device is shown in the final column.

The GPU excels at high-throughput: the peak execution rate (measured in FLOPS) is 7X higher than the CPU, and the memory bandwidth is 2.5X higher than the CPU. The GPU also consumes approximately 65% the power of the CPU; thus, for this comparison, the power efficiency in flops/watt is 10X higher. While power efficiency can vary significantly with different devices, GPUs generally provide greater power efficiency (flops/watt) than CPUs because they optimize for throughput and eliminate hardware designed to hide latency.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Table 4.4 CPU and GPU Performance Characteristics

	CPU	GPU	Winner Ratio
Example Device	AMD Phenom™ II X4	AMD Radeon™ HD 7770	
Core Frequency	2800 MHz	1 GHz	3 X
Compute Units	4	10	2.5 X
Approx. Power[1]	95 W	80 W	1.2 X
Approx. Power/Compute Unit	19 W	8 W	2.4 X
Peak Single-Precision Billion Floating-Point Ops/Sec	90	1280	14 X
Approx GFLOPS/Watt	0.9	16	18 X
Max In-flight HW Threads	4	25600	6400 X
Simultaneous Executing Threads	4	640	160 X
Memory Bandwidth	26 GB/s	72 GB/s	2.8 X
Int Add latency	0.4 ns	4 ns	10 X
FP Add Latency	1.4 ns	4 ns	2.9 X
Approx DRAM Latency	50 ns	270 ns	5.4 X
L2+L3 (GPU only L2) cache capacity	8192 KB	128 kB	64 X
Approx Kernel Launch Latency	25 µs	50 µs	2 X

1. For the power specifications of the AMD Phenom™ II x4, see http://www.amd.com/us/products/desktop/processors/phenom-ii/Pages/phenom-ii-model-number-comparison.aspx.

Conversely, CPUs excel at latency-sensitive tasks. For example, an integer add is 10X faster on the CPU than on the GPU. This is a product of both the CPUs higher clock rate (2800 MHz vs 1000 MHz for this comparison), as well as the operation latency; the CPU is optimized to perform an integer add in just one cycle, while the GPU requires four cycles. The CPU also has a latency-optimized path to DRAM, while the GPU optimizes for bandwidth and relies on many in-flight threads to hide the latency. The AMD Radeon™ HD 7770 GPU, for example, supports more than 25,000 in-flight work-items and can switch to a new wavefront (containing up to 64 work-items) in a single cycle. The CPU supports only four hardware threads, and thread-switching requires saving and restoring the CPU registers from memory. The GPU requires many active threads to both keep the execution resources busy, as well as provide enough threads to hide the long latency of cache misses.

Each GPU thread has its own register state, which enables the fast single-cycle switching between threads. Also, GPUs can be very efficient at gather/scatter operations: each thread can load from any arbitrary address, and the registers are completely decoupled from the other threads. This is substantially more flexible and higher-performing than a classic Vector ALU-style architecture (such as SSE on the CPU), which typically requires that data be accessed from contiguous and aligned memory locations. SSE supports instructions that write parts of a register (for example, `MOVLPS` and `MOVHPS`, which write the upper and lower halves, respectively, of an SSE register), but these instructions generate additional microarchitecture dependencies and frequently require additional pack instructions to format the data correctly.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

In contrast, each GPU thread shares the same program counter with 63 other threads in a wavefront. Divergent control-flow on a GPU can be quite expensive and can lead to significant under-utilization of the GPU device. When control flow substantially narrows the number of valid work-items in a wave-front, it can be faster to use the CPU device.

CPUs also tend to provide significantly more on-chip cache than GPUs. In this example, the CPU device contains 512 kB L2 cache/core plus a 6 MB L3 cache that is shared among all cores, for a total of 8 MB of cache. In contrast, the GPU device contains only 128 kB cache shared by the five compute units. The larger CPU cache serves both to reduce the average memory latency and to reduce memory bandwidth in cases where data can be re-used from the caches.

Finally, note the approximate 2X difference in kernel launch latency. The GPU launch time includes both the latency through the software stack, as well as the time to transfer the compiled kernel and associated arguments across the PCI-express bus to the discrete GPU. Notably, the launch time does not include the time to compile the kernel. The CPU can be the device-of-choice for small, quick-running problems when the overhead to launch the work on the GPU outweighs the potential speedup. Often, the work size is data-dependent, and the choice of device can be data-dependent as well. For example, an image-processing algorithm may run faster on the GPU if the images are large, but faster on the CPU when the images are small.

The differences in performance characteristics present interesting optimization opportunities. Workloads that are large and data parallel can run orders of magnitude faster on the GPU, and at higher power efficiency. Serial or small parallel workloads (too small to efficiently use the GPU resources) often run significantly faster on the CPU devices. In some cases, the same algorithm can exhibit both types of workload. A simple example is a reduction operation such as a sum of all the elements in a large array. The beginning phases of the operation can be performed in parallel and run much faster on the GPU. The end of the operation requires summing together the partial sums that were computed in parallel; eventually, the width becomes small enough so that the overhead to parallelize outweighs the computation cost, and it makes sense to perform a serial add. For these serial operations, the CPU can be significantly faster than the GPU.

4.7.2 When to Use Multiple Devices

One of the features of GPU computing is that some algorithms can run substantially faster and at better energy efficiency compared to a CPU device. Also, once an algorithm has been coded in the data-parallel task style for OpenCL, the same code typically can scale to run on GPUs with increasing compute capability (that is more compute units) or even multiple GPUs (with a little more work).

For some algorithms, the advantages of the GPU (high computation throughput, latency hiding) are offset by the advantages of the CPU (low latency, caches, fast launch time), so that the performance on either devices is similar. This case is

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

more common for mid-range GPUs and when running more mainstream algorithms. If the CPU and the GPU deliver similar performance, the user can get the benefit of either improved power efficiency (by running on the GPU) or higher peak performance (use both devices).

Usually, when the data size is small, it is faster to use the CPU because the start-up time is quicker than on the GPU due to a smaller driver overhead and avoiding the need to copy buffers from the host to the device.

4.7.3 Partitioning Work for Multiple Devices

By design, each OpenCL command queue can only schedule work on a single OpenCL device. Thus, using multiple devices requires the developer to create a separate queue for each device, then partition the work between the available command queues.

A simple scheme for partitioning work between devices would be to statically determine the relative performance of each device, partition the work so that faster devices received more work, launch all the kernels, and then wait for them to complete. In practice, however, this rarely yields optimal performance. The relative performance of devices can be difficult to determine, in particular for kernels whose performance depends on the data input. Further, the device performance can be affected by dynamic frequency scaling, OS thread scheduling decisions, or contention for shared resources, such as shared caches and DRAM bandwidth. Simple static partitioning algorithms which “guess wrong” at the beginning can result in significantly lower performance, since some devices finish and become idle while the whole system waits for the single, unexpectedly slow device.

For these reasons, a dynamic scheduling algorithm is recommended. In this approach, the workload is partitioned into smaller parts that are periodically scheduled onto the hardware. As each device completes a part of the workload, it requests a new part to execute from the pool of remaining work. Faster devices, or devices which work on easier parts of the workload, request new input faster, resulting in a natural workload balancing across the system. The approach creates some additional scheduling and kernel submission overhead, but dynamic scheduling generally helps avoid the performance cliff from a single bad initial scheduling decision, as well as higher performance in real-world system environments (since it can adapt to system conditions as the algorithm runs).

Multi-core runtimes, such as Cilk, have already introduced dynamic scheduling algorithms for multi-core CPUs, and it is natural to consider extending these scheduling algorithms to GPUs as well as CPUs. A GPU introduces several new aspects to the scheduling process:

- **Heterogeneous Compute Devices**

  Most existing multi-core schedulers target only homogenous computing devices. When scheduling across both CPU and GPU devices, the scheduler must be aware that the devices can have very different performance

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

characteristics (10X or more) for some algorithms. To some extent, dynamic scheduling is already designed to deal with heterogeneous workloads (based on data input the same algorithm can have very different performance, even when run on the same device), but a system with heterogeneous devices makes these cases more common and more extreme. Here are some suggestions for these situations.

- The scheduler should support sending different workload sizes to different devices. GPUs typically prefer larger grain sizes, and higher-performing GPUs prefer still larger grain sizes.
- The scheduler should be conservative about allocating work until after it has examined how the work is being executed. In particular, it is important to avoid the performance cliff that occurs when a slow device is assigned an important long-running task. One technique is to use small grain allocations at the beginning of the algorithm, then switch to larger grain allocations when the device characteristics are well-known.
- As a special case of the above rule, when the devices are substantially different in performance (perhaps 10X), load-balancing has only a small potential performance upside, and the overhead of scheduling the load probably eliminates the advantage. In the case where one device is far faster than everything else in the system, use only the fast device.
- The scheduler must balance small-grain-size (which increase the adaptiveness of the schedule and can efficiently use heterogeneous devices) with larger grain sizes (which reduce scheduling overhead). Note that the grain size must be large enough to efficiently use the GPU.

- **Asynchronous Launch**

  OpenCL devices are designed to be scheduled asynchronously from a command-queue. The host application can enqueue multiple kernels, flush the kernels so they begin executing on the device, then use the host core for other work. The AMD OpenCL implementation uses a separate thread for each command-queue, so work can be transparently scheduled to the GPU in the background.

  Avoid starving the high-performance GPU devices. This can occur if the physical CPU core, which must re-fill the device queue, is itself being used as a device. A simple approach to this problem is to dedicate a physical CPU core for scheduling chores. The device fission extension (see Section A.7, "cl_ext Extensions," page A-4) can be used to reserve a core for scheduling. For example, on a quad-core device, device fission can be used to create an OpenCL device with only three cores.

  Another approach is to schedule enough work to the device so that it can tolerate latency in additional scheduling. Here, the scheduler maintains a watermark of uncompleted work that has been sent to the device, and refills the queue when it drops below the watermark. This effectively increase the grain size, but can be very effective at reducing or eliminating device starvation. Developers cannot directly query the list of commands in the OpenCL command queues; however, it is possible to pass an event to each `clEnqueue` call that can be queried, in order to determine the execution

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

status (in particular the command completion time); developers also can maintain their own queue of outstanding requests.

For many algorithms, this technique can be effective enough at hiding latency so that a core does not need to be reserved for scheduling. In particular, algorithms where the work-load is largely known up-front often work well with a deep queue and watermark. Algorithms in which work is dynamically created may require a dedicated thread to provide low-latency scheduling.

- **Data Location**

  Discrete GPUs use dedicated high-bandwidth memory that exists in a separate address space. Moving data between the device address space and the host requires time-consuming transfers over a relatively slow PCI-Express bus. Schedulers should be aware of this cost and, for example, attempt to schedule work that consumes the result on the same device producing it.

  CPU and GPU devices share the same memory bandwidth, which results in additional interactions of kernel executions.

4.7.4 Synchronization Caveats

The OpenCL functions that enqueue work (`clEnqueueNDRangeKernel`) merely enqueue the requested work in the command queue; they do not cause it to begin executing. Execution begins when the user executes a synchronizing command, such as `clFlush` or `clWaitForEvents`. Enqueuing several commands before flushing can enable the host CPU to batch together the command submission, which can reduce launch overhead.

Command-queues that are configured to execute in-order are guaranteed to complete execution of each command before the next command begins. This synchronization guarantee can often be leveraged to avoid explicit `clWaitForEvents()` calls between command submissions. Using `clWaitForEvents()` requires intervention by the host CPU and additional synchronization cost between the host and the GPU; by leveraging the in-order queue property, back-to-back kernel executions can be efficiently handled directly on the GPU hardware.

AMD Southern Islands GPUs can execute multiple kernels simultaneously when there are no dependencies.

The AMD OpenCL implementation spawns a new thread to manage each command queue. Thus, the OpenCL host code is free to manage multiple devices from a single host thread. Note that `clFinish` is a blocking operation; the thread that calls `clFinish` blocks until all commands in the specified command-queue have been processed and completed. If the host thread is managing multiple devices, it is important to call `clFlush` for each command-queue before calling `clFinish`, so that the commands are flushed and execute in parallel on the devices. Otherwise, the first call to `clFinish` blocks, the

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

commands on the other devices are not flushed, and the devices appear to execute serially rather than in parallel.

For low-latency CPU response, it can be more efficient to use a dedicated spin loop and not call `clFinish()` Calling `clFinish()` indicates that the application wants to wait for the GPU, putting the thread to sleep. For low latency, the application should use `clFlush()`, followed by a loop to wait for the event to complete. This is also true for blocking maps. The application should use non-blocking maps followed by a loop waiting on the event. The following provides sample code for this.

```
if (sleep)
   {
      // this puts host thread to sleep, useful if power is a consideration
         or overhead is not a concern
      clFinish(cmd_queue_);
   }
   else
   {
      // this keeps the host thread awake, useful if latency is a concern
      clFlush(cmd_queue_);
      error_ = clGetEventInfo(event, CL_EVENT_COMMAND_EXECUTION_STATUS,
      sizeof(cl_int), &eventStatus, NULL);
      while (eventStatus > 0)
   {
      error_ = clGetEventInfo(event, CL_EVENT_COMMAND_EXECUTION_STATUS,
      sizeof(cl_int), &eventStatus, NULL);
      Sleep(0);      // be nice to other threads, allow scheduler to find
                        other work if possible
      // Choose your favorite way to yield, SwitchToThread() for example,
          in place of Sleep(0)
      }
   }
```

4.7.5 GPU and CPU Kernels

While OpenCL provides functional portability so that the same kernel can run on any device, peak performance for each device is typically obtained by tuning the OpenCL kernel for the target device.

Code optimized for the Tahiti device (the AMD Radeon™ HD 7970 GPU) typically runs well across other members of the Southern Islands family.

CPUs and GPUs have very different performance characteristics, and some of these impact how one writes an optimal kernel. Notable differences include:

- The Vector ALU floating point resources in a CPU (SSE/AVX) require the use of vectorized types (such as float4) to enable packed SSE code generation and extract good performance from the Vector ALU hardware. The GPU Vector ALU hardware is more flexible and can efficiently use the floating-point hardware; however, code that can use float4 often generates hi-quality code for both the CPU and the AMD GPUs.
- The AMD OpenCL CPU implementation runs work-items from the same work-group back-to-back on the same physical CPU core. For optimally

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

coalesced memory patterns, a common access pattern for GPU-optimized algorithms is for work-items in the same wavefront to access memory locations from the same cache line. On a GPU, these work-items execute in parallel and generate a coalesced access pattern. On a CPU, the first work-item runs to completion (or until hitting a barrier) before switching to the next. Generally, if the working set for the data used by a work-group fits in the CPU caches, this access pattern can work efficiently: the first work-item brings a line into the cache hierarchy, which the other work-items later hit. For large working-sets that exceed the capacity of the cache hierarchy, this access pattern does not work as efficiently; each work-item refetches cache lines that were already brought in by earlier work-items but were evicted from the cache hierarchy before being used. Note that AMD CPUs typically provide 512 kB to 2 MB of L2+L3 cache for each compute unit.

- CPUs do not contain any hardware resources specifically designed to accelerate local memory accesses. On a CPU, local memory is mapped to the same cacheable DRAM used for global memory, and there is no performance benefit from using the `__local` qualifier. The additional memory operations to write to LDS, and the associated barrier operations can reduce performance. One notable exception is when local memory is used to pack values to avoid non-coalesced memory patterns.
- CPU devices only support a small number of hardware threads, typically two to eight. Small numbers of active work-group sizes reduce the CPU switching overhead, although for larger kernels this is a second-order effect.

For a balanced solution that runs reasonably well on both devices, developers are encouraged to write the algorithm using float4 vectorization. The GPU is more sensitive to algorithm tuning; it also has higher peak performance potential. Thus, one strategy is to target optimizations to the GPU and aim for reasonable performance on the CPU. For peak performance on all devices, developers can choose to use conditional compilation for key code loops in the kernel, or in some cases even provide two separate kernels. Even with device-specific kernel optimizations, the surrounding host code for allocating memory, launching kernels, and interfacing with the rest of the program generally only needs to be written once.

Another approach is to leverage a CPU-targeted routine written in a standard high-level language, such as C++. In some cases, this code path may already exist for platforms that do not support an OpenCL device. The program uses OpenCL for GPU devices, and the standard routine for CPU devices. Load-balancing between devices can still leverage the techniques described in Section 4.7.3, “Partitioning Work for Multiple Devices,” page 4-32.

4.7.6 Contexts and Devices

The AMD OpenCL program creates at least one context, and each context can contain multiple devices. Thus, developers must choose whether to place all devices in the same context or create a new context for each device. Generally, it is easier to extend a context to support additional devices rather than duplicating the context for each device: buffers are allocated at the context level

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

(and automatically across all devices), programs are associated with the context, and kernel compilation (via `clBuildProgram`) can easily be done for all devices in a context. However, with current OpenCL implementations, creating a separate context for each device provides more flexibility, especially in that buffer allocations can be targeted to occur on specific devices. Generally, placing the devices in the same context is the preferred solution.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Chapter 5
OpenCL Performance and Optimization for Southern Islands Devices

This chapter discusses performance and optimization when programming for AMD Accelerated Parallel Processing GPU compute devices that are part of the Southern Islands family, as well as CPUs and multiple devices. Details specific to the Evergreen and Northern Islands families of GPUs are provided in Chapter 6, “OpenCL Performance and Optimization for Evergreen and Northern Islands Devices.”

5.1 Global Memory Optimization

Figure 5.1 is a block diagram of the GPU memory system. The up arrows are read paths, the down arrows are write paths. WC is the write combine cache.

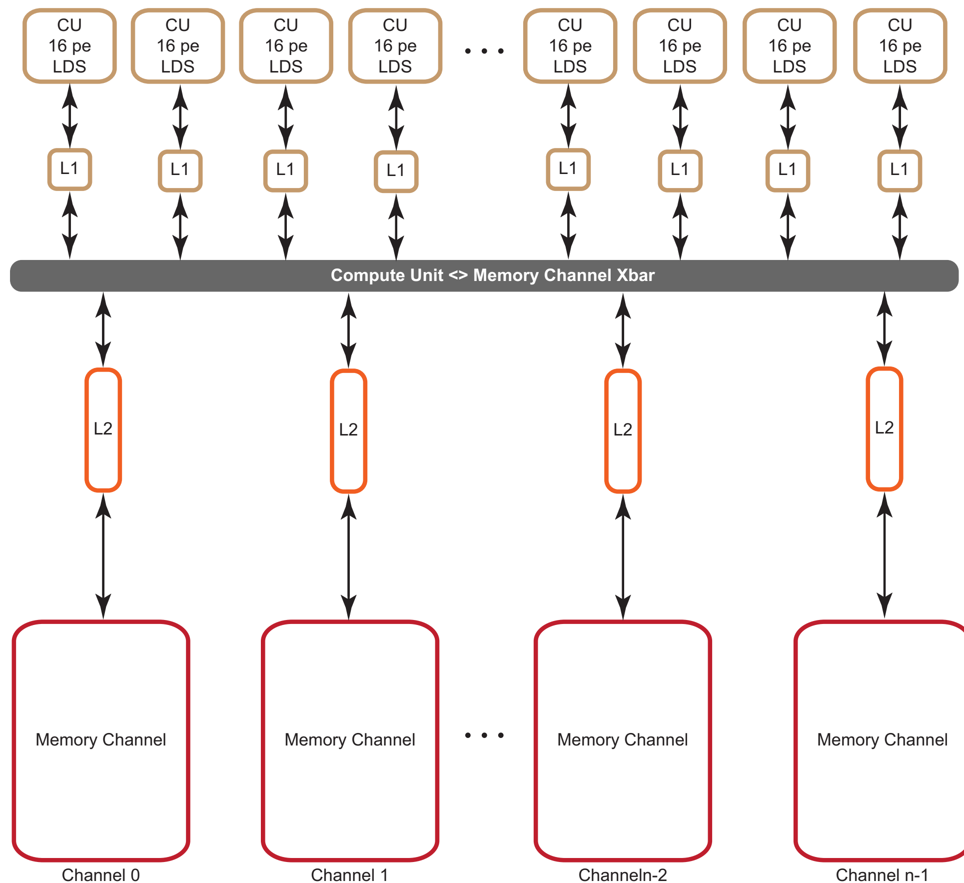

Figure 5.1 Memory System

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

The GPU consists of multiple compute units. Each compute unit contains local (on-chip) memory, L1 cache, registers, and 16 processing element (PE). Individual work-items execute on a single processing element; one or more work-groups execute on a single compute unit. On a GPU, hardware schedules groups of work-items, called wavefronts, onto compute units; thus, work-items within a wavefront execute in lock-step; the same instruction is executed on different data.

Each compute unit contains 64 kB local memory, 16 kB of read/write L1 cache, four vector units, and one scalar unit. The maximum local memory allocation is 32 kB per work-group. Each vector unit contains 512 scalar registers (SGPRs) for handling branching, constants, and other data constant across a wavefront. Vector units also contain 256 vector registers (VGPRs). VGPRs actually are scalar registers, but they are replicated across the whole wavefront. Vector units contain 16 processing elements (PEs). Each PE is scalar.

Since the L1 cache is 16 kB per compute unit, the total L1 cache size is 16 kB * (# of compute units). For the AMD Radeon™ HD 7970, this means a total of 512 kB L1 cache. L1 bandwidth can be computed as:

L1 peak bandwidth = Compute Units * (4 threads/clock) * (128 bits per thread) * (1 byte / 8 bits) * Engine Clock

For the AMD Radeon™ HD 7970, this is ~1.9 TB/s.

The peak memory bandwidth of your device is available in Appendix D, “Device Parameters.”

If two memory access requests are directed to the same controller, the hardware serializes the access. This is called a *channel conflict*. Similarly, if two memory access requests go to the same memory bank, hardware serializes the access. This is called a *bank conflict*. From a developer’s point of view, there is not much difference between channel and bank conflicts. Often, a large power of two stride results in a channel conflict. The size of the power of two stride that causes a specific type of conflict depends on the chip. A stride that results in a channel conflict on a machine with eight channels might result in a bank conflict on a machine with four.

In this document, the term bank conflict is used to refer to either kind of conflict.

5.1.1 Channel Conflicts

The important concept is memory stride: the increment in memory address, measured in elements, between successive elements fetched or stored by consecutive work-items in a kernel. Many important kernels do not exclusively use simple stride one accessing patterns; instead, they feature large non-unit strides. For instance, many codes perform similar operations on each dimension of a two- or three-dimensional array. Performing computations on the low dimension can often be done with unit stride, but the strides of the computations in the other dimensions are typically large values. This can result in significantly degraded performance when the codes are ported unchanged to GPU systems.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

A CPU with caches presents the same problem, large power-of-two strides force data into only a few cache lines.

One solution is to rewrite the code to employ array transpositions between the kernels. This allows all computations to be done at unit stride. Ensure that the time required for the transposition is relatively small compared to the time to perform the kernel calculation.

For many kernels, the reduction in performance is sufficiently large that it is worthwhile to try to understand and solve this problem.

In GPU programming, it is best to have adjacent work-items read or write adjacent memory addresses. This is one way to avoid channel conflicts.

When the application has complete control of the access pattern and address generation, the developer must arrange the data structures to minimize bank conflicts. Accesses that differ in the lower bits can run in parallel; those that differ only in the upper bits can be serialized.

In this example:

```
for (ptr=base; ptr<max; ptr += 16KB)
     R0 = *ptr ;
```

where the lower bits are all the same, the memory requests all access the same bank on the same channel and are processed serially.

This is a low-performance pattern to be avoided. When the stride is a power of 2 (and larger than the channel interleave), the loop above only accesses one channel of memory.

The hardware byte address bits are:

31:x	bank	channel	7:0 address

- On all AMD Radeon™ HD 79XX-series GPUs, there are 12 channels. A crossbar distributes the load to the appropriate memory channel. Each memory channel has a read/write global L2 cache, with 64 kB per channel. The cache line size is 64 bytes.

  Because 12 channels are not a part of the power of two memory and bank channel addressing, this is not straightforward for the AMD Radeon™ HD 79XX series. The memory channels are grouped in four quadrants, each which consisting of three channels. Bits 8, 9, and 10 of the address select a “virtual pipe.” The top two bits of this pipe select the quadrant; then, the channel within the quadrant is selected using the low bit of the pipe and the row and bank address modulo three, according to the following conditional equation.

  ```
  If (({ row, bank} %3) == 1)
       channel_within_quadrant =  1
  else
      channel_within_quadrant = 2 * pipe[0]
  ```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Figure 5.2 illustrates the memory channel mapping.

address	0	256	512	768	1024	1280	1536	1792
0	0	2	3	5	6	8	9	11
2048	1	1	4	4	7	7	10	10
4096	0	2	3	5	6	8	9	11
6144	0	2	3	5	6	8	9	11
8192	1	1	4	4	7	7	10	10
10240	0	2	3	5	6	8	9	11
12288	0	2	3	5	6	8	9	11
14336	1	1	4	4	7	7	10	10
16384	0	2	3	5	6	8	9	11
18432	0	2	3	5	6	8	9	11
20480	1	1	4	4	7	7	10	10
22528	0	2	3	5	6	8	9	11
24576	0	2	3	5	6	8	9	11
26624	1	1	4	4	7	7	10	10
28672	0	2	3	5	6	8	9	11
30720	0	2	3	5	6	8	9	11
32768	0	2	3	5	6	8	9	11
34816	1	1	4	4	7	7	10	10
36864	0	2	3	5	6	8	9	11
38912	0	2	3	5	6	8	9	11
40960	1	1	4	4	7	7	10	10
43008	0	2	3	5	6	8	9	11
45056	0	2	3	5	6	8	9	11
47104	1	1	4	4	7	7	10	10
49152	0	2	3	5	6	8	9	11
51200	0	2	3	5	6	8	9	11
53248	1	1	4	4	7	7	10	10
55296	0	2	3	5	6	8	9	11
57344	0	2	3	5	6	8	9	11
59392	1	1	4	4	7	7	10	10
61440	0	2	3	5	6	8	9	11
63488	0	2	3	5	6	8	9	11
65536	0	2	3	5	6	8	9	11
67584	1	1	4	4	7	7	10	10
69632	0	2	3	5	6	8	9	11
71680	0	2	3	5	6	8	9	11
73728	1	1	4	4	7	7	10	10
75776	0	2	3	5	6	8	9	11
77824	0	2	3	5	6	8	9	11
79872	1	1	4	4	7	7	10	10
81920	0	2	3	5	6	8	9	11
83968	0	2	3	5	6	8	9	11
86016	1	1	4	4	7	7	10	10
88064	0	2	3	5	6	8	9	11

Figure 5.2 Channel Remapping/Interleaving

Note that an increase of the address by 2048 results in a 1/3 probability the same channel is hit; increasing the address by 256 results in a 1/6 probability the same channel is hit, etc.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

- On all AMD Radeon™ HD 77XX- and 78XX-series GPUs, the lower eight bits select an element within a channel.
- The next set of bits select the channel. The number of channel bits varies, since the number of channels is not the same on all parts. With eight channels, three bits are used to select the channel; with two channels, a single bit is used.
- The next set of bits selects the memory bank. The number of bits used depends on the number of memory banks.
- The remaining bits are the rest of the address.

On AMD Radeon™ HD 78XX GPUs, the channel selection are bits 10:8 of the byte address. For the AMD Radeon™ HD 77XX, the channel selection are bits 9:8 of the byte address. This means a linear burst switches channels every 256 bytes. Since the wavefront size is 64, channel conflicts are avoided if each work-item in a wave reads a different address from a 64-word region. All AMD Radeon™ HD 7XXX series GPUs have the same layout: channel ends at bit 8, and the memory bank is to the left of the channel.

For AMD Radeon™ HD 77XX and 78XX GPUs, a burst of 2 kB (# of channels * 256 bytes) cycles through all the channels.

For AMD Radeon™ HD 77XX and 78XX GPUs, when calculating an address as y*width+x, but reading a burst on a column (incrementing y), only one memory channel of the system is used, since the width is likely a multiple of 256 words = 2048 bytes. If the width is an odd multiple of 256B, then it cycles through all channels.

If every work-item in a work-group references consecutive memory addresses and the address of work-item 0 is aligned to 256 bytes and each work-item fetches 32 bits, the entire wavefront accesses one channel. Although this seems slow, it actually is a fast pattern because it is necessary to consider the memory access over the entire device, not just a single wavefront.

One or more work-groups execute on each compute unit. On the AMD Radeon™ HD 7000-series GPUs, work-groups are dispatched in a linear order, with x changing most rapidly. For a single dimension, this is:

DispatchOrder = `get_group_id(0)`

For two dimensions, this is:

DispatchOrder = `get_group_id(0) + get_group_id(1) * get_num_groups(0)`

This is row-major-ordering of the blocks in the index space. Once all compute units are in use, additional work-groups are assigned to compute units as needed. Work-groups retire in order, so active work-groups are contiguous.

At any time, each compute unit is executing an instruction from a single wavefront. In memory intensive kernels, it is likely that the instruction is a memory access. Since there are 12 channels on the AMD Radeon™ HD 7970 GPU, at most 12 of the compute units can issue a memory access operation in

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

one cycle. It is most efficient if the accesses from 12 wavefronts go to different channels. One way to achieve this is for each wavefront to access consecutive groups of 256 = 64 * 4 bytes.

An inefficient access pattern is if each wavefront accesses all the channels. This is likely to happen if consecutive work-items access data that has a large power of two strides.

In the next example of a kernel for copying, the input and output buffers are interpreted as though they were 2D, and the work-group size is organized as 2D.

The kernel code is:

```
#define WIDTH 1024
#define DATA_TYPE float
#define A(y , x ) A[ (y) * WIDTH + (x ) ]
#define C(y , x ) C[ (y) * WIDTH+(x ) ]
kernel void copy_float (__global const
                        DATA_TYPE * A,
                        __global DATA_TYPE* C)
{
  int idx = get_global_id(0);
  int idy = get_global_id(1);
  C(idy, idx) = A( idy, idx);
}
```

By changing the width, the data type and the work-group dimensions, we get a set of kernels out of this code.

Given a 64x1 work-group size, each work-item reads a consecutive 32-bit address. Given a 1x64 work-group size, each work-item reads a value separated by the width in a power of two bytes.

To avoid power of two strides:

- Add an extra column to the data matrix.
- Change the work-group size so that it is not a power of 2[1].
- It is best to use a width that causes a rotation through all of the memory channels, instead of using the same one repeatedly.
- Change the kernel to access the matrix with a staggered offset.

5.1.1.1 Staggered Offsets

Staggered offsets apply a coordinate transformation to the kernel so that the data is processed in a different order. Unlike adding a column, this technique does not use extra space. It is also relatively simple to add to existing code.

Figure 5.3 illustrates the transformation to staggered offsets.

1. Generally, it is not a good idea to make the work-group size something other than an integer multiple of the wavefront size, but that usually is less important than avoiding channel conflicts.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

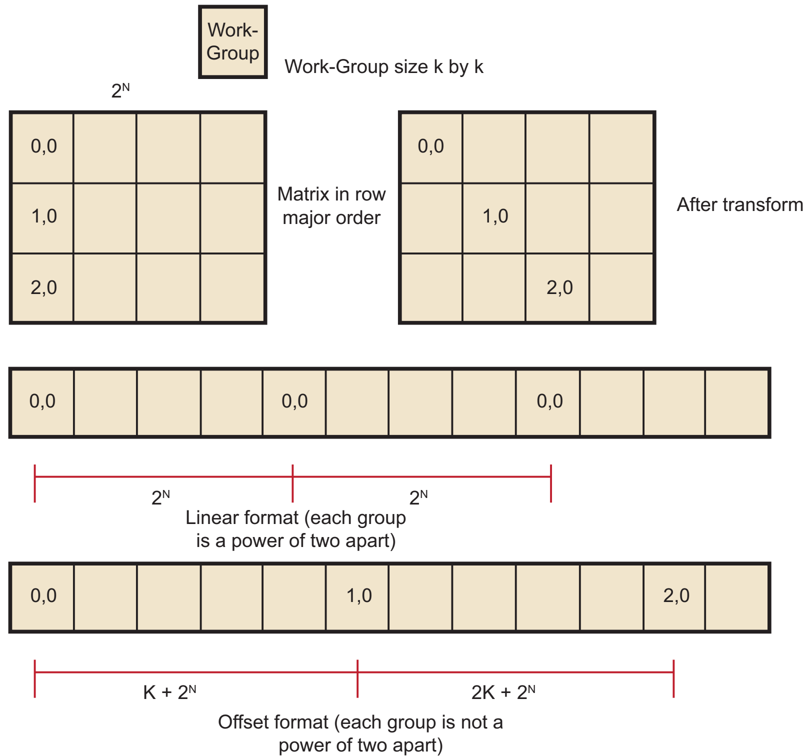

Figure 5.3 Transformation to Staggered Offsets

The global ID values reflect the order that the hardware initiates work-groups. The values of get group ID are in ascending launch order.

```
global_id(0) = get_group_id(0) * get_local_size(0) + get_local_id(0)
global_id(1) = get_group_id(1) * get_local_size(1) + get_local_id(1)
```

The hardware launch order is fixed, but it is possible to change the launch order, as shown in the following example.

Assume a work-group size of k x k, where k is a power of two, and a large 2D matrix of size 2^n x 2^m in row-major order. If each work-group must process a block in column-order, the launch order does not work out correctly: consecutive work-groups execute down the columns, and the columns are a large power-of-two apart; so, consecutive work-groups access the same channel.

By introducing a transformation, it is possible to stagger the work-groups to avoid channel conflicts. Since we are executing 2D work-groups, each work group is identified by four numbers.

1. `get_group_id(0)` - the x coordinate or the block within the column of the matrix.
2. `get_group_id(1)` - the y coordinate or the block within the row of the matrix.
3. `get_global_id(0)` - the x coordinate or the column of the matrix.
4. `get_global_id(1)` - the y coordinate or the row of the matrix.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

To transform the code, add the following four lines to the top of the kernel.

```
get_group_id_0 = get_group_id(0);
get_group_id_1 = (get_group_id(0) + get_group_id(1)) % get_local_size(0);
get_global_id_0 = get_group_id_0 * get_local_size(0) + get_local_id(0);
get_global_id_1 = get_group_id_1 * get_local_size(1) + get_local_id(1);
```

Then, change the global IDs and group IDs to the staggered form. The result is:

```
__kernel void
copy_float (
  __global const DATA_TYPE * A,
  __global DATA_TYPE * C)
{
  size_t get_group_id_0 = get_group_id(0);
  size_t get_group_id_1 = (get_group_id(0) + get_group_id(1)) %
                            get_local_size(0);

  size_t get_global_id_0 = get_group_id_0 * get_local_size(0) +
                             get_local_id(0);
  size_t get_global_id_1 = get_group_id_1 * get_local_size(1) +
                             get_local_id(1);

  int idx = get_global_id_0; //changed to staggered form
  int idy = get_global_id_1; //changed to staggered form

  C(idy , idx) = A( idy , idx);
}
```

5.1.1.2 Reads Of The Same Address

Under certain conditions, one unexpected case of a channel conflict is that reading from the same address is a conflict, even on the FastPath.

This does not happen on the read-only memories, such as constant buffers, textures, or shader resource view (SRV); but it is possible on the read/write UAV memory or OpenCL global memory.

From a hardware standpoint, reads from a fixed address have the same upper bits, so they collide and are serialized. To read in a single value, read the value in a single work-item, place it in local memory, and then use that location:

Avoid:

```
temp = input[3] // if input is from global space
```

Use:

```
if (get_local_id(0) == 0) {
  local = input[3]
}
barrier(CLK_LOCAL_MEM_FENCE);
temp = local
```

5.1.2 Coalesced Writes

Southern Island devices do not support coalesced writes; however, continuous addresses within work-groups provide maximum performance.

Each compute unit accesses the memory system in quarter-wavefront units. The compute unit transfers a 32-bit address and one element-sized piece of data for each work-item. This results in a total of 16 elements + 16 addresses per quarter-

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

wavefront. On GCN-based devices, processing quarter-wavefront requires two cycles before the data is transferred to the memory controller.

5.1.3 Hardware Variations

For a listing of the AMD GPU hardware variations, see Appendix D, “Device Parameters.” This appendix includes information on the number of memory channels, compute units, and the L2 size per device.

5.2 Local Memory (LDS) Optimization

AMD Southern Islands GPUs include a Local Data Store (LDS) cache, which accelerates local memory accesses. LDS provides high-bandwidth access (more than 10X higher than global memory), efficient data transfers between work-items in a work-group, and high-performance atomic support. LDS is much faster than L1 cache access as it has twice the peak bandwidth and far lower latency. Additionally, using LDS memory can reduce global memory bandwidth usage. Local memory offers significant advantages when the data is re-used; for example, subsequent accesses can read from local memory, thus reducing global memory bandwidth. Another advantage is that local memory does not require coalescing.

To determine local memory size:

```
clGetDeviceInfo( …, CL_DEVICE_LOCAL_MEM_SIZE, … );
```

All AMD Southern Islands GPUs contain a 64 kB LDS for each compute unit; although only 32 kB can be allocated per work-group. The LDS contains 32-banks, each bank is four bytes wide and 256 bytes deep; the bank address is determined by bits 6:2 in the address. Appendix D, “Device Parameters” shows how many LDS banks are present on the different AMD Southern Island devices. As shown below, programmers must carefully control the bank bits to avoid bank conflicts as much as possible. Bank conflicts are determined by what addresses are accessed on each half wavefront boundary. Threads 0 through 31 are checked for conflicts as are threads 32 through 63 within a wavefront.

In a single cycle, local memory can service a request for each bank (up to 32 accesses each cycle on the AMD Radeon™ HD 7970 GPU). For an AMD Radeon™ HD 7970 GPU, this delivers a memory bandwidth of over 100 GB/s for each compute unit, and more than 3.5 TB/s for the whole chip. This is more than 14X the global memory bandwidth. However, accesses that map to the same bank are serialized and serviced on consecutive cycles. LDS operations do not stall; however, the compiler inserts wait operations prior to issuing operations that depend on the results. A wavefront that generated bank conflicts does not stall implicitly, but may stall explicitly in the kernel if the compiler has inserted a wait command for the outstanding memory access. The GPU reprocesses the wavefront on subsequent cycles, enabling only the lanes receiving data, until all the conflicting accesses complete. The bank with the most conflicting accesses determines the latency for the wavefront to complete the local memory operation. The worst case occurs when all 64 work-items map to the same bank, since each

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

access then is serviced at a rate of one per clock cycle; this case takes 64 cycles to complete the local memory access for the wavefront. A program with a large number of bank conflicts (as measured by the `LDSBankConflict` performance counter in the AMD APP Profiler statistics) might benefit from using the constant or image memory rather than LDS.

Thus, the key to effectively using the local cache memory is to control the access pattern so that accesses generated on the same cycle map to different banks in the local memory. One notable exception is that accesses to the same address (even though they have the same bits 6:2) can be broadcast to all requestors and do not generate a bank conflict. The LDS hardware examines the requests generated over two cycles (32 work-items of execution) for bank conflicts. Ensure, as much as possible, that the memory requests generated from a quarter-wavefront avoid bank conflicts by using unique address bits 6:2. A simple sequential address pattern, where each work-item reads a float2 value from LDS, generates a conflict-free access pattern on the AMD Radeon™ HD 7XXX GPU. Note that a sequential access pattern, where each work-item reads a float4 value from LDS, uses only half the banks on each cycle on the AMD Radeon™ HD 7XXX GPU and delivers half the performance of the float access pattern.

Each stream processor can generate up to two 4-byte LDS requests per cycle. Byte and short reads consume four bytes of LDS bandwidth. Developers can use the large register file: each compute unit has 256 kB of register space available (8X the LDS size) and can provide up to twelve 4-byte values/cycle (6X the LDS bandwidth). Registers do not offer the same indexing flexibility as does the LDS, but for some algorithms this can be overcome with loop unrolling and explicit addressing.

LDS reads require one ALU operation to initiate them. Each operation can initiate two loads of up to four bytes each.

The AMD APP Profiler provides the following performance counter to help optimize local memory usage:

> `LDSBankConflict`: The percentage of time accesses to the LDS are stalled due to bank conflicts relative to GPU Time. In the ideal case, there are no bank conflicts in the local memory access, and this number is zero.

Local memory is software-controlled “scratchpad” memory. In contrast, caches typically used on CPUs monitor the access stream and automatically capture recent accesses in a tagged cache. The scratchpad allows the kernel to explicitly load items into the memory; they exist in local memory until the kernel replaces them, or until the work-group ends. To declare a block of local memory, use the `__local` keyword; for example:

```
__local float localBuffer[64]
```

These declarations can be either in the parameters to the kernel call or in the body of the kernel. The `__local` syntax allocates a single block of memory, which is shared across all work-items in the workgroup.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

To write data into local memory, write it into an array allocated with `__local`. For example:

```
localBuffer[i] = 5.0;
```

A typical access pattern is for each work-item to collaboratively write to the local memory: each work-item writes a subsection, and as the work-items execute in parallel they write the entire array. Combined with proper consideration for the access pattern and bank alignment, these collaborative write approaches can lead to highly efficient memory accessing.

The following example is a simple kernel section that collaboratively writes, then reads from, local memory:

```
__kernel void localMemoryExample (__global float *In, __global float *Out) {
    __local float localBuffer[64];
    uint tx = get_local_id(0);
    uint gx = get_global_id(0);

    // Initialize local memory:
    // Copy from this work-group's section of global memory to local:
    // Each work-item writes one element; together they write it all
    localBuffer[tx] = In[gx];

    // Ensure writes have completed:
    barrier(CLK_LOCAL_MEM_FENCE);

    // Toy computation to compute a partial factorial, shows re-use from local
    float f = localBuffer[tx];
    for (uint i=tx+1; i<64; i++) {
        f *= localBuffer[i];
    }
    Out[gx] = f;
}
```

Note the host code cannot read from, or write to, local memory. Only the kernel can access local memory.

Local memory is consistent across work-items only at a work-group barrier; thus, before reading the values written collaboratively, the kernel must include a `barrier()` instruction. An important optimization is the case where the local work-group size is less than, or equal to, the wavefront size. Because the wavefront executes as an atomic unit, the explicit barrier operation is not required. The compiler automatically removes these barriers if the kernel specifies a `reqd_work_group_size` (see section 5.8 of the *OpenCL Specification*) that is less than the wavefront size. Developers are strongly encouraged to include the barriers where appropriate, and rely on the compiler to remove the barriers when possible, rather than manually removing the `barriers()`. This technique results in more portable code, including the ability to run kernels on CPU devices.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

5.3 Constant Memory Optimization

Constants (data from read-only buffers shared by a wavefront) are loaded to SGPRs from memory through the L1 (and L2) cache using scalar memory read instructions. The scalar instructions can use up to two SGPR sources per cycle; vector instructions can use one SGPR source per cycle. (There are 512 SGPRs per SIMD, 4 SIMDs per CU; so a 32 CU configuration like Tahiti has 256 kB of SGPRs.)

Southern Islands hardware supports specific inline literal constants. These constants are “free” in that they do not increase code size:

0
integers 1.. 64
integers -1 .. -16
0.5 single or double floats
-0.5
1.0
-1.0
2.0
-2.0
4.0
-4.0

Any other literal constant increases the code size by at least 32 bits.

The AMD implementation of OpenCL provides three levels of performance for the “constant” memory type.

1. Simple Direct-Addressing Patterns

   Very high bandwidth can be attained when the compiler has available the constant address at compile time and can embed the constant address into the instruction. Each processing element can load up to 4x4-byte direct-addressed constant values each cycle. Typically, these cases are limited to simple non-array constants and function parameters. The executing kernel loads the constants into scalar registers and concurrently populates the constant cache. The cache is a tagged cache, typically each 8k blocks is shared among four compute units. If the constant data is already present in the constant cache, the load is serviced by the cache and does not require any global memory bandwidth. The constant cache size for each device is given in Appendix D, “Device Parameters”; it varies from 4k to 48k per GPU.

2. Same Index

   Hardware acceleration also takes place when all work-items in a wavefront reference the same constant address. In this case, the data is loaded from memory one time, stored in the L1 cache, and then broadcast to all wave-fronts. This can reduce significantly the required memory bandwidth.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

3. Varying Index

   More sophisticated addressing patterns, including the case where each work-item accesses different indices, are not hardware accelerated and deliver the same performance as a global memory read with the potential for cache hits.

To further improve the performance of the AMD OpenCL stack, two methods allow users to take advantage of hardware constant buffers. These are:

1. Globally scoped constant arrays. These arrays are initialized, globally scoped, and in the constant address space (as specified in section 6.5.3 of the OpenCL specification). If the size of an array is below 64 kB, it is placed in hardware constant buffers; otherwise, it uses global memory. An example of this is a lookup table for math functions.
2. Per-pointer attribute specifying the maximum pointer size. This is specified using the `max_constant_size(N)` attribute. The attribute form conforms to section 6.10 of the OpenCL 1.0 specification. This attribute is restricted to top-level kernel function arguments in the constant address space. This restriction prevents a pointer of one size from being passed as an argument to a function that declares a different size. It informs the compiler that indices into the pointer remain inside this range and it is safe to allocate a constant buffer in hardware, if it fits. Using a constant pointer that goes outside of this range results in undefined behavior. All allocations are aligned on the 16-byte boundary. For example:

```
kernel void mykernel(global int* a,
constant int* b __attribute__((max_constant_size (65536)))
)
{
size_t idx = get_global_id(0);
a[idx] = b[idx & 0x3FFF];
}
```

A kernel that uses constant buffers must use `CL_DEVICE_MAX_CONSTANT_ARGS` to query the device for the maximum number of constant buffers the kernel can support. This value might differ from the maximum number of hardware constant buffers available. In this case, if the number of hardware constant buffers is less than the `CL_DEVICE_MAX_CONSTANT_ARGS`, the compiler allocates the largest constant buffers in hardware first and allocates the rest of the constant buffers in global memory. As an optimization, if a constant pointer **A** uses *n* bytes of memory, where *n* is less than 64 kB, and constant pointer **B** uses *m* bytes of memory, where *m* is less than (64 kB – *n*) bytes of memory, the compiler can allocate the constant buffer pointers in a single hardware constant buffer. This optimization can be applied recursively by treating the resulting allocation as a single allocation and finding the next smallest constant pointer that fits within the space left in the constant buffer.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

5.4 OpenCL Memory Resources: Capacity and Performance

Table 5.1 summarizes the hardware capacity and associated performance for the structures associated with the five OpenCL Memory Types. This information specific to the AMD Radeon™ HD 7970 GPUs with 3 GB video memory. See Appendix D, "Device Parameters" for more details about other GPUs.

Table 5.1 Hardware Performance Parameters

OpenCL Memory Type	Hardware Resource	Size/CU	Size/GPU	Peak Read Bandwidth/ Stream Core
Private	GPRs	256k	8192k	12 bytes/cycle
Local	LDS	64k	2048k	8 bytes/cycle
Constant	Direct-addressed constant		48k	4 bytes/cycle
	Same-indexed constant			4 bytes/cycle
	Varying-indexed constant			~0.14 bytes/cycle
Images	L1 Cache	16k	512k[1]	4 bytes/cycle
	L2 Cache		768[2]k	~0.4 bytes/cycle
Global	Global Memory		3G	~0.14 bytes/cycle

1. Applies to images and buffers.
2. Applies to images and buffers.

The compiler tries to map private memory allocations to the pool of GPRs in the GPU. In the event GPRs are not available, private memory is mapped to the "scratch" region, which has the same performance as global memory. Section 5.6.2, "Resource Limits on Active Wavefronts," page 5-17, has more information on register allocation and identifying when the compiler uses the scratch region. GPRs provide the highest-bandwidth access of any hardware resource. In addition to reading up to 12 bytes/cycle per processing element from the register file, the hardware can access results produced in the previous cycle without consuming any register file bandwidth.

Same-indexed constants can be cached in the L1 and L2 cache. Note that "same-indexed" refers to the case where all work-items in the wavefront reference the same constant index on the same cycle. The performance shown assumes an L1 cache hit.

Varying-indexed constants, which are cached only in L2, use the same path as global memory access and are subject to the same bank and alignment constraints described in Section 5.1, "Global Memory Optimization," page 5-1.

The L1 and L2 read/write caches are constantly enabled. As of SDK 2.4, read only buffers can be cached in L1 and L2.

The L1 cache can service up to four address requests per cycle, each delivering up to 16 bytes. The bandwidth shown assumes an access size of 16 bytes; smaller access sizes/requests result in a lower peak bandwidth for the L1 cache. Using float4 with images increases the request size and can deliver higher L1 cache bandwidth.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Each memory channel on the GPU contains an L2 cache that can deliver up to 64 bytes/cycle. The AMD Radeon™ HD 7970 GPU has 12 memory channels; thus, it can deliver up to 768 bytes/cycle; divided among 2048 stream cores, this provides up to ~0.4 bytes/cycle for each stream core.

Global Memory bandwidth is limited by external pins, not internal bus bandwidth. The AMD Radeon™ HD 7970 GPU supports up to 264 GB/s of memory bandwidth which is an average of 0.14 bytes/cycle for each stream core.

Note that Table 5.1 shows the performance for the AMD Radeon™ HD 7970 GPU. The "Size/Compute Unit" column and many of the bandwidths/processing element apply to all Southern Islands-class GPUs; however, the "Size/GPU" column and the bandwidths for varying-indexed constant, L2, and global memory vary across different GPU devices. The resource capacities and peak bandwidth for other AMD GPU devices can be found in Appendix D, "Device Parameters."

5.5 Using LDS or L1 Cache

There are a number of considerations when deciding between LDS and L1 cache for a given algorithm.

LDS supports read/modify/write operations, as well as atomics. It is well-suited for code that requires fast read/write, read/modify/write, or scatter operations that otherwise are directed to global memory. On current AMD hardware, L1 is part of the read path; hence, it is suited to cache-read-sensitive algorithms, such as matrix multiplication or convolution.

LDS is typically larger than L1 (for example: 64 kB vs 16 kB on Southern Islands devices). If it is not possible to obtain a high L1 cache hit rate for an algorithm, the larger LDS size can help. On the AMD Radeon™ HD 7970 device, the theoretical LDS peak bandwidth is 3.8 TB/s, compared to L1 at 1.9 TB/sec.

The native data type for L1 is a four-vector of 32-bit words. On L1, fill and read addressing are linked. It is important that L1 is initially filled from global memory with a coalesced access pattern; once filled, random accesses come at no extra processing cost.

Currently, the native format of LDS is a 32-bit word. The theoretical LDS peak bandwidth is achieved when each thread operates on a two-vector of 32-bit words (16 threads per clock operate on 32 banks). If an algorithm requires coalesced 32-bit quantities, it maps well to LDS. The use of four-vectors or larger can lead to bank conflicts, although the compiler can mitigate some of these.

From an application point of view, filling LDS from global memory, and reading from it, are independent operations that can use independent addressing. Thus, LDS can be used to explicitly convert a scattered access pattern to a coalesced pattern for read and write to global memory. Or, by taking advantage of the LDS read broadcast feature, LDS can be filled with a coalesced pattern from global memory, followed by all threads iterating through the same LDS words simultaneously.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

LDS reuses the data already pulled into cache by other wavefronts. Sharing across work-groups is not possible because OpenCL does not guarantee that LDS is in a particular state at the beginning of work-group execution. L1 content, on the other hand, is independent of work-group execution, so that successive work-groups can share the content in the L1 cache of a given Vector ALU. However, it currently is not possible to explicitly control L1 sharing across work-groups.

The use of LDS is linked to GPR usage and wavefront-per-Vector ALU count. Better sharing efficiency requires a larger work-group, so that more work-items share the same LDS. Compiling kernels for larger work-groups typically results in increased register use, so that fewer wavefronts can be scheduled simultaneously per Vector ALU. This, in turn, reduces memory latency hiding. Requesting larger amounts of LDS per work-group results in fewer wavefronts per Vector ALU, with the same effect.

LDS typically involves the use of barriers, with a potential performance impact. This is true even for read-only use cases, as LDS must be explicitly filled in from global memory (after which a barrier is required before reads can commence).

5.6 NDRange and Execution Range Optimization

Probably the most effective way to exploit the potential performance of the GPU is to provide enough threads to keep the device completely busy. The programmer specifies a three-dimensional NDRange over which to execute the kernel; bigger problems with larger NDRanges certainly help to more effectively use the machine. The programmer also controls how the global NDRange is divided into local ranges, as well as how much work is done in each work-item, and which resources (registers and local memory) are used by the kernel. All of these can play a role in how the work is balanced across the machine and how well it is used. This section introduces the concept of latency hiding, how many wavefronts are required to hide latency on AMD GPUs, how the resource usage in the kernel can impact the active wavefronts, and how to choose appropriate global and local work-group dimensions.

5.6.1 Hiding ALU and Memory Latency

The read-after-write latency for most arithmetic operations (a floating-point add, for example) is only four cycles. For most Southern Island devices, each CU can execute 64 vector ALU instructions per cycle, 16 per wavefront. Also, a wavefront can issue a scalar ALU instruction every four cycles. To achieve peak ALU power, a minimum of four wavefronts must be scheduled for each CU.

Global memory reads generate a reference to the off-chip memory and experience a latency of 300 to 600 cycles. The wavefront that generates the global memory access is made idle until the memory request completes. During this time, the compute unit can process other independent wavefronts, if they are available.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Kernel execution time also plays a role in hiding memory latency: longer chains of ALU instructions keep the functional units busy and effectively hide more latency. To better understand this concept, consider a global memory access which takes 400 cycles to execute. Assume the compute unit contains many other wavefronts, each of which performs five ALU instructions before generating another global memory reference. As discussed previously, the hardware executes each instruction in the wavefront in four cycles; thus, all five instructions occupy the ALU for 20 cycles. Note the compute unit interleaves two of these wavefronts and executes the five instructions from both wavefronts (10 total instructions) in 40 cycles. To fully hide the 400 cycles of latency, the compute unit requires (400/40) = 10 pairs of wavefronts, or 20 total wavefronts. If the wavefront contains 10 instructions rather than 5, the wavefront pair would consume 80 cycles of latency, and only 10 wavefronts would be required to hide the 400 cycles of latency.

Generally, it is not possible to predict how the compute unit schedules the available wavefronts, and thus it is not useful to try to predict exactly which ALU block executes when trying to hide latency. Instead, consider the overall ratio of ALU operations to fetch operations – this metric is reported by the AMD APP Profiler in the `ALUFetchRatio` counter. Each ALU operation keeps the compute unit busy for four cycles, so you can roughly divide 500 cycles of latency by (`4*ALUFetchRatio`) to determine how many wavefronts must be in-flight to hide that latency. Additionally, a low value for the `ALUBusy` performance counter can indicate that the compute unit is not providing enough wavefronts to keep the execution resources in full use. (This counter also can be low if the kernel exhausts the available DRAM bandwidth. In this case, generating more wavefronts does not improve performance; it can reduce performance by creating more contention.)

Increasing the wavefronts/compute unit does not indefinitely improve performance; once the GPU has enough wavefronts to hide latency, additional active wavefronts provide little or no performance benefit. A closely related metric to wavefronts/compute unit is “occupancy,” which is defined as the ratio of active wavefronts to the maximum number of possible wavefronts supported by the hardware. Many of the important optimization targets and resource limits are expressed in wavefronts/compute units, so this section uses this metric rather than the related “occupancy” term.

5.6.2 Resource Limits on Active Wavefronts

AMD GPUs have two important global resource constraints that limit the number of in-flight wavefronts:

- Southern Islands devices support a maximum of 16 work-groups per CU if a work-group is larger than one wavefront.
- The maximum number of wavefronts that can be scheduled to a CU is 40, or 10 per Vector Unit.

These limits are largely properties of the hardware and, thus, difficult for developers to control directly. Fortunately, these are relatively generous limits.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Frequently, the register and LDS usage in the kernel determines the limit on the number of active wavefronts/compute unit, and these can be controlled by the developer.

5.6.2.1 GPU Registers

Southern Islands registers are scalar, so each is 32-bits. Each wavefront can have at most 256 registers (VGPRs). To compute the number of wavefronts per CU, take (256/# registers)*4.

For example, a kernel that uses 120 registers (120x32-bit values) can run with eight active wavefronts on each compute unit. Because of the global limits described earlier, each compute unit is limited to 40 wavefronts; thus, kernels can use up to 25 registers (25x32-bit values) without affecting the number of wavefronts/compute unit.

AMD provides the following tools to examine the number of general-purpose registers (GPRs) used by the kernel.

- The AMD APP Profiler displays the number of GPRs used by the kernel.
- Alternatively, the AMD APP Profiler generates the ISA dump (described in Section 4.3, “Analyzing Processor Kernels,” page 4-9), which then can be searched for the string `:NUM_GPRS`.
- The AMD APP KernelAnalyzer also shows the GPR used by the kernel, across a wide variety of GPU compilation targets.

The compiler generates spill code (shuffling values to, and from, memory) if it cannot fit all the live values into registers. Spill code uses long-latency global memory and can have a large impact on performance. Spilled registers can be cached in Southern Island devices, thus reducing the impact on performance. The AMD APP Profiler reports the static number of register spills in the `ScratchReg` field. Generally, it is a good idea to re-write the algorithm to use fewer GPRs, or tune the work-group dimensions specified at launch time to expose more registers/kernel to the compiler, in order to reduce the scratch register usage to 0.

5.6.2.2 Specifying the Default Work-Group Size at Compile-Time

The number of registers used by a work-item is determined when the kernel is compiled. The user later specifies the size of the work-group. Ideally, the OpenCL compiler knows the size of the work-group at compile-time, so it can make optimal register allocation decisions. Without knowing the work-group size, the compiler must assume an upper-bound size to avoid allocating more registers in the work-item than the hardware actually contains.

OpenCL provides a mechanism to specify a work-group size that the compiler can use to optimize the register allocation. In particular, specifying a smaller work-group size at compile time allows the compiler to allocate more registers for each kernel, which can avoid spill code and improve performance. The kernel attribute syntax is:

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

```
__attribute__((reqd_work_group_size(X, Y, Z)))
```

Section 6.7.2 of the OpenCL specification explains the attribute in more detail.

5.6.2.3 Local Memory (LDS) Size

In addition to registers, shared memory can also serve to limit the active wavefronts/compute unit. Each compute unit has 64 kB of LDS, which is shared among all active work-groups. Note that the maximum allocation size is 32 kB. LDS is allocated on a per-work-group granularity, so it is possible (and useful) for multiple wavefronts to share the same local memory allocation. However, large LDS allocations eventually limits the number of workgroups that can be active. Table 5.2 provides more details about how LDS usage can impact the wavefronts/compute unit.

Table 5.2 Effect of LDS Usage on Wavefronts/CU[1]

Local Memory / Work-Group	LDS-Limited Work-Groups	LDS-Limited Wavefronts/ Compute-Unit (Assume 4 Wavefronts/ Work-Group)	LDS-Limited Wavefronts/ Compute-Unit (Assume 3 Wavefronts/ Work-Group)	LDS-Limited Wavefronts/ Compute-Unit (Assume 2 Wavefronts/ Work-Group)
<=4K	16	40	40	32
4.0K-4.2K	15	40	40	30
4.2K-4.5K	14	40	40	28
4.5K-4.9K	13	40	39	26
4.9K-5.3K	12	40	36	24
5.3K-5.8K	11	40	33	22
5.8K-6.4K	10	40	30	20
6.4K-7.1K	9	36	27	18
7.1K-8.0K	8	32	24	16
8.0K-9.1K	7	28	21	14
9.1K-10.6K	6	24	18	12
10.6K-12.8K	5	20	15	10
12.8K-16.0K	4	16	12	8
16.0K-21.3K	3	12	9	6
21.3K-32.0K	2	8	6	4

1. Assumes each work-group uses four wavefronts (the maximum supported by the AMD OpenCL SDK).

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

AMD provides the following tools to examine the amount of LDS used by the kernel:

- The AMD APP Profiler displays the LDS usage. See the `LocalMem` counter.
- Alternatively, use the AMD APP Profiler to generate the ISA dump (described in Section 4.3, "Analyzing Processor Kernels," page 4-9), then search for the string `SQ_LDS_ALLOC:SIZE` in the ISA dump. Note that the value is shown in hexadecimal format.

5.6.3 Partitioning the Work

In OpenCL, each kernel executes on an index point that exists in a global NDRange. The partition of the NDRange can have a significant impact on performance; thus, it is recommended that the developer explicitly specify the global (`#work-groups`) and local (`#work-items/work-group`) dimensions, rather than rely on OpenCL to set these automatically (by setting `local_work_size` to NULL in `clEnqueueNDRangeKernel`). This section explains the guidelines for partitioning at the global, local, and work/kernel levels.

5.6.3.1 Global Work Size

OpenCL does not explicitly limit the number of work-groups that can be submitted with a `clEnqueueNDRangeKernel` command. The hardware limits the available in-flight threads, but the OpenCL SDK automatically partitions a large number of work-groups into smaller pieces that the hardware can process. For some large workloads, the amount of memory available to the GPU can be a limitation; the problem might require so much memory capacity that the GPU cannot hold it all. In these cases, the programmer must partition the workload into multiple `clEnqueueNDRangeKernel` commands. The available device memory can be obtained by querying `clDeviceInfo`.

At a minimum, ensure that the workload contains at least as many work-groups as the number of compute units in the hardware. Work-groups cannot be split across multiple compute units, so if the number of work-groups is less than the available compute units, some units are idle. See Appendix D, "Device Parameters" for a table of device parameters, including the number of compute units, or use `clGetDeviceInfo(…CL_DEVICE_MAX_COMPUTE_UNITS)` to determine the value dynamically.

5.6.3.2 Local Work Size (#Work-Items per Work-Group)

OpenCL limits the number of work-items in each group. Call `clDeviceInfo` with the `CL_DEVICE_MAX_WORK_GROUP_SIZE` to determine the maximum number of work-groups supported by the hardware. Currently, AMD GPUs with SDK 2.1 return 256 as the maximum number of work-items per work-group. Note the number of work-items is the product of all work-group dimensions; for example, a work-group with dimensions 32x16 requires 512 work-items, which is not allowed with the current AMD OpenCL SDK.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

The fundamental unit of work on AMD GPUs is called a wavefront. Each wavefront consists of 64 work-items; thus, the optimal local work size is an integer multiple of 64 (specifically 64, 128, 192, or 256) work-items per work-group.

Work-items in the same work-group can share data through LDS memory and also use high-speed local atomic operations. Thus, larger work-groups enable more work-items to efficiently share data, which can reduce the amount of slower global communication. However, larger work-groups reduce the number of global work-groups, which, for small workloads, could result in idle compute units. Generally, larger work-groups are better as long as the global range is big enough to provide 1-2 Work-Groups for each compute unit in the system; for small workloads it generally works best to reduce the work-group size in order to avoid idle compute units. Note that it is possible to make the decision dynamically, when the kernel is launched, based on the launch dimensions and the target device characteristics.

5.6.3.3 Work-Group Dimensions vs Size

The local NDRange can contain up to three dimensions, here labeled X, Y, and Z. The X dimension is returned by `get_local_id(0)`, Y is returned by `get_local_id(1)`, and Z is returned by `get_local_id(2)`. The GPU hardware schedules the kernels so that the X dimension moves fastest as the work-items are packed into wavefronts. For example, the 128 threads in a 2D work-group of dimension 32x4 (X=32 and Y=4) are packed into two wavefronts as follows (notation shown in X,Y order).

<table>
<tr><td rowspan="4">WaveFront0</td><td>0,0</td><td>1,0</td><td>2,0</td><td>3,0</td><td>4,0</td><td>5,0</td><td>6,0</td><td>7,0</td><td>8,0</td><td>9,0</td><td>10,0</td><td>11,0</td><td>12,0</td><td>13,0</td><td>14,0</td><td>15,0</td></tr>
<tr><td>16,0</td><td>17,0</td><td>18,0</td><td>19,0</td><td>20,0</td><td>21,0</td><td>22,0</td><td>23,0</td><td>24,0</td><td>25,0</td><td>26,0</td><td>27,0</td><td>28,0</td><td>29,0</td><td>30,0</td><td>31,0</td></tr>
<tr><td>0,1</td><td>1,1</td><td>2,1</td><td>3,1</td><td>4,1</td><td>5,1</td><td>6,1</td><td>7,1</td><td>8,1</td><td>9,1</td><td>10,1</td><td>11,1</td><td>12,1</td><td>13,1</td><td>14,1</td><td>15,1</td></tr>
<tr><td>16,1</td><td>17,1</td><td>18,1</td><td>19,1</td><td>20,1</td><td>21,1</td><td>22,1</td><td>23,1</td><td>24,1</td><td>25,1</td><td>26,1</td><td>27,1</td><td>28,1</td><td>29,1</td><td>30,1</td><td>31,1</td></tr>
<tr><td rowspan="4">WaveFront1</td><td>0,2</td><td>1,2</td><td>2,2</td><td>3,2</td><td>4,2</td><td>5,2</td><td>6,2</td><td>7,2</td><td>8,2</td><td>9,2</td><td>10,2</td><td>11,2</td><td>12,2</td><td>13,2</td><td>14,2</td><td>15,2</td></tr>
<tr><td>16,2</td><td>17,2</td><td>18,2</td><td>19,2</td><td>20,2</td><td>21,2</td><td>22,2</td><td>23,2</td><td>24,2</td><td>25,2</td><td>26,2</td><td>27,2</td><td>28,2</td><td>29,2</td><td>30,2</td><td>31,2</td></tr>
<tr><td>0,3</td><td>1,3</td><td>2,3</td><td>3,3</td><td>4,3</td><td>5,3</td><td>6,3</td><td>7,3</td><td>8,3</td><td>9,3</td><td>10,3</td><td>11,3</td><td>12,3</td><td>13,3</td><td>14,3</td><td>15,3</td></tr>
<tr><td>16,3</td><td>17,3</td><td>18,3</td><td>19,3</td><td>20,3</td><td>21,3</td><td>22,3</td><td>23,3</td><td>24,3</td><td>25,3</td><td>26,3</td><td>27,3</td><td>28,3</td><td>29,3</td><td>30,3</td><td>31,3</td></tr>
</table>

The total number of work-items in the work-group is typically the most important parameter to consider, in particular when optimizing to hide latency by increasing wavefronts/compute unit. However, the choice of XYZ dimensions for the same overall work-group size can have the following second-order effects.

- Work-items in the same quarter-wavefront execute on the same cycle in the processing engine. Thus, global memory coalescing and local memory bank conflicts can be impacted by dimension, particularly if the fast-moving X dimension is small. Typically, it is best to choose an X dimension of at least 16, then optimize the memory patterns for a block of 16 work-items which differ by 1 in the X dimension.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

- Work-items in the same wavefront have the same program counter and execute the same instruction on each cycle. The packing order can be important if the kernel contains divergent branches. If possible, pack together work-items that are likely to follow the same direction when control-flow is encountered. For example, consider an image-processing kernel where each work-item processes one pixel, and the control-flow depends on the color of the pixel. It might be more likely that a square of 8x8 pixels is the same color than a 64x1 strip; thus, the 8x8 would see less divergence and higher performance.
- When in doubt, a square 16x16 work-group size is a good start.

5.6.4 Summary of NDRange Optimizations

As shown above, execution range optimization is a complex topic with many interacting variables and which frequently requires some experimentation to determine the optimal values. Some general guidelines are:

- Select the work-group size to be a multiple of 64, so that the wavefronts are fully populated.
- Use a work-group size of 64, and schedule four work-groups per compute unit.
- Latency hiding depends on both the number of wavefronts/compute unit, as well as the execution time for each kernel. Generally, two to eight wavefronts/compute unit is desirable, but this can vary significantly, depending on the complexity of the kernel and the available memory bandwidth. The AMD APP Profiler and associated performance counters can help to select an optimal value.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

5.7 Instruction Selection Optimizations

5.7.1 Instruction Bandwidths

Table 5.3 lists the throughput of instructions for GPUs.

Table 5.3 Instruction Throughput (Operations/Cycle for Each Stream Processor)

		Rate (Operations/Cycle) for each Stream Processor	
	Instruction	**One Quarter-Double-Precision-Speed Devices**	**Full Double-Precision-Speed Devices**
Single Precision FP Rates	SPFP FMA	1/4	4
	SPFP MAD	4	4
	ADD	4	4
	MUL	4	4
	INV	1	1
	RQSRT	1	1
	LOG	1	1
Double Precision FP Rates	FMA	1/4	1
	MAD	1/4	1
	ADD	1/2	2
	MUL	1/4	1
	INV (approx.)	1/4	1
	RQSRT (approx.)	1/4	1
Integer Instruction Rates	MAD	1	1
	ADD	4	4
	MUL	1	1
	Bit-shift	4	4
	Bitwise XOR	4	4
Conversion	Float-to-Int	1	1
	Int-to-Float	1	1
24-Bit Integer Inst Rates	MAD	4	4
	ADD	4	4
	MUL	4	4

Double-precision is supported on all Southern Islands devices at varying rates. The use of single-precision calculation is encouraged, if that precision is acceptable. Single-precision data is also half the size of double-precision, which requires less chip bandwidth and is not as demanding on the cache structures.

Generally, the throughput and latency for 32-bit integer operations is the same as for single-precision floating point operations.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

24-bit integer MULs and MADs have four times the throughput of 32-bit integer multiplies. 24-bit signed and unsigned integers are natively supported on the Southern Islands family of devices. The use of OpenCL built-in functions for `mul24` and `mad24` is encouraged. Note that `mul24` can be useful for array indexing operations.

Packed 16-bit and 8-bit operations are not natively supported; however, in cases where it is known that no overflow will occur, some algorithms may be able to effectively pack 2 to 4 values into the 32-bit registers natively supported by the hardware.

The MAD instruction is an IEEE-compliant multiply followed by an IEEE-compliant add; it has the same accuracy as two separate MUL/ADD operations. No special compiler flags are required for the compiler to convert separate MUL/ADD operations to use the MAD instruction.

Table 5.3 shows the throughput for each stream processing core. To obtain the peak throughput for the whole device, multiply the number of stream cores and the engine clock (see Appendix D, “Device Parameters”). For example, according to Table 5.3, a Tahiti device can perform one double-precision ADD operations/2 cycles in each stream core. An AMD Radeon™ HD 7970 GPU has 2048 Stream Cores and an engine clock of 925 MHz, so the entire GPU has a throughput rate of (.5*2048*925 MHz) = 947 GFlops for double-precision adds.

5.7.2 AMD Media Instructions

AMD provides a set of media instructions for accelerating media processing. Notably, the sum-of-absolute differences (SAD) operation is widely used in motion estimation algorithms. For a brief listing and description of the AMD media operations, see the third bullet in Section A.8, “AMD Vendor-Specific Extensions,” page A-4. For the Southern Islands family of devices, new media instructions have been added; these are available under the `cl_amd_media_ops2` extensions.

5.7.3 Math Libraries

The Southern Islands environment contains new instructions for increasing the previous performance of floating point division, trigonometric range reduction, certain type conversions with double-precision values, floating-point classification, and frexp/ldexp.

OpenCL supports two types of math library operation: `native_function()` and `function()`. Native_functions are generally supported in hardware and can run substantially faster, although at somewhat lower accuracy. The accuracy for the non-native functions is specified in section 7.4 of the *OpenCL Specification*. The accuracy for the native functions is implementation-defined. Developers are encouraged to use the native functions when performance is more important than accuracy.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Compared to previous families of GPUs, the accuracy of certain native functions is increased in the Southern Islands family. We recommend retesting applications where native function accuracy was insufficient on previous GPU devices.

5.7.4 Compiler Optimizations

The OpenCL compiler currently recognizes a few patterns and transforms them into a single instruction. By following these patterns, a developer can generate highly efficient code. The currently accepted patterns are:

- Bitfield extract on signed/unsigned integers.

  `(A >> B) & C ==> [u]bit_extract`

  where

  - B and C are compile time constants,
  - A is a 8/16/32bit integer type, and
  - C is a mask.
- Bitfield insert on signed/unsigned integers

  `((A & B) << C) | ((D & E) << F ==> ubit_insert`

  where

  - B and E have no conflicting bits (`B^E == 0`),
  - B, C, E, and F are compile-time constants, and
  - B and E are masks.
  - The first bit set in B is greater than the number of bits in E plus the first bit set in E, or the first bit set in E is greater than the number of bits in B plus the first bit set in B.
  - If B, C, E, or F are equivalent to the value 0, this optimization is also supported.

5.8 Additional Performance Guidance

This section is a collection of performance tips for GPU compute and AMD-specific optimizations.

5.8.1 Loop Unroll `pragma`

The compiler directive `#pragma unroll` *<unroll-factor>* can be placed immediately prior to a loop as a hint to the compiler to unroll a loop. *<unroll-factor>* must be a positive integer, 1 or greater. When *<unroll-factor>* is 1, loop unrolling is disabled. When *<unroll-factor>* is 2 or greater, the compiler uses this as a hint for the number of times the loop is to be unrolled.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Examples for using this loop follow.

No unrolling example:

```
#pragma unroll 1
for (int i = 0; i < n; i++) {
...
}
```

Partial unrolling example:

```
#pragma unroll 4
for (int i = 0; i < 128; i++) {
...
}
```

Currently, the unroll pragma requires that the loop boundaries can be determined at compile time. Both loop bounds must be known at compile time. If *n* is not given, it is equivalent to the number of iterations of the loop when both loop bounds are known. If the unroll-factor is not specified, and the compiler can determine the loop count, the compiler fully unrolls the loop. If the unroll-factor is not specified, and the compiler cannot determine the loop count, the compiler does no unrolling.

5.8.2 Memory Tiling

There are many possible physical memory layouts for images. AMD Accelerated Parallel Processing devices can access memory in a tiled or in a linear arrangement.

- Linear – A linear layout format arranges the data linearly in memory such that element addresses are sequential. This is the layout that is familiar to CPU programmers. This format must be used for OpenCL buffers; it can be used for images.
- Tiled – A tiled layout format has a pre-defined sequence of element blocks arranged in sequential memory addresses (see Figure 5.4 for a conceptual illustration). A microtile consists of ABIJ; a macrotile consists of the top-left 16 squares for which the arrows are red. Only images can use this format. Translating from user address space to the tiled arrangement is transparent to the user. Tiled memory layouts provide an optimized memory access pattern to make more efficient use of the RAM attached to the GPU compute device. This can contribute to lower latency.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

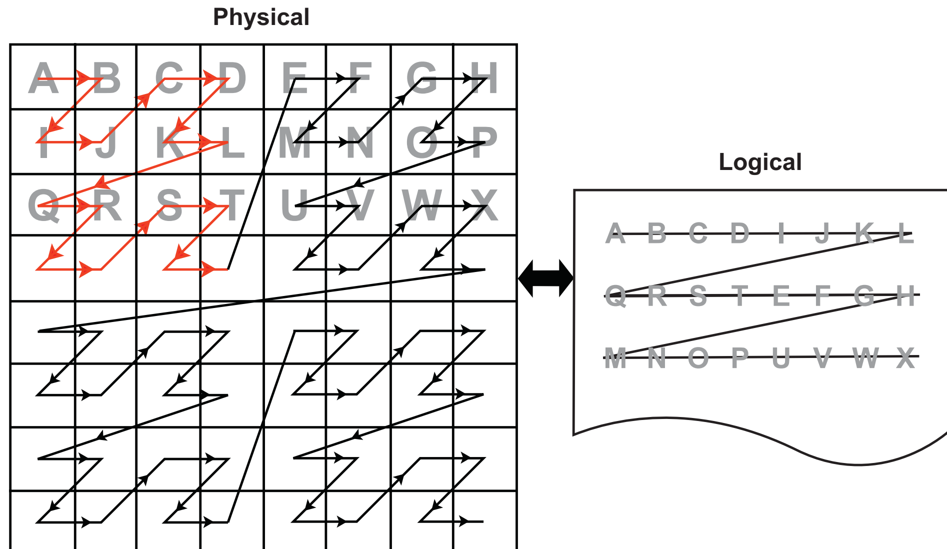

Figure 5.4 One Example of a Tiled Layout Format

Memory Access Pattern –

Memory access patterns in compute kernels are usually different from those in the pixel shaders. Whereas the access pattern for pixel shaders is in a hierarchical, space-filling curve pattern and is tuned for tiled memory performance (generally for textures), the access pattern for a compute kernel is linear across each row before moving to the next row in the global id space. This has an effect on performance, since pixel shaders have implicit blocking, and compute kernels do not. If accessing a tiled image, best performance is achieved if the application tries to use workgroups with 16x16 (or 8x8) work-items.

5.8.3 General Tips

- Avoid declaring global arrays on the kernel's stack frame as these typically cannot be allocated in registers and require expensive global memory operations.
- Use predication rather than control-flow. The predication allows the GPU to execute both paths of execution in parallel, which can be faster than attempting to minimize the work through clever control-flow. The reason for this is that if no memory operation exists in a `?:` operator (also called a ternary operator), this operation is translated into a single `cmov_logical` instruction, which is executed in a single cycle. An example of this is:

```
If (A>B) {
 C += D;
} else {
 C -= D;
}
```

Replace this with:

```
int factor = (A>B) ? 1:-1;
C += factor*D;
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

In the first block of code, this translates into an IF/ELSE/ENDIF sequence of conditional code, each taking ~8 cycles. If divergent, this code executes in ~36 clocks; otherwise, in ~28 clocks. A branch not taken costs four cycles (one instruction slot); a branch taken adds four slots of latency to fetch instructions from the instruction cache, for a total of 16 clocks. Since the execution mask is saved, then modified, then restored for the branch, ~12 clocks are added when divergent, ~8 clocks when not.

In the second block of code, the `?:` operator executes in the vector units, so no extra CF instructions are generated. Since the instructions are sequentially dependent, this block of code executes in 12 cycles, for a 1.3x speed improvement. To see this, the first cycle is the (A>B) comparison, the result of which is input to the second cycle, which is the `cmov_logical` factor, bool, 1, -1. The final cycle is a MAD instruction that: mad C, factor, D, C. If the ratio between conditional code and ALU instructions is low, this is a good pattern to remove the control flow.

- Loop Unrolling
  - OpenCL kernels typically are high instruction-per-clock applications. Thus, the overhead to evaluate control-flow and execute branch instructions can consume a significant part of resource that otherwise can be used for high-throughput compute operations.
  - The AMD Accelerated Parallel Processing OpenCL compiler performs simple loop unrolling optimizations; however, for more complex loop unrolling, it may be beneficial to do this manually.
- If possible, create a reduced-size version of your data set for easier debugging and faster turn-around on performance experimentation. GPUs do not have automatic caching mechanisms and typically scale well as resources are added. In many cases, performance optimization for the reduced-size data implementation also benefits the full-size algorithm.
- When tuning an algorithm, it is often beneficial to code a simple but accurate algorithm that is retained and used for functional comparison. GPU tuning can be an iterative process, so success requires frequent experimentation, verification, and performance measurement.
- The profiling and analysis tools report statistics on a per-kernel granularity. To narrow the problem further, it might be useful to remove or comment-out sections of code, then re-run the timing and profiling tool.
- Avoid writing code with dynamic pointer assignment on the GPU. For example:

```
kernel void dyn_assign(global int* a, global int* b, global int* c)
{
        global int* d;
        size_t idx = get_global_id(0);
        if (idx & 1) {
                d = b;
        } else {
                d = c;
        }
        a[idx] = d[idx];
}
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

This is inefficient because the GPU compiler must know the base pointer that every load comes from and in this situation, the compiler cannot determine what 'd' points to. So, both B and C are assigned to the same GPU resource, removing the ability to do certain optimizations.

- If the algorithm allows changing the work-group size, it is possible to get better performance by using larger work-groups (more work-items in each work-group) because the workgroup creation overhead is reduced. On the other hand, the OpenCL CPU runtime uses a task-stealing algorithm at the work-group level, so when the kernel execution time differs because it contains conditions and/or loops of varying number of iterations, it might be better to increase the number of work-groups. This gives the runtime more flexibility in scheduling work-groups to idle CPU cores. Experimentation might be needed to reach optimal work-group size.
- Since the AMD OpenCL runtime supports only in-order queuing, using `clFinish()` on a queue and queuing a blocking command gives the same result. The latter saves the overhead of another API command.

  For example:

```
clEnqueueWriteBuffer(myCQ, buff, CL_FALSE, 0, buffSize, input, 0, NULL, 
NULL);
clFinish(myCQ);
```

  is equivalent, for the AMD OpenCL runtime, to:

```
clEnqueueWriteBuffer(myCQ, buff, CL_TRUE, 0, buffSize, input, 0, NULL, 
NULL);
```

5.8.4 Guidance for CUDA Programmers Using OpenCL

- Porting from CUDA to OpenCL is relatively straightforward. Multiple vendors have documents describing how to do this, including AMD:

http://developer.amd.com/documentation/articles/pages/OpenCL-and-the-ATI-Stream-v2.0-Beta.aspx#four

- Some specific performance recommendations which differ from other GPU architectures:
  - Use a workgroup size that is a multiple of 64. CUDA code can use a workgroup size of 32; this uses only half the available compute resources on an AMD Radeon™ HD 7970 GPU.
  - AMD GPUs have a very high single-precision flops capability (3.788 teraflops in a single AMD Radeon™ HD 7970 GPU). Algorithms that benefit from such throughput can deliver excellent performance on AMD Accelerated Parallel Processing hardware.

5.8.5 Guidance for CPU Programmers Using OpenCL to Program GPUs

OpenCL is the industry-standard toolchain for programming GPUs and parallel devices from many vendors. It is expected that many programmers skilled in CPU programming will program GPUs for the first time using OpenCL. This section provides some guidance for experienced programmers who are

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

programming a GPU for the first time. It specifically highlights the key differences in optimization strategy.

- Study the local memory (LDS) optimizations. These greatly affect the GPU performance. Note the difference in the organization of local memory on the GPU as compared to the CPU cache. Local memory is shared by many work-items (64 on Tahiti). This contrasts with a CPU cache that normally is dedicated to a single work-item. GPU kernels run well when they collaboratively load the shared memory.
- GPUs have a large amount of raw compute horsepower, compared to memory bandwidth and to "control flow" bandwidth. This leads to some high-level differences in GPU programming strategy.
  - A CPU-optimized algorithm may test branching conditions to minimize the workload. On a GPU, it is frequently faster simply to execute the workload.
  - A CPU-optimized version can use memory to store and later load pre-computed values. On a GPU, it frequently is faster to recompute values rather than saving them in registers. Per-thread registers are a scarce resource on the CPU; in contrast, GPUs have many available per-thread register resources.
- Use `float4` and the OpenCL built-ins for vector types (`vload`, `vstore`, etc.). These enable the AMD Accelerated Parallel Processing OpenCL implementation to generate efficient, packed SSE instructions when running on the CPU. Vectorization is an optimization that benefits both the AMD CPU and GPU.

5.8.6 Optimizing Kernel Code

5.8.6.1 Using Vector Data Types

The CPU contains a vector unit, which can be efficiently used if the developer is writing the code using vector data types.

For architectures before Bulldozer, the instruction set is called SSE, and the vector width is 128 bits. For Bulldozer, there the instruction set is called AVX, for which the vector width is increased to 256 bits.

Using four-wide vector types (int4, float4, etc.) is preferred, even with Bulldozer.

5.8.6.2 Local Memory

The CPU does not benefit much from local memory; sometimes it is detrimental to performance. As local memory is emulated on the CPU by using the caches, accessing local memory and global memory are the same speed, assuming the information from the global memory is in the cache.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

5.8.6.3 Using Special CPU Instructions

The Bulldozer family of CPUs supports FMA4 instructions, exchanging instructions of the form `a*b+c` with `fma(a,b,c)` or `mad(a,b,c)` allows for the use of the special hardware instructions for multiplying and adding.

There also is hardware support for OpenCL functions that give the new hardware implementation of rotating.

For example:

```
sum.x += tempA0.x * tempB0.x + tempA0.y * tempB1.x + tempA0.z * tempB2.x +
tempA0.w * tempB3.x;
```

can be written as a composition of mad instructions which use fused multiple add (FMA):

```
sum.x += mad(tempA0.x, tempB0.x, mad(tempA0.y, tempB1.x, mad(tempA0.z,
tempB2.x, tempA0.w*tempB3.x)));
```

5.8.6.4 Avoid Barriers When Possible

Using barriers in a kernel on the CPU causes a significant performance penalty compared to the same kernel without barriers. Use a barrier only if the kernel requires it for correctness, and consider changing the algorithm to reduce barriers usage.

5.8.7 Optimizing Kernels for Southern Island GPUs

5.8.7.1 Remove Conditional Assignments

A conditional of the form “if-then-else” generates branching. Use the `select()` function to replace these structures with conditional assignments that do not cause branching. For example:

```
if(x==1) r=0.5;
if(x==2) r=1.0;
```

becomes

```
r = select(r, 0.5, x==1);
r = select(r, 1.0, x==2);
```

Note that if the body of the `if` statement contains an I/O, the `if` statement cannot be eliminated.

5.8.7.2 Bypass Short-Circuiting

A conditional expression with many terms can compile into nested conditional code due to the C-language requirement that expressions must short circuit. To prevent this, move the expression out of the control flow statement. For example:

```
if(a&&b&&c&&d){…}
```

becomes

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

```
bool cond = a&&b&&c&&d;
if(cond){…}
```

The same applies to conditional expressions used in loop constructs (`do`, `while`, `for`).

5.8.7.3 Unroll Small Loops

If the loop bounds are known, and the loop is small (less than 16 or 32 instructions), unrolling the loop usually increases performance.

5.8.7.4 Avoid Nested `if`s

Because the GPU is a Vector ALU architecture, there is a cost to executing an if-then-else block because both sides of the branch are evaluated, then one result is retained while the other is discarded. When `if` blocks are nested, the results are twice as bad; in general, if blocks are nested k levels deep, 2^k nested conditional structures are generated. In this situation, restructure the code to eliminate nesting.

5.8.7.5 Experiment With `do`/`while`/`for` Loops

`for` loops can generate more conditional code than equivalent `do` or `while` loops. Experiment with these different loop types to find the one with best performance.

5.9 Specific Guidelines for Southern Islands GPUs

The AMD Southern Islands (SI) family of products is quite different from previous generations. These are referred to as SI chips and are based on what is publicly called Graphics Core Next.

SI compute units are much different than those of previous chips. With previous generations, a compute unit (Vector ALU) was VLIW in nature, so four (Cayman GPUs) or five (all other Evergreen/Northern Islands GPUs) instructions could be packed into a single ALU instruction slot (called a bundle). It was not always easy to schedule instructions to fill all of these slots, so achieving peak ALU utilization was a challenge.

With SI GPUs, the compute units are now scalar; however, there now are four Vector ALUs per compute unit. Each Vector ALU requires at least one wavefront scheduled to it to achieve peak ALU utilization.

Along with the four Vector ALUs within a compute unit, there is also a scalar unit. The scalar unit is used to handle branching instructions, constant cache accesses, and other operations that occur per wavefront. The advantage to having a scalar unit for each compute unit is that there are no longer large penalties for branching, aside from thread divergence.

The instruction set for SI is scalar, as are GPRs. Also, the instruction set is no longer clause-based. There are two types of GPRs: scalar GPRs (SGPRs) and vector GPRs (VGPRs). Each Vector ALU has its own SGPR and VGPR pool. There are 512 SGPRs and 256 VGPRs per Vector ALU. VGPRs handle all vector

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

instructions (any instruction that is handled per thread, such as `v_add_f32`, a floating point add). SGPRs are used for scalar instructions: any instruction that is executed once per wavefront, such as a branch, a scalar ALU instruction, and constant cache fetches. (SGPRs are also used for constants, all buffer/texture definitions, and sampler definitions; some kernel arguments are stored, at least temporarily, in SGPRs.) SGPR allocation is in increments of eight, and VGPR allocation is in increments of four. These increments also represent the minimum allocation size of these resources.

Typical vector instructions execute in four cycles; typical scalar ALU instructions in one cycle. This allows each compute unit to execute one Vector ALU and one scalar ALU instruction every four clocks (each compute unit is offset by one cycle from the previous one).

All Southern Islands GPUs have double-precision support. For Tahiti (AMD Radeon™ HD 79XX series), double precision adds run at one-half the single precision add rate. Double-precision multiplies and MAD instructions run at one-quarter the floating-point rate.

The double-precision rate of Pitcairn (AMD Radeon™ HD 78XX series) and Cape Verde (AMD Radeon™ HD 77XX series) is one quarter that of Tahiti. This also affects the performance of single-precision fused multiple add (FMA).

Similar to previous generations local data share (LDS) is a shared resource within a compute unit. The maximum LDS allocation size for a work-group is still 32 kB, however each compute unit has a total of 64 kB of LDS. On SI GPUs, LDS memory has 32 banks; thus, it is important to be aware of LDS bank conflicts on half-wavefront boundaries. The allocation granularity for LDS is 256 bytes; the minimum size is 0 bytes. It is much easier to achieve high LDS bandwidth use on SI hardware.

L1 cache is still shared within a compute unit. The size has now increased to 16 kB per compute unit for all SI GPUs. The caches now are read/write, so sharing data between work-items in a work-group (for example, when LDS does not suffice) is much faster.

It is possible to schedule a maximum of 10 wavefronts per compute unit, assuming there are no limitations by other resources, such as registers or local memory; but there is a limit of 16 work-groups per compute unit if the work-groups are larger than a single wavefront. If the dispatch is larger than what can fit at once on the GPU, the GPU schedules new work-groups as others finish.

Since there are no more clauses in the SI instruction set architecture (ISA), the compiler inserts “wait” commands to indicate that the compute unit needs the results of a memory operation before proceeding. If the scalar unit determines that a wait is required (the data is not yet ready), the Vector ALU can switch to another wavefront. There are different types of wait commands, depending on the memory access.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Notes –

- Vectorization is no longer needed, nor desirable; in fact, it can affect performance because it requires a greater number of VGPRs for storage. I is recommended not to combine work-items.
- Register spilling is no greater a problem with four wavefronts per work-group than it is with one wavefront per work-group. This is because each wavefront has the same number of SGPRs and VGPRs available in either case.
- Read coalescing does not work for 64-bit data sizes. This means reads for float2, int2, and double might be slower than expected.
- Work-groups with 256 work-items can be used to ensure that each compute unit is being used. Barriers now are much faster.
- The engine is wider than previous generations; this means larger dispatches are required to keep the all the compute units busy.
- A single wavefront can take twice as long to execute compared to previous generations (assuming ALU bound). This is because GPUs with VLIW-4 could execute the four instructions in a VLIW bundle in eight clocks (typical), and SI GPUs can execute one vector instruction in four clocks (typical).
- Execution of kernel dispatches can overlap if there are no dependencies between them and if there are resources available in the GPU. This is critical when writing benchmarks it is important that the measurements are accurate and that “false dependencies” do not cause unnecessary slowdowns.

  An example of false dependency is:

  a. Application creates a kernel “foo”.

  b. Application creates input and output buffers.

  c. Application binds input and output buffers to kernel “foo”.

  d. Application repeatedly dispatches “foo” with the same parameters.

If the output data is the same each time, then this is a false dependency because there is no reason to stall concurrent execution of dispatches. To avoid stalls, use multiple output buffers. The number of buffers required to get peak performance depends on the kernel.

Table 5.4 compares the resource limits for Northern Islands and Southern Islands GPUs.

Table 5.4 Resource Limits for Northern Islands and Southern Islands

	VLIW Width	VGPRs	SGPRs	LDS Size	LDS Max Alloc	L1$/CU	L2$/Channel
Northern Islands	4	256 (128-bit)	-	32 kB	32 kB	8 kB	64 kB
Southern Islands	1	256 (32-bit)	512	64 kB	32 kB	16 kB	64 kB

Table 5.5 provides a simplified picture showing the Northern Island compute unit arrangement.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

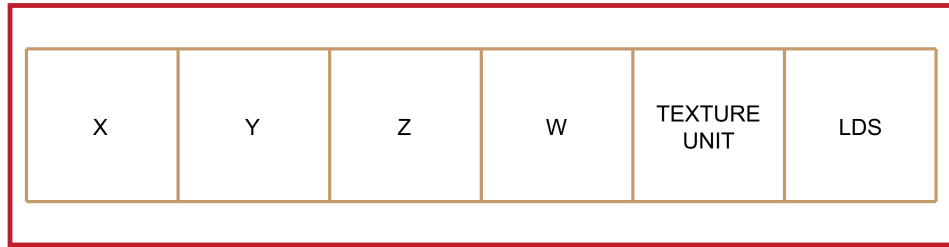

Figure 5.5 Northern Islands Compute Unit Arrangement

Table 5.6 provides a simplified picture showing the Southern Island compute unit arrangement.

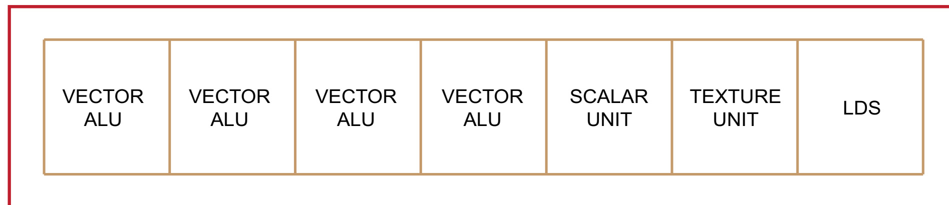

Figure 5.6 Southern Island Compute Unit Arrangement

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Chapter 6 OpenCL Performance and Optimization for Evergreen and Northern Islands Devices

This chapter discusses performance and optimization when programming for AMD Accelerated Parallel Processing GPU compute devices that are part of the Southern Islands family, as well as CPUs and multiple devices. Details specific to the Evergreen and Northern Islands families of GPUs are provided in Chapter 5, "OpenCL Performance and Optimization for Southern Islands Devices."

6.1 Global Memory Optimization

Figure 6.1 is a block diagram of the GPU memory system. The up arrows are read paths, the down arrows are write paths. WC is the write combine cache.

The GPU consists of multiple compute units. Each compute unit contains 32 kB local (on-chip) memory, L1 cache, registers, and 16 processing element (pe). Each processing element contains a five-way (or four-way, depending on the GPU type) VLIW processor. Individual work-items execute on a single processing element; one or more work-groups execute on a single compute unit. On a GPU, hardware schedules the work-items. On the ATI Radeon™ HD 5000 series of GPUs, hardware schedules groups of work-items, called wavefronts, onto stream cores; thus, work-items within a wavefront execute in lock-step; the same instruction is executed on different data.

The L1 cache is 8 kB per compute unit. (For the ATI Radeon™ HD 5870 GPU, this means 160 kB for the 20 compute units.) The L1 cache bandwidth on the ATI Radeon™ HD 5870 GPU is one terabyte per second:

L1 Bandwidth = Compute Units * Wavefront Size/Compute Unit * EngineClock

Multiple compute units share L2 caches. The L2 cache size on the ATI Radeon™ HD 5870 GPUs is 512 kB:

L2 Cache Size = Number or channels * L2 per Channel

The bandwidth between L1 caches and the shared L2 cache is 435 GB/s:

L2 Bandwidth = Number of channels * Wavefront Size * Engine Clock

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

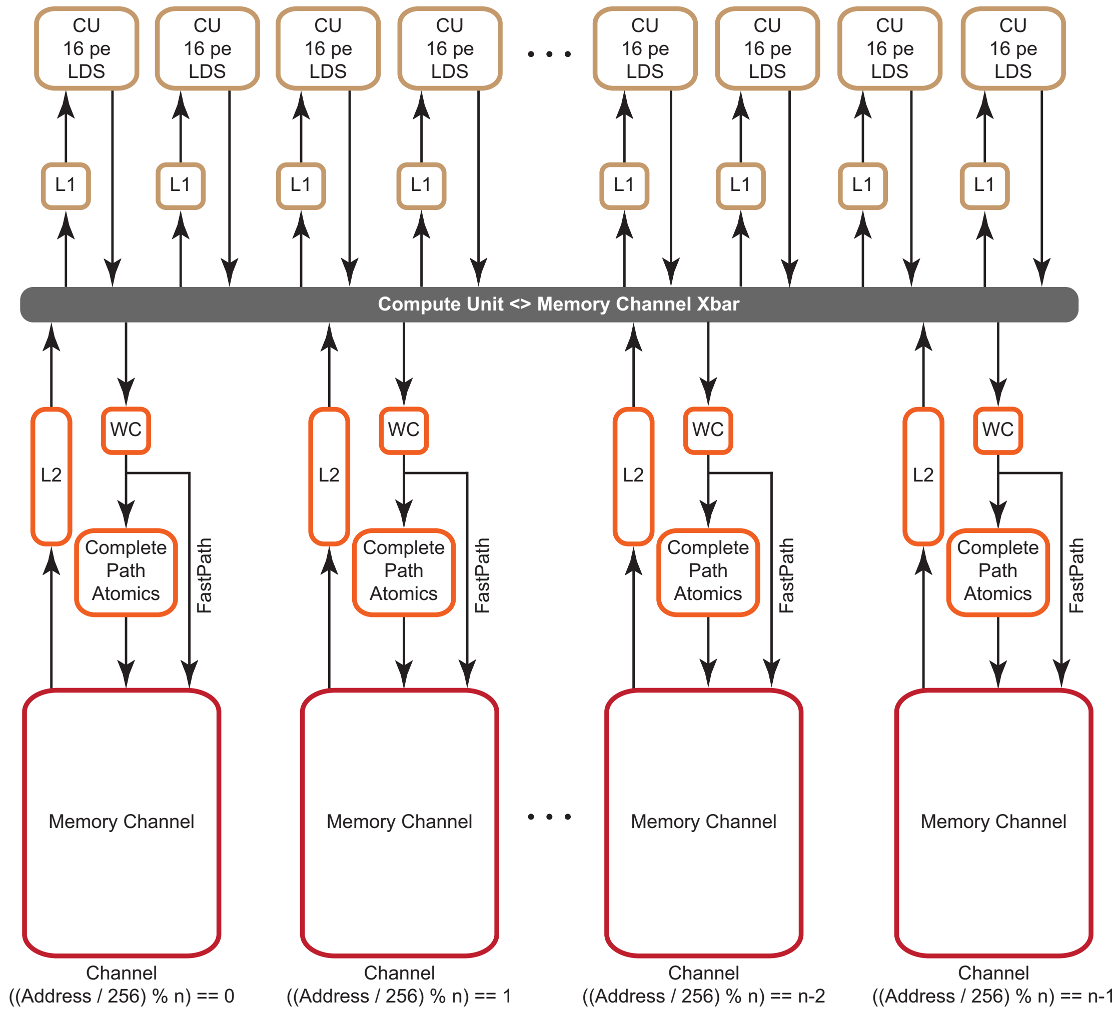

Figure 6.1 Memory System

The ATI Radeon™ HD 5870 GPU has eight memory controllers (“Memory Channel” in Figure 6.1). The memory controllers are connected to multiple banks of memory. The memory is GDDR5, with a clock speed of 1200 MHz and a data rate of 4800 Mb/pin. Each channel is 32-bits wide, so the peak bandwidth for the ATI Radeon™ HD 5870 GPU is:

(8 memory controllers) * (4800 Mb/pin) * (32 bits) * (1 B/8b) = 154 GB/s

The peak memory bandwidth of your device is available in Appendix D, “Device Parameters.”

If two memory access requests are directed to the same controller, the hardware serializes the access. This is called a *channel conflict*. Similarly, if two memory access requests go to the same memory bank, hardware serializes the access. This is called a *bank conflict*. From a developer’s point of view, there is not much difference between channel and bank conflicts. A large power of two stride results in a channel conflict; a larger power of two stride results in a bank conflict. The size of the power of two stride that causes a specific type of conflict depends

on the chip. A stride that results in a channel conflict on a machine with eight channels might result in a bank conflict on a machine with four.

In this document, the term bank conflict is used to refer to either kind of conflict.

6.1.1 Two Memory Paths

ATI Radeon™ HD 5000 series graphics processors have two, independent memory paths between the compute units and the memory:

- FastPath performs only basic operations, such as loads and stores (data sizes must be a multiple of 32 bits). This often is faster and preferred when there are no advanced operations.
- CompletePath, supports additional advanced operations, including atomics and sub-32-bit (byte/short) data transfers.

6.1.1.1 Performance Impact of FastPath and CompletePath

There is a large difference in performance on ATI Radeon™ HD 5000 series hardware between FastPath and CompletePath. Figure 6.2 shows two kernels (one FastPath, the other CompletePath) and the delivered DRAM bandwidth for each kernel on the ATI Radeon™ HD 5870 GPU. Note that an atomic add forces CompletePath.

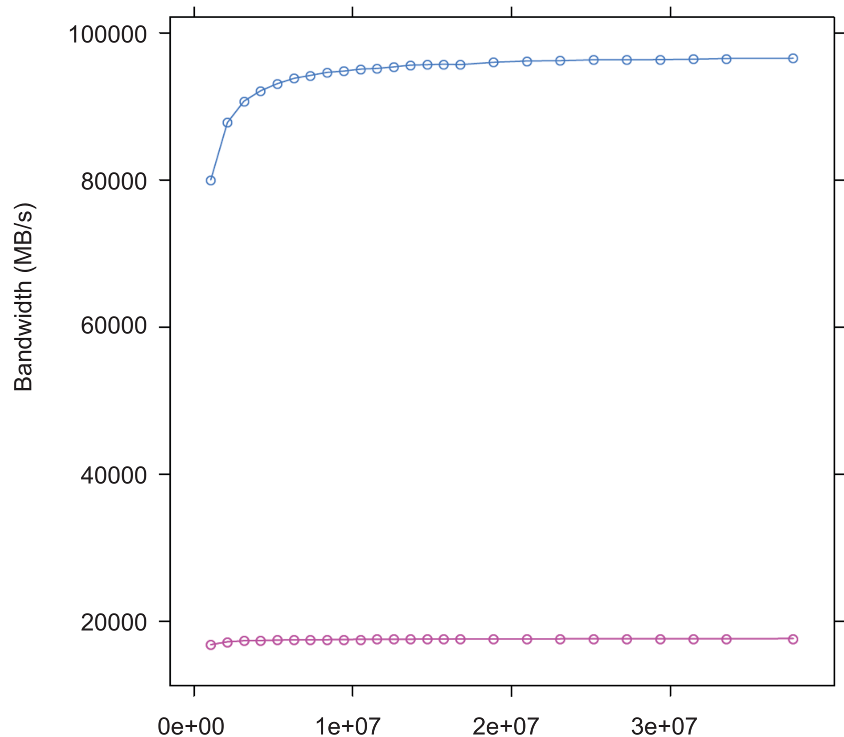

Figure 6.2 FastPath (blue) vs CompletePath (red) Using float1

The kernel code follows. Note that the atomic extension must be enabled under OpenCL 1.0.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

```
__kernel void
  CopyFastPath(__global const float * input,
               __global float * output)
{
    int gid = get_global_id(0);
    output[gid] = input[gid];
    return ;
}
__kernel void
  CopyComplete(__global const float * input, __global float* output)
{
    int gid = get_global_id(0);
    if (gid <0){
      atom_add((__global int *) output,1);
      }
    output[gid] = input[gid];
    return ;
}
```

Table 6.1 lists the effective bandwidth and ratio to maximum bandwidth.

Table 6.1 Bandwidths for 1D Copies

Kernel	Effective Bandwidth	Ratio to Peak Bandwidth
copy 32-bit 1D FP	96 GB/s	63%
copy 32-bit 1D CP	18 GB/s	12%

The difference in performance between FastPath and CompletePath is significant. If your kernel uses CompletePath, consider if there is another way to approach the problem that uses FastPath. OpenCL read-only images always use FastPath.

6.1.1.2 Determining The Used Path

Since the path selection is done automatically by the OpenCL compiler, your kernel may be assigned to CompletePath. This section explains the strategy the compiler uses, and how to find out what path was used.

The compiler is conservative when it selects memory paths. The compiler often maps all user data into a single unordered access view (UAV),[1] so a single atomic operation (even one that is not executed) may force all loads and stores to use CompletePath.

The effective bandwidth listing above shows two OpenCL kernels and the associated performance. The first kernel uses the FastPath while the second uses the CompletePath. The second kernel is forced to CompletePath because in CopyComplete, the compiler noticed the use of an atomic.

There are two ways to find out which path is used. The first method uses the AMD APP Profiler, which provides the following three performance counters for this purpose:

1. UAVs allow compute shaders to store results in (or write results to) a buffer at any arbitrary location. On DX11 hardware, UAVs can be created from buffers and textures. On DX10 hardware, UAVs cannot be created from typed resources (textures). This is the same as a random access target (RAT).

1. FastPath counter: The total bytes written through the FastPath (no atomics, 32-bit types only).
2. CompletePath counter: The total bytes read and written through the CompletePath (supports atomics and non-32-bit types).
3. PathUtilization counter: The percentage of bytes read and written through the FastPath or CompletePath compared to the total number of bytes transferred over the bus.

The second method is static and lets you determine the path by looking at a machine-level ISA listing (using the AMD APP KernelAnalyzer in OpenCL).

```
MEM_RAT_CACHELESS -> FastPath
MEM_RAT -> CompPath
MEM_RAT_NOP_RTN -> Comp_load
```

FastPath operations appear in the listing as:

```
...
TEX: ...
... VFETCH ...
... MEM_RAT_CACHELESS_STORE_RAW: ...
...
```

The `vfetch` Instruction is a load type that in graphics terms is called a vertex fetch (the group control TEX indicates that the load uses the L1 cache.)

The instruction `MEM_RAT_CACHELESS` indicates that FastPath operations are used.

Loads in CompletePath are a split-phase operation. In the first phase, hardware copies the old value of a memory location into a special buffer. This is done by performing atomic operations on the memory location. After the value has reached the buffer, a normal load is used to read the value. Note that RAT stands for random access target, which is the same as an unordered access view (UAV); it allows, on DX11 hardware, writes to, and reads from, any arbitrary location in a buffer.

The listing shows:

```
.. MEM_RAT_NOP_RTN_ACK: RAT(1)
.. WAIT_ACK: Outstanding_acks <= 0
.. TEX: ADDR(64) CNT(1)
.. VFETCH ...
```

The instruction sequence means the following:

`MEM_RAT` Read into a buffer using CompletePath, do no operation on the memory location, and send an ACK when done.

`WAIT_ACK` Suspend execution of the wavefront until the ACK is received. If there is other work pending this might be free, but if there is no other work to be done this could take 100's of cycles.

`TEX` Use the L1 cache for the next instruction.

`VFETCH` Do a load instruction to (finally) get the value.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Stores appear as:

```
.. MEM_RAT_STORE_RAW: RAT(1)
```

The instruction MEM_RAT_STORE is the store along the CompletePath.

MEM_RAT means CompletePath; MEM_RAT_CACHELESS means FastPath.

6.1.2 Channel Conflicts

The important concept is memory stride: the increment in memory address, measured in elements, between successive elements fetched or stored by consecutive work-items in a kernel. Many important kernels do not exclusively use simple stride one accessing patterns; instead, they feature large non-unit strides. For instance, many codes perform similar operations on each dimension of a two- or three-dimensional array. Performing computations on the low dimension can often be done with unit stride, but the strides of the computations in the other dimensions are typically large values. This can result in significantly degraded performance when the codes are ported unchanged to GPU systems. A CPU with caches presents the same problem, large power-of-two strides force data into only a few cache lines.

One solution is to rewrite the code to employ array transpositions between the kernels. This allows all computations to be done at unit stride. Ensure that the time required for the transposition is relatively small compared to the time to perform the kernel calculation.

For many kernels, the reduction in performance is sufficiently large that it is worthwhile to try to understand and solve this problem.

In GPU programming, it is best to have adjacent work-items read or write adjacent memory addresses. This is one way to avoid channel conflicts.

When the application has complete control of the access pattern and address generation, the developer must arrange the data structures to minimize bank conflicts. Accesses that differ in the lower bits can run in parallel; those that differ only in the upper bits can be serialized.

In this example:

```
for (ptr=base; ptr<max; ptr += 16KB)
     R0 = *ptr ;
```

where the lower bits are all the same, the memory requests all access the same bank on the same channel and are processed serially.

This is a low-performance pattern to be avoided. When the stride is a power of 2 (and larger than the channel interleave), the loop above only accesses one channel of memory.

The hardware byte address bits are:

31:x	bank	channel	7:0 address

- On all ATI Radeon™ HD 5000-series GPUs, the lower eight bits select an element within a channel.
- The next set of bits select the channel. The number of channel bits varies, since the number of channels is not the same on all parts. With eight channels, three bits are used to select the channel; with two channels, a single bit is used.
- The next set of bits selects the memory bank. The number of bits used depends on the number of memory banks.
- The remaining bits are the rest of the address.

On the ATI Radeon™ HD 5870 GPU, the channel selection are bits 10:8 of the byte address. This means a linear burst switches channels every 256 bytes. Since the wavefront size is 64, channel conflicts are avoided if each work-item in a wave reads a different address from a 64-word region. All ATI Radeon™ HD 5000 series GPUs have the same layout: channel ends at bit 8, and the memory bank is to the left of the channel.

A burst of 2 kB (8 * 256 bytes) cycles through all the channels.

When calculating an address as y*width+x, but reading a burst on a column (incrementing y), only one memory channel of the system is used, since the width is likely a multiple of 256 words = 2048 bytes. If the width is an odd multiple of 256B, then it cycles through all channels.

Similarly, the bank selection bits on the ATI Radeon™ HD 5870 GPU are bits 14:11, so the bank switches every 2 kB. A linear burst of 32 kB cycles through all banks and channels of the system. If accessing a 2D surface along a column, with a y*width+x calculation, and the width is some multiple of 2 kB dwords (32 kB), then only 1 bank and 1 channel are accessed of the 16 banks and 8 channels available on this GPU.

All ATI Radeon™ HD 5000-series GPUs have an interleave of 256 bytes (64 dwords).

If every work-item in a work-group references consecutive memory addresses and the address of work-item 0 is aligned to 256 bytes and each work-item fetches 32 bits, the entire wavefront accesses one channel. Although this seems slow, it actually is a fast pattern because it is necessary to consider the memory access over the entire device, not just a single wavefront.

One or more work-groups execute on each compute unit. On the ATI Radeon™ HD 5000-series GPUs, work-groups are dispatched in a linear order, with x changing most rapidly. For a single dimension, this is:

DispatchOrder = `get_group_id(0)`

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

For two dimensions, this is:

DispatchOrder = `get_group_id(0) + get_group_id(1) * get_num_groups(0)`

This is row-major-ordering of the blocks in the index space. Once all compute units are in use, additional work-groups are assigned to compute units as needed. Work-groups retire in order, so active work-groups are contiguous.

At any time, each compute unit is executing an instruction from a single wavefront. In memory intensive kernels, it is likely that the instruction is a memory access. Since there are eight channels on the ATI Radeon™ HD 5870 GPU, at most eight of the compute units can issue a memory access operation in one cycle. It is most efficient if the accesses from eight wavefronts go to different channels. One way to achieve this is for each wavefront to access consecutive groups of 256 = 64 * 4 bytes.

An inefficient access pattern is if each wavefront accesses all the channels. This is likely to happen if consecutive work-items access data that has a large power of two strides.

In the next example of a kernel for copying, the input and output buffers are interpreted as though they were 2D, and the work-group size is organized as 2D.

The kernel code is:

```
#define WIDTH 1024
#define DATA_TYPE float
#define A(y , x ) A[ (y) * WIDTH + (x ) ]
#define C(y , x ) C[ (y) * WIDTH+(x ) ]
kernel void copy_float (__global const
                        DATA_TYPE * A,
                        __global DATA_TYPE* C)
{
  int idx = get_global_id(0);
  int idy = get_global_id(1);
  C(idy, idx) = A( idy, idx);
}
```

By changing the width, the data type and the work-group dimensions, we get a set of kernels out of this code.

Given a 64x1 work-group size, each work-item reads a consecutive 32-bit address. Given a 1x64 work-group size, each work-item reads a value separated by the width in a power of two bytes.

Table 6.2 shows how much the launch dimension can affect performance. It lists each kernel's effective bandwidth and ratio to maximum bandwidth.

Table 6.2 Bandwidths for Different Launch Dimensions

Kernel	Effective Bandwidth	Ratio to Peak Bandwidth
copy 32-bit 1D FP	96 GB/s	63%

Kernel	Effective Bandwidth	Ratio to Peak Bandwidth
copy 32-bit 1D CP	18 GB/s	12%
copy 32-bit 2D	.3 - 93 GB/s	0 - 60%
copy 128-bit 2D	7 - 122 GB/s	5 - 80%

To avoid power of two strides:

- Add an extra column to the data matrix.
- Change the work-group size so that it is not a power of 2[1].
- It is best to use a width that causes a rotation through all of the memory channels, instead of using the same one repeatedly.
- Change the kernel to access the matrix with a staggered offset.

6.1.2.1 Staggered Offsets

Staggered offsets apply a coordinate transformation to the kernel so that the data is processed in a different order. Unlike adding a column, this technique does not use extra space. It is also relatively simple to add to existing code.

Figure 6.3 illustrates the transformation to staggered offsets.

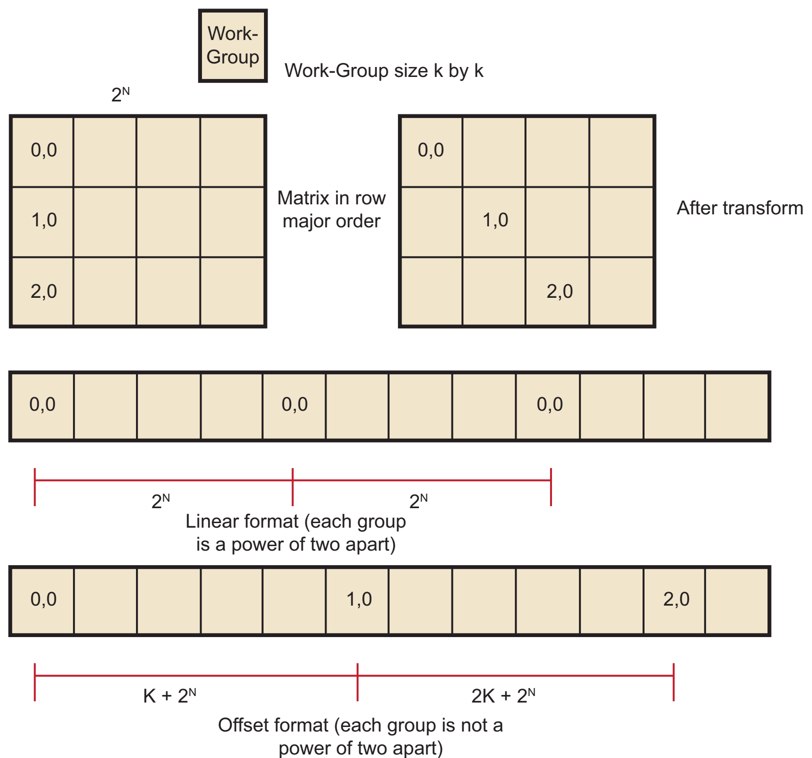

Figure 6.3 Transformation to Staggered Offsets

1. Generally, it is not a good idea to make the work-group size something other than an integer multiple of the wavefront size, but that usually is less important than avoiding channel conflicts.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

The global ID values reflect the order that the hardware initiates work-groups. The values of get group ID are in ascending launch order.

```
global_id(0) = get_group_id(0) * get_local_size(0) + get_local_id(0)
global_id(1) = get_group_id(1) * get_local_size(1) + get_local_id(1)
```

The hardware launch order is fixed, but it is possible to change the launch order, as shown in the following example.

Assume a work-group size of k x k, where k is a power of two, and a large 2D matrix of size 2^n x 2^m in row-major order. If each work-group must process a block in column-order, the launch order does not work out correctly: consecutive work-groups execute down the columns, and the columns are a large power-of-two apart; so, consecutive work-groups access the same channel.

By introducing a transformation, it is possible to stagger the work-groups to avoid channel conflicts. Since we are executing 2D work-groups, each work group is identified by four numbers.

1. `get_group_id(0)` - the x coordinate or the block within the column of the matrix.
2. `get_group_id(1)` - the y coordinate or the block within the row of the matrix.
3. `get_global_id(0)` - the x coordinate or the column of the matrix.
4. `get_global_id(1)` - the y coordinate or the row of the matrix.

To transform the code, add the following four lines to the top of the kernel.

```
get_group_id_0 = get_group_id(0);
get_group_id_1 = (get_group_id(0) + get_group_id(1)) % get_local_size(0);
get_global_id_0 = get_group_id_0 * get_local_size(0) + get_local_id(0);
get_global_id_1 = get_group_id_1 * get_local_size(1) + get_local_id(1);
```

Then, change the global IDs and group IDs to the staggered form. The result is:

```
__kernel void
copy_float (
  __global const DATA_TYPE * A,
  __global DATA_TYPE * C)
{
  size_t get_group_id_0 = get_group_id(0);
  size_t get_group_id_1 = (get_group_id(0) + get_group_id(1)) %
                           get_local_size(0);

  size_t get_global_id_0 = get_group_id_0 * get_local_size(0) +
                            get_local_id(0);
  size_t get_global_id_1 = get_group_id_1 * get_local_size(1) +
                            get_local_id(1);

  int idx = get_global_id_0; //changed to staggered form
  int idy = get_global_id_1; //changed to staggered form

  C(idy , idx) = A( idy , idx);
}
```

6.1.2.2 Reads Of The Same Address

Under certain conditions, one unexpected case of a channel conflict is that reading from the same address is a conflict, even on the FastPath.

This does not happen on the read-only memories, such as constant buffers, textures, or shader resource view (SRV); but it is possible on the read/write UAV memory or OpenCL global memory.

From a hardware standpoint, reads from a fixed address have the same upper bits, so they collide and are serialized. To read in a single value, read the value in a single work-item, place it in local memory, and then use that location:

Avoid:

```
temp = input[3] // if input is from global space
```

Use:

```
if (get_local_id(0) == 0) {
  local = input[3]
}
barrier(CLK_LOCAL_MEM_FENCE);
temp = local
```

6.1.3 Float4 Or Float1

The internal memory paths on ATI Radeon™ HD 5000-series devices support 128-bit transfers. This allows for greater bandwidth when transferring data in float4 format. In certain cases (when the data size is a multiple of four), float4 operations are faster.

The performance of these kernels can be seen in Figure 6.4. Change to float4 after eliminating the conflicts.

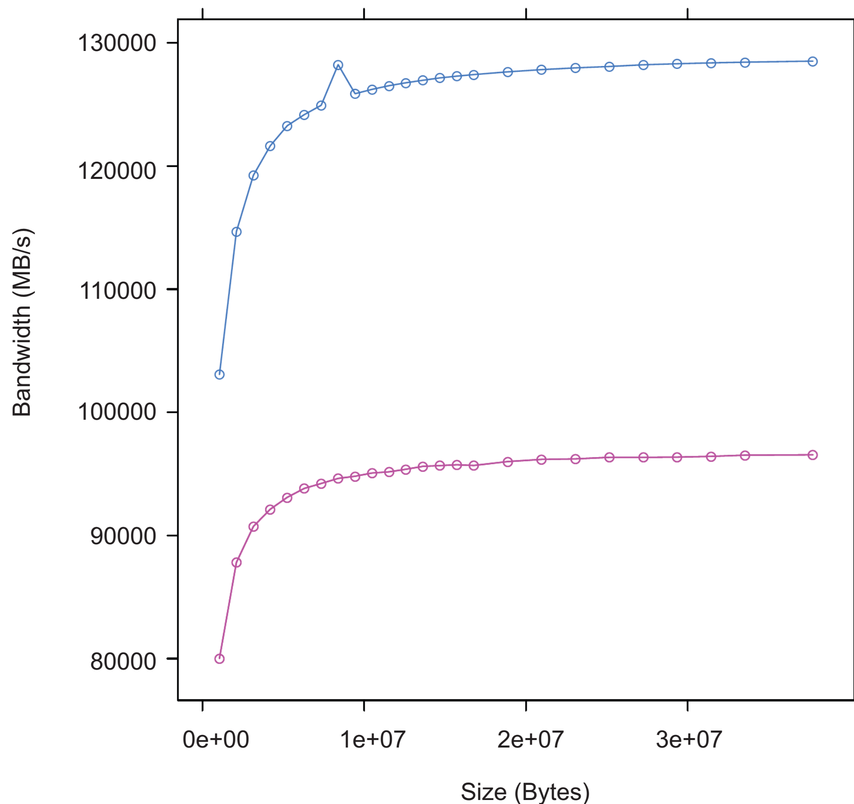

Figure 6.4 Two Kernels: One Using float4 (blue), the Other float1 (red)

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

The following code example has two kernels, both of which can do a simple copy, but Copy4 uses float4 data types.

```
__kernel void
Copy4(__global const float4 * input,
      __global float4 * output)
{
  int gid = get_global_id(0);
  output[gid] = input[gid];
  return;
}
__kernel void
Copy1(__global const float * input,
      __global float * output)
{
  int gid = get_global_id(0);
  output[gid] = input[gid];
  return;
}
```

Copying data as float4 gives the best result: 84% of absolute peak. It also speeds up the 2D versions of the copy (see Table 6.3).

Table 6.3 Bandwidths Including float1 and float4

Kernel	Effective Bandwidth	Ratio to Peak Bandwidth
copy 32-bit 1D FP	96 GB/s	63%
copy 32-bit 1D CP	18 GB/s	12%
copy 32-bit 2D	.3 - 93 GB/s	0 - 61%
copy 128-bit 2D	7 - 122 GB/s	5 - 80%
copy4 float4 1D FP	127 GB/s	83%

6.1.4 Coalesced Writes

On some other vendor devices, it is important to reorder your data to use coalesced writes. The ATI Radeon™ HD 5000-series devices also support coalesced writes, but this optimization is less important than other considerations, such as avoiding bank conflicts.

In non-coalesced writes, each compute unit accesses the memory system in quarter-wavefront units. The compute unit transfers a 32-bit address and one element-sized piece of data for each work-item. This results in a total of 16 elements + 16 addresses per quarter-wavefront. On ATI Radeon™ HD 5000-series devices, processing quarter-wavefront requires two cycles before the data is transferred to the memory controller.

In coalesced writes, the compute unit transfers one 32-bit address and 16 element-sized pieces of data for each quarter-wavefront, for a total of 16 elements +1 address per quarter-wavefront. For coalesced writes, processing quarter-wavefront takes one cycle instead of two. While this is twice as fast, the times are small compared to the rate the memory controller can handle the data. See Figure 6.5.

On ATI Radeon™ HD 5000-series devices, the coalescing is only done on the FastPath because it supports only 32-bit access.

If a work-item does not write, coalesce detection ignores it.

The first kernel Copy1 maximizes coalesced writes: work-item k writes to address k. The second kernel writes a shifted pattern: In each quarter-wavefront of 16 work-items, work-item k writes to address k-1, except the first work-item in each quarter-wavefront writes to address k+16. There is not enough order here to coalesce on some other vendor machines. Finally, the third kernel has work-item k write to address k when k is even, and write address 63-k when k is odd. This pattern never coalesces.

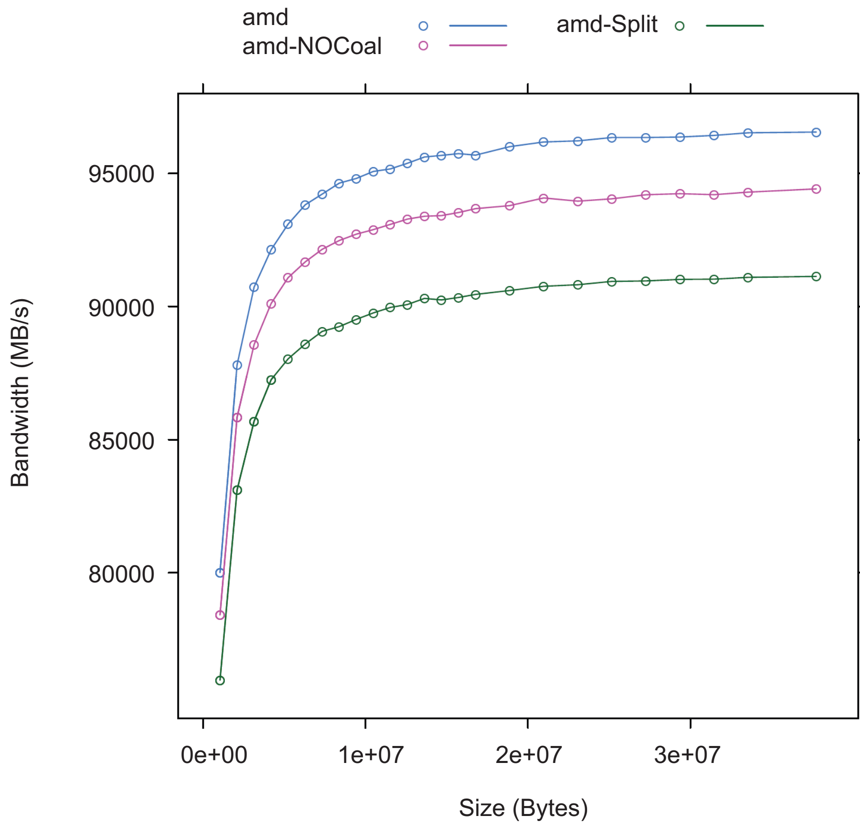

Figure 6.5 Effect of Varying Degrees of Coalescing - Coal (blue), NoCoal (red), Split (green)

Write coalescing can be an important factor for AMD GPUs.

The following are sample kernels with different coalescing patterns.

```
// best access pattern
__kernel void
Copy1(__global const float * input, __global float * output)
{
  uint gid = get_global_id(0);
  output[gid] = input[gid];
  return;
}
__kernel void NoCoal (__global const float * input,
__global float * output)
// (shift by 16)
{
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

```
  int gid = get_global_id(0)-1;
  if((get_local_id(0) & 0xf) == 0)
   {
     gid = gid +16;
   }
  output[gid] = input[gid];
  return;
}
__kernel void
// inefficient pattern
Split (__global const float * input, __global float * output)
{
  int gid = get_global_id(0);
  if((gid & 0x1) == 0) {
  gid = (gid & (˜63)) +62 - get_local_id(0);
}
output[gid] = input[gid];
return;
}
```

Table 6.4 lists the effective bandwidth and ratio to maximum bandwidth for each kernel type.

Table 6.4 Bandwidths Including Coalesced Writes

Kernel	Effective Bandwidth	Ratio to Peak Bandwidth
copy 32-bit 1D FP	96 GB/s	63%
copy 32-bit 1D CP	18 GB/s	12%
copy 32-bit 2D	.3 - 93 GB/s	0 - 61%
copy 128-bit 2D	7 - 122 GB/s	5 - 80%
copy4 float4 1D FP	127 GB/s	83%
Coal 32-bit	97	63%
NoCoal 32-bit	93 GB/s	61%
Split 32-bit	90 GB/s	59%

There is not much performance difference, although the coalesced version is slightly faster.

6.1.5 Alignment

The program in Figure 6.6 shows how the performance of a simple, unaligned access (float1) of this kernel varies as the size of offset varies. Each transfer was large (16 MB). The performance gain by adjusting alignment is small, so generally this is not an important consideration on AMD GPUs.

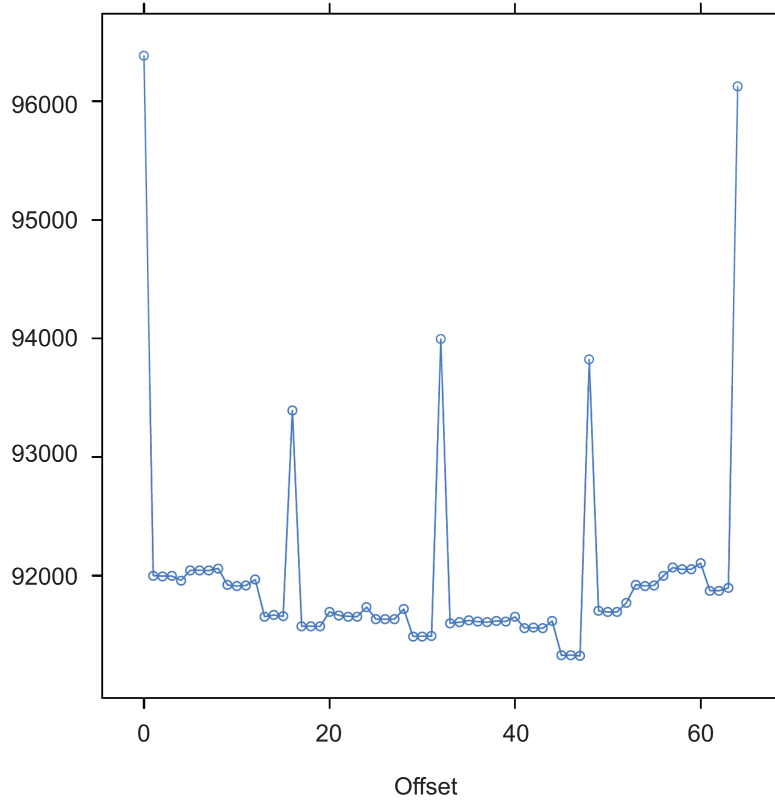

Figure 6.6 Unaligned Access Using float1

```
__kernel void
CopyAdd(global const float * input,
__global float * output,
const int offset)
{
int gid = get_global_id(0)+ offset;
output[gid] = input[gid];
return;
}
```

Table 6.5 lists the effective bandwidth and ratio to maximum bandwidth for each kernel type.

Table 6.5 Bandwidths Including Unaligned Access

Kernel	Effective Bandwidth	Ratio to Peak Bandwidth
copy 32-bit 1D FP	96 GB/s	63%
copy 32-bit 1D CP	18 GB/s	12%
copy 32-bit 2D	.3 - 93 GB/s	0 - 61%
copy 128-bit 2D	7 - 122 GB/s	5 - 80%
copy4 float4 1D FP	127 GB/s	83%
Coal	97	63%
NoCoal 32-bit	90 GB/s	59%
Split 32-bit	90 GB/s	59%
CopyAdd 32-bit	92 GB/s	60%

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

6.1.6 Summary of Copy Performance

The performance of a copy can vary greatly, depending on how the code is written. The measured bandwidth for these copies varies from a low of 0.3 GB/s, to a high of 127 GB/s.

The recommended order of steps to improve performance is:

1. Examine the code to ensure you are using FastPath, not CompletePath, everywhere possible. Check carefully to see if you are minimizing the number of kernels that use CompletePath operations. You might be able to use textures, image-objects, or constant buffers to help.
2. Examine the data-set sizes and launch dimensions to see if you can eliminate bank conflicts.
3. Try to use float4 instead of float1.
4. Try to change the access pattern to allow write coalescing. This is important on some hardware platforms, but only of limited importance for AMD GPU devices.
5. Finally, look at changing the access pattern to allow data alignment.

6.1.7 Hardware Variations

For a listing of the AMD GPU hardware variations, see Appendix D, “Device Parameters.” This appendix includes information on the number of memory channels, compute units, and the L2 size per device.

6.2 Local Memory (LDS) Optimization

AMD Evergreen GPUs include a Local Data Store (LDS) cache, which accelerates local memory accesses. LDS is not supported in OpenCL on AMD R700-family GPUs. LDS provides high-bandwidth access (more than 10X higher than global memory), efficient data transfers between work-items in a work-group, and high-performance atomic support. Local memory offers significant advantages when the data is re-used; for example, subsequent accesses can read from local memory, thus reducing global memory bandwidth. Another advantage is that local memory does not require coalescing.

To determine local memory size:

```
clGetDeviceInfo( …, CL_DEVICE_LOCAL_MEM_SIZE, … );
```

All AMD Evergreen GPUs contain a 32K LDS for each compute unit. On high-end GPUs, the LDS contains 32-banks, each bank is four bytes wide and 256 bytes deep; the bank address is determined by bits 6:2 in the address. On lower-end GPUs, the LDS contains 16 banks, each bank is still 4 bytes in size, and the bank used is determined by bits 5:2 in the address. Appendix D, “Device Parameters” shows how many LDS banks are present on the different AMD Evergreen products. As shown below, programmers should carefully control the bank bits to avoid bank conflicts as much as possible.

In a single cycle, local memory can service a request for each bank (up to 32 accesses each cycle on the ATI Radeon™ HD 5870 GPU). For an ATI Radeon™ HD 5870 GPU, this delivers a memory bandwidth of over 100 GB/s for each compute unit, and more than 2 TB/s for the whole chip. This is more than 14X the global memory bandwidth. However, accesses that map to the same bank are serialized and serviced on consecutive cycles. A wavefront that generates bank conflicts stalls on the compute unit until all LDS accesses have completed. The GPU reprocesses the wavefront on subsequent cycles, enabling only the lanes receiving data, until all the conflicting accesses complete. The bank with the most conflicting accesses determines the latency for the wavefront to complete the local memory operation. The worst case occurs when all 64 work-items map to the same bank, since each access then is serviced at a rate of one per clock cycle; this case takes 64 cycles to complete the local memory access for the wavefront. A program with a large number of bank conflicts (as measured by the `LDSBankConflict` performance counter) might benefit from using the constant or image memory rather than LDS.

Thus, the key to effectively using the local cache memory is to control the access pattern so that accesses generated on the same cycle map to different banks in the local memory. One notable exception is that accesses to the same address (even though they have the same bits 6:2) can be broadcast to all requestors and do not generate a bank conflict. The LDS hardware examines the requests generated over two cycles (32 work-items of execution) for bank conflicts. Ensure, as much as possible, that the memory requests generated from a quarter-wavefront avoid bank conflicts by using unique address bits 6:2. A simple sequential address pattern, where each work-item reads a float2 value from LDS, generates a conflict-free access pattern on the ATI Radeon™ HD 5870 GPU. Note that a sequential access pattern, where each work-item reads a float4 value from LDS, uses only half the banks on each cycle on the ATI Radeon™ HD 5870 GPU and delivers half the performance of the float access pattern.

Each stream processor can generate up to two 4-byte LDS requests per cycle. Byte and short reads consume four bytes of LDS bandwidth. Since each stream processor can execute five operations (or four, depending on the GPU type) in the VLIW each cycle (typically requiring 10-15 input operands), two local memory requests might not provide enough bandwidth to service the entire instruction. Developers can use the large register file: each compute unit has 256 kB of register space available (8X the LDS size) and can provide up to twelve 4-byte values/cycle (6X the LDS bandwidth). Registers do not offer the same indexing flexibility as does the LDS, but for some algorithms this can be overcome with loop unrolling and explicit addressing.

LDS reads require one ALU operation to initiate them. Each operation can initiate two loads of up to four bytes each.

The AMD APP Profiler provides the following performance counter to help optimize local memory usage:

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

`LDSBankConflict`: The percentage of time accesses to the LDS are stalled due to bank conflicts relative to GPU Time. In the ideal case, there are no bank conflicts in the local memory access, and this number is zero.

Local memory is software-controlled "scratchpad" memory. In contrast, caches typically used on CPUs monitor the access stream and automatically capture recent accesses in a tagged cache. The scratchpad allows the kernel to explicitly load items into the memory; they exist in local memory until the kernel replaces them, or until the work-group ends. To declare a block of local memory, use the `__local` keyword; for example:

```
__local float localBuffer[64]
```

These declarations can be either in the parameters to the kernel call or in the body of the kernel. The `__local` syntax allocates a single block of memory, which is shared across all work-items in the workgroup.

To write data into local memory, write it into an array allocated with `__local`. For example:

```
localBuffer[i] = 5.0;
```

A typical access pattern is for each work-item to collaboratively write to the local memory: each work-item writes a subsection, and as the work-items execute in parallel they write the entire array. Combined with proper consideration for the access pattern and bank alignment, these collaborative write approaches can lead to highly efficient memory accessing. Local memory is consistent across work-items only at a work-group barrier; thus, before reading the values written collaboratively, the kernel must include a `barrier()` instruction.

The following example is a simple kernel section that collaboratively writes, then reads from, local memory:

```
__kernel void localMemoryExample (__global float *In, __global float *Out) {
    __local float localBuffer[64];
    uint tx = get_local_id(0);
    uint gx = get_global_id(0);

    // Initialize local memory:
    // Copy from this work-group's section of global memory to local:
    // Each work-item writes one element; together they write it all
    localBuffer[tx] = In[gx];

    // Ensure writes have completed:
    barrier(CLK_LOCAL_MEM_FENCE);

    // Toy computation to compute a partial factorial, shows re-use from local

    float f = localBuffer[tx];
    for (uint i=tx+1; i<64; i++) {
            f *= localBuffer[i];
    }
    Out[gx] = f;
}
```

Note the host code cannot read from, or write to, local memory. Only the kernel can access local memory.

Local memory is consistent across work-items only at a work-group barrier; thus, before reading the values written collaboratively, the kernel must include a `barrier()` instruction. An important optimization is the case where the local work-group size is less than, or equal to, the wavefront size. Because the wavefront executes as an atomic unit, the explicit barrier operation is not required. The compiler automatically removes these barriers if the kernel specifies a `reqd_work_group_size`

(see section 5.8 of the *OpenCL Specification*) that is less than the wavefront size. Developers are strongly encouraged to include the barriers where appropriate, and rely on the compiler to remove the barriers when possible, rather than manually removing the `barriers()`. This technique results in more portable code, including the ability to run kernels on CPU devices.

6.3 Constant Memory Optimization

The AMD implementation of OpenCL provides three levels of performance for the “constant” memory type.

1. Simple Direct-Addressing Patterns

   Very high bandwidth can be attained when the compiler has available the constant address at compile time and can embed the constant address into the instruction. Each processing element can load up to 4x4-byte direct-addressed constant values each cycle. Typically, these cases are limited to simple non-array constants and function parameters. The GPU loads the constants into a hardware cache at the beginning of the clause that uses the constants. The cache is a tagged cache, typically each 8k blocks is shared among four compute units. If the constant data is already present in the constant cache, the load is serviced by the cache and does not require any global memory bandwidth. The constant cache size for each device is given in Appendix D, “Device Parameters”; it varies from 4k to 48k per GPU.

2. Same Index

   Hardware acceleration also takes place when all work-items in a wavefront reference the same constant address. In this case, the data is loaded from memory one time, stored in the L1 cache, and then broadcast to all wave-fronts. This can reduce significantly the required memory bandwidth.

3. Varying Index

   More sophisticated addressing patterns, including the case where each work-item accesses different indices, are not hardware accelerated and deliver the same performance as a global memory read with the potential for cache hits.

To further improve the performance of the AMD OpenCL stack, two methods allow users to take advantage of hardware constant buffers. These are:

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

1. Globally scoped constant arrays. These arrays are initialized, globally scoped, and in the constant address space (as specified in section 6.5.3 of the OpenCL specification). If the size of an array is below 64 kB, it is placed in hardware constant buffers; otherwise, it uses global memory. An example of this is a lookup table for math functions.
2. Per-pointer attribute specifying the maximum pointer size. This is specified using the `max_constant_size(N)` attribute. The attribute form conforms to section 6.10 of the OpenCL 1.0 specification. This attribute is restricted to top-level kernel function arguments in the constant address space. This restriction prevents a pointer of one size from being passed as an argument to a function that declares a different size. It informs the compiler that indices into the pointer remain inside this range and it is safe to allocate a constant buffer in hardware, if it fits. Using a constant pointer that goes outside of this range results in undefined behavior. All allocations are aligned on the 16-byte boundary. For example:

```
kernel void mykernel(global int* a,
constant int* b __attribute__((max_constant_size (65536)))
)
{
size_t idx = get_global_id(0);
a[idx] = b[idx & 0x3FFF];
}
```

A kernel that uses constant buffers must use `CL_DEVICE_MAX_CONSTANT_ARGS` to query the device for the maximum number of constant buffers the kernel can support. This value might differ from the maximum number of hardware constant buffers available. In this case, if the number of hardware constant buffers is less than the `CL_DEVICE_MAX_CONSTANT_ARGS`, the compiler allocates the largest constant buffers in hardware first and allocates the rest of the constant buffers in global memory. As an optimization, if a constant pointer **A** uses *n* bytes of memory, where *n* is less than 64 kB, and constant pointer **B** uses *m* bytes of memory, where *m* is less than (64 kB – *n*) bytes of memory, the compiler can allocate the constant buffer pointers in a single hardware constant buffer. This optimization can be applied recursively by treating the resulting allocation as a single allocation and finding the next smallest constant pointer that fits within the space left in the constant buffer.

6.4 OpenCL Memory Resources: Capacity and Performance

Table 6.6 summarizes the hardware capacity and associated performance for the structures associated with the five OpenCL Memory Types. This information specific to the ATI Radeon™ HD5870 GPUs with 1 GB video memory. See Appendix D, “Device Parameters” for more details about other GPUs.

Table 6.6 Hardware Performance Parameters

OpenCL Memory Type	Hardware Resource	Size/CU	Size/GPU	Peak Read Bandwidth/ Stream Core
Private	GPRs	256k	5120k	48 bytes/cycle
Local	LDS	32k	640k	8 bytes/cycle
Constant	Direct-addressed constant		48k	16 bytes/cycle
	Same-indexed constant			4 bytes/cycle
	Varying-indexed constant			~0.6 bytes/cycle
Images	L1 Cache	8k	160k	4 bytes/cycle
	L2 Cache		512k	~1.6 bytes/cycle
Global	Global Memory		1G	~0.6 bytes/cycle

The compiler tries to map private memory allocations to the pool of GPRs in the GPU. In the event GPRs are not available, private memory is mapped to the "scratch" region, which has the same performance as global memory. Section 6.6.2, "Resource Limits on Active Wavefronts," page 6-24, has more information on register allocation and identifying when the compiler uses the scratch region. GPRs provide the highest-bandwidth access of any hardware resource. In addition to reading up to 48 bytes/cycle from the register file, the hardware can access results produced in the previous cycle (through the Previous Vector/Previous Scalar register) without consuming any register file bandwidth. GPRs have some restrictions about which register ports can be read on each cycle; but generally, these are not exposed to the OpenCL programmer.

Same-indexed constants can be cached in the L1 and L2 cache. Note that "same-indexed" refers to the case where all work-items in the wavefront reference the same constant index on the same cycle. The performance shown assumes an L1 cache hit.

Varying-indexed constants use the same path as global memory access and are subject to the same bank and alignment constraints described in Section 6.1, "Global Memory Optimization," page 6-1.

The L1 and L2 caches are currently only enabled for images and same-indexed constants. As of SDK 2.4, read only buffers can be cached in L1 and L2. To enable this, the developer must indicate to the compiler that the buffer is read only and does not alias with other buffers. For example, use:

```
kernel void mykernel(__global int const * restrict mypointerName)
```

The `const` indicates to the compiler that `mypointerName` is read only from the kernel, and the `restrict` attribute indicates to the compiler that no other pointer aliases with `mypointerName`.

The L1 cache can service up to four address request per cycle, each delivering up to 16 bytes. The bandwidth shown assumes an access size of 16 bytes; smaller access sizes/requests result in a lower peak bandwidth for the L1 cache. Using float4 with images increases the request size and can deliver higher L1 cache bandwidth.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Each memory channel on the GPU contains an L2 cache that can deliver up to 64 bytes/cycle. The ATI Radeon™ HD 5870 GPU has eight memory channels; thus, it can deliver up to 512bytes/cycle; divided among 320 stream cores, this provides up to ~1.6 bytes/cycle for each stream core.

Global Memory bandwidth is limited by external pins, not internal bus bandwidth. The ATI Radeon™ HD 5870 GPU supports up to 153 GB/s of memory bandwidth which is an average of 0.6 bytes/cycle for each stream core.

Note that Table 6.6 shows the performance for the ATI Radeon™ HD 5870 GPU. The “Size/Compute Unit” column and many of the bandwidths/processing element apply to all Evergreen-class GPUs; however, the “Size/GPU” column and the bandwidths for varying-indexed constant, L2, and global memory vary across different GPU devices. The resource capacities and peak bandwidth for other AMD GPU devices can be found in Appendix D, “Device Parameters.”

6.5 Using LDS or L1 Cache

There are a number of considerations when deciding between LDS and L1 cache for a given algorithm.

LDS supports read/modify/write operations, as well as atomics. It is well-suited for code that requires fast read/write, read/modify/write, or scatter operations that otherwise are directed to global memory. On current AMD hardware, L1 is part of the read path; hence, it is suited to cache-read-sensitive algorithms, such as matrix multiplication or convolution.

LDS is typically larger than L1 (for example: 32 kB vs 8 kB on Cypress). If it is not possible to obtain a high L1 cache hit rate for an algorithm, the larger LDS size can help. The theoretical LDS peak bandwidth is 2 TB/s, compared to L1 at 1 TB/sec. Currently, OpenCL is limited to 1 TB/sec LDS bandwidth.

The native data type for L1 is a four-vector of 32-bit words. On L1, fill and read addressing are linked. It is important that L1 is initially filled from global memory with a coalesced access pattern; once filled, random accesses come at no extra processing cost.

Currently, the native format of LDS is a 32-bit word. The theoretical LDS peak bandwidth is achieved when each thread operates on a two-vector of 32-bit words (16 threads per clock operate on 32 banks). If an algorithm requires coalesced 32-bit quantities, it maps well to LDS. The use of four-vectors or larger can lead to bank conflicts.

From an application point of view, filling LDS from global memory, and reading from it, are independent operations that can use independent addressing. Thus, LDS can be used to explicitly convert a scattered access pattern to a coalesced pattern for read and write to global memory. Or, by taking advantage of the LDS read broadcast feature, LDS can be filled with a coalesced pattern from global memory, followed by all threads iterating through the same LDS words simultaneously.

LDS is shared between the work-items in a work-group. Sharing across work-groups is not possible because OpenCL does not guarantee that LDS is in a particular state at the beginning of work-group execution. L1 content, on the other hand, is independent of work-group execution, so that successive work-groups can share the content in the L1 cache of a given Vector ALU. However, it currently is not possible to explicitly control L1 sharing across work-groups.

The use of LDS is linked to GPR usage and wavefront-per-Vector ALU count. Better sharing efficiency requires a larger work-group, so that more work items share the same LDS. Compiling kernels for larger work groups typically results in increased register use, so that fewer wavefronts can be scheduled simultaneously per Vector ALU. This, in turn, reduces memory latency hiding. Requesting larger amounts of LDS per work-group results in fewer wavefronts per Vector ALU, with the same effect.

LDS typically involves the use of barriers, with a potential performance impact. This is true even for read-only use cases, as LDS must be explicitly filled in from global memory (after which a barrier is required before reads can commence).

6.6 NDRange and Execution Range Optimization

Probably the most effective way to exploit the potential performance of the GPU is to provide enough threads to keep the device completely busy. The programmer specifies a three-dimensional NDRange over which to execute the kernel; bigger problems with larger NDRanges certainly help to more effectively use the machine. The programmer also controls how the global NDRange is divided into local ranges, as well as how much work is done in each work-item, and which resources (registers and local memory) are used by the kernel. All of these can play a role in how the work is balanced across the machine and how well it is used. This section introduces the concept of latency hiding, how many wavefronts are required to hide latency on AMD GPUs, how the resource usage in the kernel can impact the active wavefronts, and how to choose appropriate global and local work-group dimensions.

6.6.1 Hiding ALU and Memory Latency

The read-after-write latency for most arithmetic operations (a floating-point add, for example) is only eight cycles. For most AMD GPUs, each compute unit can execute 16 VLIW instructions on each cycle. Each wavefront consists of 64 work-items; each compute unit executes a quarter-wavefront on each cycle, and the entire wavefront is executed in four consecutive cycles. Thus, to hide eight cycles of latency, the program must schedule two wavefronts. The compute unit executes the first wavefront on four consecutive cycles; it then immediately switches and executes the other wavefront for four cycles. Eight cycles have elapsed, and the ALU result from the first wavefront is ready, so the compute unit can switch back to the first wavefront and continue execution. Compute units running two wavefronts (128 threads) completely hide the ALU pipeline latency.

Global memory reads generate a reference to the off-chip memory and experience a latency of 300 to 600 cycles. The wavefront that generates the

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

global memory access is made idle until the memory request completes. During this time, the compute unit can process other independent wavefronts, if they are available.

Kernel execution time also plays a role in hiding memory latency: longer kernels keep the functional units busy and effectively hide more latency. To better understand this concept, consider a global memory access which takes 400 cycles to execute. Assume the compute unit contains many other wavefronts, each of which performs five ALU instructions before generating another global memory reference. As discussed previously, the hardware executes each instruction in the wavefront in four cycles; thus, all five instructions occupy the ALU for 20 cycles. Note the compute unit interleaves two of these wavefronts and executes the five instructions from both wavefronts (10 total instructions) in 40 cycles. To fully hide the 400 cycles of latency, the compute unit requires (400/40) = 10 pairs of wavefronts, or 20 total wavefronts. If the wavefront contains 10 instructions rather than 5, the wavefront pair would consume 80 cycles of latency, and only 10 wavefronts would be required to hide the 400 cycles of latency.

Generally, it is not possible to predict how the compute unit schedules the available wavefronts, and thus it is not useful to try to predict exactly which ALU block executes when trying to hide latency. Instead, consider the overall ratio of ALU operations to fetch operations – this metric is reported by the AMD APP Profiler in the `ALUFetchRatio` counter. Each ALU operation keeps the compute unit busy for four cycles, so you can roughly divide 500 cycles of latency by (`4*ALUFetchRatio`) to determine how many wavefronts must be in-flight to hide that latency. Additionally, a low value for the `ALUBusy` performance counter can indicate that the compute unit is not providing enough wavefronts to keep the execution resources in full use. (This counter also can be low if the kernel exhausts the available DRAM bandwidth. In this case, generating more wavefronts does not improve performance; it can reduce performance by creating more contention.)

Increasing the wavefronts/compute unit does not indefinitely improve performance; once the GPU has enough wavefronts to hide latency, additional active wavefronts provide little or no performance benefit. A closely related metric to wavefronts/compute unit is “occupancy,” which is defined as the ratio of active wavefronts to the maximum number of possible wavefronts supported by the hardware. Many of the important optimization targets and resource limits are expressed in wavefronts/compute units, so this section uses this metric rather than the related “occupancy” term.

6.6.2 Resource Limits on Active Wavefronts

AMD GPUs have two important global resource constraints that limit the number of in-flight wavefronts:

- Each compute unit supports a maximum of eight work-groups. Recall that AMD OpenCL supports up to 256 work-items (four wavefronts) per work-

group; effectively, this means each compute unit can support up to 32 wavefronts.

- Each GPU has a global (across all compute units) limit on the number of active wavefronts. The GPU hardware is generally effective at balancing the load across available compute units. Thus, it is useful to convert this global limit into an average wavefront/compute unit so that it can be compared to the other limits discussed in this section. For example, the ATI Radeon™ HD 5870 GPU has a global limit of 496 wavefronts, shared among 20 compute units. Thus, it supports an average of 24.8 wavefronts/compute unit. Appendix D, "Device Parameters" contains information on the global number of wavefronts supported by other AMD GPUs. Some AMD GPUs support up to 96 wavefronts/compute unit.

These limits are largely properties of the hardware and, thus, difficult for developers to control directly. Fortunately, these are relatively generous limits. Frequently, the register and LDS usage in the kernel determines the limit on the number of active wavefronts/compute unit, and these can be controlled by the developer.

6.6.2.1 GPU Registers

Each compute unit provides 16384 GP registers, and each register contains 4x32-bit values (either single-precision floating point or a 32-bit integer). The total register size is 256 kB of storage per compute unit. These registers are shared among all active wavefronts on the compute unit; each kernel allocates only the registers it needs from the shared pool. This is unlike a CPU, where each thread is assigned a fixed set of architectural registers. However, using many registers in a kernel depletes the shared pool and eventually causes the hardware to throttle the maximum number of active wavefronts.

Table 6.7 shows how the registers used in the kernel impacts the register-limited wavefronts/compute unit.

For example, a kernel that uses 30 registers (120x32-bit values) can run with eight active wavefronts on each compute unit. Because of the global limits described earlier, each compute unit is limited to 32 wavefronts; thus, kernels can use up to seven registers (28 values) without affecting the number of wavefronts/compute unit. Finally, note that in addition to the GPRs shown in the table, each kernel has access to four clause temporary registers.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Table 6.7 Impact of Register Type on Wavefronts/CU

GP Registers used by Kernel	Register-Limited Wavefronts / Compute-Unit
0-1	248
2	124
3	82
4	62
5	49
6	41
7	35
8	31
9	27
10	24
11	22
12	20
13	19
14	17
15	16
16	15
17	14
18-19	13
19-20	12
21-22	11
23-24	10
25-27	9
28-31	8
32-35	7
36-41	6
42-49	5
50-62	4
63-82	3
83-124	2

AMD provides the following tools to examine the number of general-purpose registers (GPRs) used by the kernel.

- The AMD APP Profiler displays the number of GPRs used by the kernel.
- Alternatively, the AMD APP Profiler generates the ISA dump (described in Section 4.3, “Analyzing Processor Kernels,” page 4-9), which then can be searched for the string `:NUM_GPRS`.
- The AMD APP KernelAnalyzer also shows the GPR used by the kernel, across a wide variety of GPU compilation targets.

The compiler generates spill code (shuffling values to, and from, memory) if it cannot fit all the live values into registers. Spill code uses long-latency global memory and can have a large impact on performance. The AMD APP Profiler reports the static number of register spills in the `ScratchReg` field. Generally, it is a good idea to re-write the algorithm to use fewer GPRs, or tune the work-group dimensions specified at launch time to expose more registers/kernel to the compiler, in order to reduce the scratch register usage to 0.

6.6.2.2 Specifying the Default Work-Group Size at Compile-Time

The number of registers used by a work-item is determined when the kernel is compiled. The user later specifies the size of the work-group. Ideally, the OpenCL compiler knows the size of the work-group at compile-time, so it can make optimal register allocation decisions. Without knowing the work-group size, the compiler must assume an upper-bound size to avoid allocating more registers in the work-item than the hardware actually contains.

For example, if the compiler allocates 70 registers for the work-item, Table 6.7 shows that only three wavefronts (192 work-items) are supported. If the user later launches the kernel with a work-group size of four wavefronts (256 work-items), the launch fails because the work-group requires 70*256=17920 registers, which is more than the hardware allows. To prevent this from happening, the compiler performs the register allocation with the conservative assumption that the kernel is launched with the largest work-group size (256 work-items). The compiler guarantees that the kernel does not use more than 62 registers (the maximum number of registers which supports a work-group with four wave-fronts), and generates low-performing register spill code, if necessary.

Fortunately, OpenCL provides a mechanism to specify a work-group size that the compiler can use to optimize the register allocation. In particular, specifying a smaller work-group size at compile time allows the compiler to allocate more registers for each kernel, which can avoid spill code and improve performance. The kernel attribute syntax is:

```
__attribute__((reqd_work_group_size(X, Y, Z)))
```

Section 6.7.2 of the OpenCL specification explains the attribute in more detail.

6.6.2.3 Local Memory (LDS) Size

In addition to registers, shared memory can also serve to limit the active wavefronts/compute unit. Each compute unit has 32k of LDS, which is shared among all active work-groups. LDS is allocated on a per-work-group granularity, so it is possible (and useful) for multiple wavefronts to share the same local memory allocation. However, large LDS allocations eventually limits the number of workgroups that can be active. Table 6.8 provides more details about how LDS usage can impact the wavefronts/compute unit.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Table 6.8 Effect of LDS Usage on Wavefronts/CU[1]

Local Memory / Work-Group	LDS-Limited Work-Groups	LDS-Limited Wavefronts/ Compute-Unit (Assume 4 Wavefronts/ Work-Group)	LDS-Limited Wavefronts/ Compute-Unit (Assume 3 Wavefronts/ Work-Group)	LDS-Limited Wavefronts/ Compute-Unit (Assume 2 Wavefronts/ Work-Group)
<=4K	8	32	24	16
4.0K-4.6K	7	28	21	14
4.6K-5.3K	6	24	18	12
5.3K-6.4K	5	20	15	10
6.4K-8.0K	4	16	12	8
8.0K-10.7K	3	12	9	6
10.7K-16.0K	2	8	6	4
16.0K-32.0K	1	4	3	2

1. Assumes each work-group uses four wavefronts (the maximum supported by the AMD OpenCL SDK).

AMD provides the following tools to examine the amount of LDS used by the kernel:

- The AMD APP Profiler displays the LDS usage. See the `LocalMem` counter.
- Alternatively, use the AMD APP Profiler to generate the ISA dump (described in Section 4.3, "Analyzing Processor Kernels," page 4-9), then search for the string `SQ_LDS_ALLOC:SIZE` in the ISA dump. Note that the value is shown in hexadecimal format.

6.6.3 Partitioning the Work

In OpenCL, each kernel executes on an index point that exists in a global NDRange. The partition of the NDRange can have a significant impact on performance; thus, it is recommended that the developer explicitly specify the global (`#work-groups`) and local (`#work-items/work-group`) dimensions, rather than rely on OpenCL to set these automatically (by setting `local_work_size` to NULL in `clEnqueueNDRangeKernel`). This section explains the guidelines for partitioning at the global, local, and work/kernel levels.

6.6.3.1 Global Work Size

OpenCL does not explicitly limit the number of work-groups that can be submitted with a `clEnqueueNDRangeKernel` command. The hardware limits the available in-flight threads, but the OpenCL SDK automatically partitions a large number of work-groups into smaller pieces that the hardware can process. For some large workloads, the amount of memory available to the GPU can be a limitation; the problem might require so much memory capacity that the GPU cannot hold it all.

In these cases, the programmer must partition the workload into multiple `clEnqueueNDRangeKernel` commands. The available device memory can be obtained by querying `clDeviceInfo`.

At a minimum, ensure that the workload contains at least as many work-groups as the number of compute units in the hardware. Work-groups cannot be split across multiple compute units, so if the number of work-groups is less than the available compute units, some units are idle. Evergreen and Northern Islands GPUs have 2-24 compute units. (See Appendix D, “Device Parameters” for a table of device parameters, including the number of compute units, or use `clGetDeviceInfo(…CL_DEVICE_MAX_COMPUTE_UNITS)` to determine the value dynamically).

6.6.3.2 Local Work Size (#Work-Items per Work-Group)

OpenCL limits the number of work-items in each group. Call `clDeviceInfo` with the `CL_DEVICE_MAX_WORK_GROUP_SIZE` to determine the maximum number of work-groups supported by the hardware. Currently, AMD GPUs with SDK 2.1 return 256 as the maximum number of work-items per work-group. Note the number of work-items is the product of all work-group dimensions; for example, a work-group with dimensions 32x16 requires 512 work-items, which is not allowed with the current AMD OpenCL SDK.

The fundamental unit of work on AMD GPUs is called a wavefront. Each wavefront consists of 64 work-items; thus, the optimal local work size is an integer multiple of 64 (specifically 64, 128, 192, or 256) work-items per work-group.

Work-items in the same work-group can share data through LDS memory and also use high-speed local atomic operations. Thus, larger work-groups enable more work-items to efficiently share data, which can reduce the amount of slower global communication. However, larger work-groups reduce the number of global work-groups, which, for small workloads, could result in idle compute units. Generally, larger work-groups are better as long as the global range is big enough to provide 1-2 Work-Groups for each compute unit in the system; for small workloads it generally works best to reduce the work-group size in order to avoid idle compute units. Note that it is possible to make the decision dynamically, when the kernel is launched, based on the launch dimensions and the target device characteristics.

6.6.3.3 Moving Work to the Kernel

Often, work can be moved from the work-group into the kernel. For example, a matrix multiply where each work-item computes a single element in the output array can be written so that each work-item generates multiple elements. This technique can be important for effectively using the processing elements available in the five-wide (or four-wide, depending on the GPU type) VLIW processing engine (see the `ALUPacking` performance counter reported by the AMD APP Profiler). The mechanics of this technique often is as simple as adding a `for` loop around the kernel, so that the kernel body is run multiple times inside

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

this loop, then adjusting the global work size to reduce the work-items. Typically, the local work-group is unchanged, and the net effect of moving work into the kernel is that each work-group does more effective processing, and fewer global work-groups are required.

When moving work to the kernel, often it is best to combine work-items that are separated by 16 in the NDRange index space, rather than combining adjacent work-items. Combining the work-items in this fashion preserves the memory access patterns optimal for global and local memory accesses. For example, consider a kernel where each kernel accesses one four-byte element in array A. The resulting access pattern is:

Work-item	0	1	2	3	...
Cycle0	A+0	A+1	A+2	A+3	

If we naively combine four adjacent work-items to increase the work processed per kernel, so that the first work-item accesses array elements A+0 to A+3 on successive cycles, the overall access pattern is:

Work-item	0	1	2	3	4	5	
Cycle0	A+0	A+4	A+8	A+12	A+16	A+20	
Cycle1	A+1	A+5	A+9	A+13	A+17	A+21	...
Cycle2	A+2	A+6	A+10	A+14	A+18	A+22	
Cycle3	A+3	A+7	A+11	A+15	A+19	A+23	

This pattern shows that on the first cycle the access pattern contains “holes.” Also, this pattern results in bank conflicts on the LDS. A better access pattern is to combine four work-items so that the first work-item accesses array elements A+0, A+16, A+32, and A+48. The resulting access pattern is:

Work-item	0	1	2	3	4	5	
Cycle0	A+0	A+1	A+2	A+3	A+4	A+5	
Cycle1	A+16	A+17	A+18	A+19	A+20	A+21	...
Cycle2	A+32	A+33	A+34	A+35	A+36	A+37	
Cycle3	A+48	A+49	A+50	A+51	A+52	A+53	

Note that this access patterns preserves the sequentially-increasing addressing of the original kernel and generates efficient global and LDS memory references.

Increasing the processing done by the kernels can allow more processing to be done on the fixed pool of local memory available to work-groups. For example, consider a case where an algorithm requires 32x32 elements of shared memory. If each work-item processes only one element, it requires 1024 work-items/work-group, which exceeds the maximum limit. Instead, each kernel can be written to

process four elements, and a work-group of 16x16 work-items could be launched to process the entire array. A related example is a blocked algorithm, such as a matrix multiply; the performance often scales with the size of the array that can be cached and used to block the algorithm. By moving processing tasks into the kernel, the kernel can use the available local memory rather than being limited by the work-items/work-group.

6.6.3.4 Work-Group Dimensions vs Size

The local NDRange can contain up to three dimensions, here labeled X, Y, and Z. The X dimension is returned by `get_local_id(0)`, Y is returned by `get_local_id(1)`, and Z is returned by `get_local_id(2)`. The GPU hardware schedules the kernels so that the X dimensions moves fastest as the work-items are packed into wavefronts. For example, the 128 threads in a 2D work-group of dimension 32x4 (X=32 and Y=4) would be packed into two wavefronts as follows (notation shown in X,Y order):

<table>
<tr><td rowspan="4">WaveFront0</td><td>0,0</td><td>1,0</td><td>2,0</td><td>3,0</td><td>4,0</td><td>5,0</td><td>6,0</td><td>7,0</td><td>8,0</td><td>9,0</td><td>10,0</td><td>11,0</td><td>12,0</td><td>13,0</td><td>14,0</td><td>15,0</td></tr>
<tr><td>16,0</td><td>17,0</td><td>18,0</td><td>19,0</td><td>20,0</td><td>21,0</td><td>22,0</td><td>23,0</td><td>24,0</td><td>25,0</td><td>26,0</td><td>27,0</td><td>28,0</td><td>29,0</td><td>30,0</td><td>31,0</td></tr>
<tr><td>0,1</td><td>1,1</td><td>2,1</td><td>3,1</td><td>4,1</td><td>5,1</td><td>6,1</td><td>7,1</td><td>8,1</td><td>9,1</td><td>10,1</td><td>11,1</td><td>12,1</td><td>13,1</td><td>14,1</td><td>15,1</td></tr>
<tr><td>16,1</td><td>17,1</td><td>18,1</td><td>19,1</td><td>20,1</td><td>21,1</td><td>22,1</td><td>23,1</td><td>24,1</td><td>25,1</td><td>26,1</td><td>27,1</td><td>28,1</td><td>29,1</td><td>30,1</td><td>31,1</td></tr>
<tr><td rowspan="4">WaveFront1</td><td>0,2</td><td>1,2</td><td>2,2</td><td>3,2</td><td>4,2</td><td>5,2</td><td>6,2</td><td>7,2</td><td>8,2</td><td>9,2</td><td>10,2</td><td>11,2</td><td>12,2</td><td>13,2</td><td>14,2</td><td>15,2</td></tr>
<tr><td>16,2</td><td>17,2</td><td>18,2</td><td>19,2</td><td>20,2</td><td>21,2</td><td>22,2</td><td>23,2</td><td>24,2</td><td>25,2</td><td>26,2</td><td>27,2</td><td>28,2</td><td>29,2</td><td>30,2</td><td>31,2</td></tr>
<tr><td>0,3</td><td>1,3</td><td>2,3</td><td>3,3</td><td>4,3</td><td>5,3</td><td>6,3</td><td>7,3</td><td>8,3</td><td>9,3</td><td>10,3</td><td>11,3</td><td>12,3</td><td>13,3</td><td>14,3</td><td>15,3</td></tr>
<tr><td>16,3</td><td>17,3</td><td>18,3</td><td>19,3</td><td>20,3</td><td>21,3</td><td>22,3</td><td>23,3</td><td>24,3</td><td>25,3</td><td>26,3</td><td>27,3</td><td>28,3</td><td>29,3</td><td>30,3</td><td>31,3</td></tr>
</table>

The total number of work-items in the work-group is typically the most important parameter to consider, in particular when optimizing to hide latency by increasing wavefronts/compute unit. However, the choice of XYZ dimensions for the same overall work-group size can have the following second-order effects.

- Work-items in the same quarter-wavefront execute on the same cycle in the processing engine. Thus, global memory coalescing and local memory bank conflicts can be impacted by dimension, particularly if the fast-moving X dimension is small. Typically, it is best to choose an X dimension of at least 16, then optimize the memory patterns for a block of 16 work-items which differ by 1 in the X dimension.
- Work-items in the same wavefront have the same program counter and execute the same instruction on each cycle. The packing order can be important if the kernel contains divergent branches. If possible, pack together work-items that are likely to follow the same direction when control-flow is encountered. For example, consider an image-processing kernel where each work-item processes one pixel, and the control-flow depends on the color of the pixel. It might be more likely that a square of 8x8 pixels is the same color than a 64x1 strip; thus, the 8x8 would see less divergence and higher performance.
- When in doubt, a square 16x16 work-group size is a good start.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

6.6.4 Optimizing for Cedar

To focus the discussion, this section has used specific hardware characteristics that apply to most of the Evergreen series. The value Evergreen part, referred to as Cedar and used in products such as the ATI Radeon™ HD 5450 GPU, has different architecture characteristics, as shown below.

	Evergreen Cypress, Juniper, Redwood	Evergreen Cedar
Work-items/Wavefront	64	32
Stream Cores / CU	16	8
GP Registers / CU	16384	8192
Local Memory Size	32K	32K
Maximum Work-Group Size	256	128

Note the maximum workgroup size can be obtained with `clGetDeviceInfo...(...,CL_DEVICE_MAX_WORK_GROUP_SIZE,...).` Applications must ensure that the requested kernel launch dimensions that are fewer than the threshold reported by this API call.

The difference in total register size can impact the compiled code and cause register spill code for kernels that were tuned for other devices. One technique that can be useful is to specify the required work-group size as 128 (half the default of 256). In this case, the compiler has the same number of registers available as for other devices and uses the same number of registers. The developer must ensure that the kernel is launched with the reduced work size (128) on Cedar-class devices.

6.6.5 Summary of NDRange Optimizations

As shown above, execution range optimization is a complex topic with many interacting variables and which frequently requires some experimentation to determine the optimal values. Some general guidelines are:

- Select the work-group size to be a multiple of 64, so that the wavefronts are fully populated.
- Always provide at least two wavefronts (128 work-items) per compute unit. For a ATI Radeon™ HD 5870 GPU, this implies 40 wave-fronts or 2560 work-items. If necessary, reduce the work-group size (but not below 64 work-items) to provide work-groups for all compute units in the system.
- Latency hiding depends on both the number of wavefronts/compute unit, as well as the execution time for each kernel. Generally, two to eight wavefronts/compute unit is desirable, but this can vary significantly, depending on the complexity of the kernel and the available memory bandwidth. The AMD APP Profiler and associated performance counters can help to select an optimal value.

6.7 Using Multiple OpenCL Devices

The AMD OpenCL runtime supports both CPU and GPU devices. This section introduces techniques for appropriately partitioning the workload and balancing it across the devices in the system.

6.7.1 CPU and GPU Devices

Table 6.9 lists some key performance characteristics of two exemplary CPU and GPU devices: a quad-core AMD Phenom II X4 processor running at 2.8 GHz, and a mid-range ATI Radeon™ 5670 GPU running at 750 MHz. The "best" device in each characteristic is highlighted, and the ratio of the best/other device is shown in the final column.

Table 6.9 CPU and GPU Performance Characteristics

	CPU	GPU	Winner Ratio
Example Device	AMD Phenom™ II X4	ATI Radeon™ HD 5670	
Core Frequency	2800 MHz	750 MHz	4 X
Compute Units	4	5	1.3 X
Approx. Power[1]	95 W	64 W	1.5 X
Approx. Power/Compute Unit	19 W	13 W	1.5 X
Peak Single-Precision Billion Floating-Point Ops/Sec	90	600	7 X
Approx GFLOPS/Watt	0.9	9.4	10 X
Max In-flight HW Threads	4	15872	3968 X
Simultaneous Executing Threads	4	80	20 X
Memory Bandwidth	26 GB/s	64 GB/s	2.5 X
Int Add latency	0.4 ns	10.7 ns	30 X
FP Add Latency	1.4 ns	10.7 ns	7 X
Approx DRAM Latency	50 ns	300 ns	6 X
L2+L3 cache capacity	8192 KB	128 kB	64 X
Approx Kernel Launch Latency	25 µs	225 µs	9 X

1. For the power specifications of the AMD Phenom™ II x4, see http://www.amd.com/us/products/desktop/processors/phenom-ii/Pages/phenom-ii-model-number-comparison.aspx. For the power specifications of the ATI Radeon™ HD 5670, see http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/ati-radeon-hd-5670-overview/Pages/ati-radeon-hd-5670-specifications.aspx.

The GPU excels at high-throughput: the peak execution rate (measured in FLOPS) is 7X higher than the CPU, and the memory bandwidth is 2.5X higher than the CPU. The GPU also consumes approximately 65% the power of the CPU; thus, for this comparison, the power efficiency in flops/watt is 10X higher. While power efficiency can vary significantly with different devices, GPUs generally provide greater power efficiency (flops/watt) than CPUs because they optimize for throughput and eliminate hardware designed to hide latency.

Conversely, CPUs excel at latency-sensitive tasks. For example, an integer add is 30X faster on the CPU than on the GPU. This is a product of both the CPUs

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

higher clock rate (2800 MHz vs 750 MHz for this comparison), as well as the operation latency; the CPU is optimized to perform an integer add in just one cycle, while the GPU requires eight cycles. The CPU also has a latency-optimized path to DRAM, while the GPU optimizes for bandwidth and relies on many in-flight threads to hide the latency. The ATI Radeon™ HD 5670 GPU, for example, supports more than 15,000 in-flight threads and can switch to a new thread in a single cycle. The CPU supports only four hardware threads, and thread-switching requires saving and restoring the CPU registers from memory. The GPU requires many active threads to both keep the execution resources busy, as well as provide enough threads to hide the long latency of cache misses.

Each GPU thread has its own register state, which enables the fast single-cycle switching between threads. Also, GPUs can be very efficient at gather/scatter operations: each thread can load from any arbitrary address, and the registers are completely decoupled from the other threads. This is substantially more flexible and higher-performing than a classic Vector ALU-style architecture (such as SSE on the CPU), which typically requires that data be accessed from contiguous and aligned memory locations. SSE supports instructions that write parts of a register (for example, `MOVLPS` and `MOVHPS`, which write the upper and lower halves, respectively, of an SSE register), but these instructions generate additional microarchitecture dependencies and frequently require additional pack instructions to format the data correctly.

In contrast, each GPU thread shares the same program counter with 63 other threads in a wavefront. Divergent control-flow on a GPU can be quite expensive and can lead to significant under-utilization of the GPU device. When control flow substantially narrows the number of valid work-items in a wave-front, it can be faster to use the CPU device.

CPUs also tend to provide significantly more on-chip cache than GPUs. In this example, the CPU device contains 512k L2 cache/core plus a 6 MB L3 cache that is shared among all cores, for a total of 8 MB of cache. In contrast, the GPU device contains only 128 k cache shared by the five compute units. The larger CPU cache serves both to reduce the average memory latency and to reduce memory bandwidth in cases where data can be re-used from the caches.

Finally, note the approximate 9X difference in kernel launch latency. The GPU launch time includes both the latency through the software stack, as well as the time to transfer the compiled kernel and associated arguments across the PCI-express bus to the discrete GPU. Notably, the launch time does not include the time to compile the kernel. The CPU can be the device-of-choice for small, quick-running problems when the overhead to launch the work on the GPU outweighs the potential speedup. Often, the work size is data-dependent, and the choice of device can be data-dependent as well. For example, an image-processing algorithm may run faster on the GPU if the images are large, but faster on the CPU when the images are small.

The differences in performance characteristics present interesting optimization opportunities. Workloads that are large and data parallel can run orders of

magnitude faster on the GPU, and at higher power efficiency. Serial or small parallel workloads (too small to efficiently use the GPU resources) often run significantly faster on the CPU devices. In some cases, the same algorithm can exhibit both types of workload. A simple example is a reduction operation such as a sum of all the elements in a large array. The beginning phases of the operation can be performed in parallel and run much faster on the GPU. The end of the operation requires summing together the partial sums that were computed in parallel; eventually, the width becomes small enough so that the overhead to parallelize outweighs the computation cost, and it makes sense to perform a serial add. For these serial operations, the CPU can be significantly faster than the GPU.

6.7.2 When to Use Multiple Devices

One of the features of GPU computing is that some algorithms can run substantially faster and at better energy efficiency compared to a CPU device. Also, once an algorithm has been coded in the data-parallel task style for OpenCL, the same code typically can scale to run on GPUs with increasing compute capability (that is more compute units) or even multiple GPUs (with a little more work).

For some algorithms, the advantages of the GPU (high computation throughput, latency hiding) are offset by the advantages of the CPU (low latency, caches, fast launch time), so that the performance on either devices is similar. This case is more common for mid-range GPUs and when running more mainstream algorithms. If the CPU and the GPU deliver similar performance, the user can get the benefit of either improved power efficiency (by running on the GPU) or higher peak performance (use both devices).

Usually, when the data size is small, it is faster to use the CPU because the start-up time is quicker than on the GPU due to a smaller driver overhead and avoiding the need to copy buffers from the host to the device.

6.7.3 Partitioning Work for Multiple Devices

By design, each OpenCL command queue can only schedule work on a single OpenCL device. Thus, using multiple devices requires the developer to create a separate queue for each device, then partition the work between the available command queues.

A simple scheme for partitioning work between devices would be to statically determine the relative performance of each device, partition the work so that faster devices received more work, launch all the kernels, and then wait for them to complete. In practice, however, this rarely yields optimal performance. The relative performance of devices can be difficult to determine, in particular for kernels whose performance depends on the data input. Further, the device performance can be affected by dynamic frequency scaling, OS thread scheduling decisions, or contention for shared resources, such as shared caches and DRAM bandwidth. Simple static partitioning algorithms which “guess wrong” at the beginning can result in significantly lower performance, since some

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

devices finish and become idle while the whole system waits for the single, unexpectedly slow device.

For these reasons, a dynamic scheduling algorithm is recommended. In this approach, the workload is partitioned into smaller parts that are periodically scheduled onto the hardware. As each device completes a part of the workload, it requests a new part to execute from the pool of remaining work. Faster devices, or devices which work on easier parts of the workload, request new input faster, resulting in a natural workload balancing across the system. The approach creates some additional scheduling and kernel submission overhead, but dynamic scheduling generally helps avoid the performance cliff from a single bad initial scheduling decision, as well as higher performance in real-world system environments (since it can adapt to system conditions as the algorithm runs).

Multi-core runtimes, such as Cilk, have already introduced dynamic scheduling algorithms for multi-core CPUs, and it is natural to consider extending these scheduling algorithms to GPUs as well as CPUs. A GPU introduces several new aspects to the scheduling process:

- **Heterogeneous Compute Devices**

  Most existing multi-core schedulers target only homogenous computing devices. When scheduling across both CPU and GPU devices, the scheduler must be aware that the devices can have very different performance characteristics (10X or more) for some algorithms. To some extent, dynamic scheduling is already designed to deal with heterogeneous workloads (based on data input the same algorithm can have very different performance, even when run on the same device), but a system with heterogeneous devices makes these cases more common and more extreme. Here are some suggestions for these situations.

  – The scheduler should support sending different workload sizes to different devices. GPUs typically prefer larger grain sizes, and higher-performing GPUs prefer still larger grain sizes.

  – The scheduler should be conservative about allocating work until after it has examined how the work is being executed. In particular, it is important to avoid the performance cliff that occurs when a slow device is assigned an important long-running task. One technique is to use small grain allocations at the beginning of the algorithm, then switch to larger grain allocations when the device characteristics are well-known.

  – As a special case of the above rule, when the devices are substantially different in performance (perhaps 10X), load-balancing has only a small potential performance upside, and the overhead of scheduling the load probably eliminates the advantage. In the case where one device is far faster than everything else in the system, use only the fast device.

  – The scheduler must balance small-grain-size (which increase the adaptiveness of the schedule and can efficiently use heterogeneous devices) with larger grain sizes (which reduce scheduling overhead). Note that the grain size must be large enough to efficiently use the GPU.

- **Asynchronous Launch**

  OpenCL devices are designed to be scheduled asynchronously from a command-queue. The host application can enqueue multiple kernels, flush the kernels so they begin executing on the device, then use the host core for other work. The AMD OpenCL implementation uses a separate thread for each command-queue, so work can be transparently scheduled to the GPU in the background.

  One situation that should be avoided is starving the high-performance GPU devices. This can occur if the physical CPU core, which must re-fill the device queue, is itself being used as a device. A simple approach to this problem is to dedicate a physical CPU core for scheduling chores. The device fission extension (see Section A.7, "cl_ext Extensions," page A-4) can be used to reserve a core for scheduling. For example, on a quad-core device, device fission can be used to create an OpenCL device with only three cores.

  Another approach is to schedule enough work to the device so that it can tolerate latency in additional scheduling. Here, the scheduler maintains a watermark of uncompleted work that has been sent to the device, and refills the queue when it drops below the watermark. This effectively increase the grain size, but can be very effective at reducing or eliminating device starvation. Developers cannot directly query the list of commands in the OpenCL command queues; however, it is possible to pass an event to each `clEnqueue` call that can be queried, in order to determine the execution status (in particular the command completion time); developers also can maintain their own queue of outstanding requests.

  For many algorithms, this technique can be effective enough at hiding latency so that a core does not need to be reserved for scheduling. In particular, algorithms where the work-load is largely known up-front often work well with a deep queue and watermark. Algorithms in which work is dynamically created may require a dedicated thread to provide low-latency scheduling.

- **Data Location**

  Discrete GPUs use dedicated high-bandwidth memory that exists in a separate address space. Moving data between the device address space and the host requires time-consuming transfers over a relatively slow PCI-Express bus. Schedulers should be aware of this cost and, for example, attempt to schedule work that consumes the result on the same device producing it.

  CPU and GPU devices share the same memory bandwidth, which results in additional interactions of kernel executions.

6.7.4 Synchronization Caveats

The OpenCL functions that enqueue work (`clEnqueueNDRangeKernel`) merely enqueue the requested work in the command queue; they do not cause it to begin executing. Execution begins when the user executes a synchronizing command, such as `clFlush` or `clWaitForEvents`. Enqueuing several commands

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

before flushing can enable the host CPU to batch together the command submission, which can reduce launch overhead.

Command-queues that are configured to execute in-order are guaranteed to complete execution of each command before the next command begins. This synchronization guarantee can often be leveraged to avoid explicit `clWaitForEvents()` calls between command submissions. Using `clWaitForEvents()` requires intervention by the host CPU and additional synchronization cost between the host and the GPU; by leveraging the in-order queue property, back-to-back kernel executions can be efficiently handled directly on the GPU hardware.

AMD Evergreen GPUs currently do not support the simultaneous execution of multiple kernels. For efficient execution, design a single kernel to use all the available execution resources on the GPU.

The AMD OpenCL implementation spawns a new thread to manage each command queue. Thus, the OpenCL host code is free to manage multiple devices from a single host thread. Note that `clFinish` is a blocking operation; the thread that calls `clFinish` blocks until all commands in the specified command-queue have been processed and completed. If the host thread is managing multiple devices, it is important to call `clFlush` for each command-queue before calling `clFinish`, so that the commands are flushed and execute in parallel on the devices. Otherwise, the first call to `clFinish` blocks, the commands on the other devices are not flushed, and the devices appear to execute serially rather than in parallel.

For low-latency CPU response, it can be more efficient to use a dedicated spin loop and not call `clFinish()` Calling `clFinish()` indicates that the application wants to wait for the GPU, putting the thread to sleep. For low latency, the application should use `clFlush()`, followed by a loop to wait for the event to complete. This is also true for blocking maps. The application should use non-blocking maps followed by a loop waiting on the event. The following provides sample code for this.

```
if (sleep)
   {
      // this puts host thread to sleep, useful if power is a consideration
         or overhead is not a concern
      clFinish(cmd_queue_);
   }
   else
   {
      // this keeps the host thread awake, useful if latency is a concern
      clFlush(cmd_queue_);
      error_ = clGetEventInfo(event, CL_EVENT_COMMAND_EXECUTION_STATUS,
      sizeof(cl_int), &eventStatus, NULL);
      while (eventStatus > 0)
   {
      error_ = clGetEventInfo(event, CL_EVENT_COMMAND_EXECUTION_STATUS,
      sizeof(cl_int), &eventStatus, NULL);
      Sleep(0);     // be nice to other threads, allow scheduler to find
                       other work if possible
```

```
        // Choose your favorite way to yield, SwitchToThread() for example,
            in place of Sleep(0)
        }
    }
```

6.7.5 GPU and CPU Kernels

While OpenCL provides functional portability so that the same kernel can run on any device, peak performance for each device is typically obtained by tuning the OpenCL kernel for the target device.

Code optimized for the Cypress device (the ATI Radeon™ HD 5870 GPU) typically runs well across other members of the Evergreen family. There are some differences in cache size and LDS bandwidth that might impact some kernels (see Appendix D, “Device Parameters”). The Cedar ASIC has a smaller wavefront width and fewer registers (see Section 6.6.4, “Optimizing for Cedar,” page 6-32, for optimization information specific to this device).

As described in Section 6.9, “Clause Boundaries,” page 6-46, CPUs and GPUs have very different performance characteristics, and some of these impact how one writes an optimal kernel. Notable differences include:

- The Vector ALU floating point resources in a CPU (SSE) require the use of vectorized types (float4) to enable packed SSE code generation and extract good performance from the Vector ALU hardware. The GPU VLIW hardware is more flexible and can efficiently use the floating-point hardware even without the explicit use of float4. See Section 6.8.4, “VLIW and SSE Packing,” page 6-43, for more information and examples; however, code that can use float4 often generates hi-quality code for both the CPU and the AMD GPUs.
- The AMD OpenCL CPU implementation runs work-items from the same work-group back-to-back on the same physical CPU core. For optimally coalesced memory patterns, a common access pattern for GPU-optimized algorithms is for work-items in the same wavefront to access memory locations from the same cache line. On a GPU, these work-items execute in parallel and generate a coalesced access pattern. On a CPU, the first work-item runs to completion (or until hitting a barrier) before switching to the next. Generally, if the working set for the data used by a work-group fits in the CPU caches, this access pattern can work efficiently: the first work-item brings a line into the cache hierarchy, which the other work-items later hit. For large working-sets that exceed the capacity of the cache hierarchy, this access pattern does not work as efficiently; each work-item refetches cache lines that were already brought in by earlier work-items but were evicted from the cache hierarchy before being used. Note that AMD CPUs typically provide 512k to 2 MB of L2+L3 cache for each compute unit.
- CPUs do not contain any hardware resources specifically designed to accelerate local memory accesses. On a CPU, local memory is mapped to the same cacheable DRAM used for global memory, and there is no performance benefit from using the `__local` qualifier. The additional memory operations to write to LDS, and the associated barrier operations can reduce

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

performance. One notable exception is when local memory is used to pack values to avoid non-coalesced memory patterns.

- CPU devices only support a small number of hardware threads, typically two to eight. Small numbers of active work-group sizes reduce the CPU switching overhead, although for larger kernels this is a second-order effect.

For a balanced solution that runs reasonably well on both devices, developers are encouraged to write the algorithm using float4 vectorization. The GPU is more sensitive to algorithm tuning; it also has higher peak performance potential. Thus, one strategy is to target optimizations to the GPU and aim for reasonable performance on the CPU. For peak performance on all devices, developers can choose to use conditional compilation for key code loops in the kernel, or in some cases even provide two separate kernels. Even with device-specific kernel optimizations, the surrounding host code for allocating memory, launching kernels, and interfacing with the rest of the program generally only needs to be written once.

Another approach is to leverage a CPU-targeted routine written in a standard high-level language, such as C++. In some cases, this code path may already exist for platforms that do not support an OpenCL device. The program uses OpenCL for GPU devices, and the standard routine for CPU devices. Load-balancing between devices can still leverage the techniques described in Section 6.7.3, “Partitioning Work for Multiple Devices,” page 6-35.

6.7.6 Contexts and Devices

The AMD OpenCL program creates at least one context, and each context can contain multiple devices. Thus, developers must choose whether to place all devices in the same context or create a new context for each device. Generally, it is easier to extend a context to support additional devices rather than duplicating the context for each device: buffers are allocated at the context level (and automatically across all devices), programs are associated with the context, and kernel compilation (via `clBuildProgram`) can easily be done for all devices in a context. However, with current OpenCL implementations, creating a separate context for each device provides more flexibility, especially in that buffer allocations can be targeted to occur on specific devices. Generally, placing the devices in the same context is the preferred solution.

6.8 Instruction Selection Optimizations

6.8.1 Instruction Bandwidths

Table 6.10 lists the throughput of instructions for GPUs.

Table 6.10 Instruction Throughput (Operations/Cycle for Each Stream Processor)

		Rate (Operations/Cycle) for each Stream Processor	
	Instruction	**Non-Double-Precision-Capable (Evergreen and later) Devices**	**Double-Precision-Capable Devices (Evergreen and later)**
Single Precision FP Rates	SPFP FMA	0	4
	SPFP MAD	5	5
	ADD	5	5
	MUL	5	5
	INV	1	1
	RQSRT	1	1
	LOG	1	1
Double Precision FP Rates	FMA	0	1
	MAD	0	1
	ADD	0	2
	MUL	0	1
	INV (approx.)	0	1
	RQSRT (approx.)	0	1
Integer Instruction Rates	MAD	1	1
	ADD	5	5
	MUL	1	1
	Bit-shift	5	5
	Bitwise XOR	5	5
Conversion	Float-to-Int	1	1
	Int-to-Float	1	1
24-Bit Integer Inst Rates	MAD	5	5
	ADD	5	5
	MUL	5	5

Note that single precision MAD operations have five times the throughput of the double-precision rate, and that double-precision is only supported on the AMD Radeon™ HD69XX devices. The use of single-precision calculation is encouraged, if that precision is acceptable. Single-precision data is also half the size of double-precision, which requires less chip bandwidth and is not as demanding on the cache structures.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Generally, the throughput and latency for 32-bit integer operations is the same as for single-precision floating point operations.

24-bit integer MULs and MADs have five times the throughput of 32-bit integer multiplies. 24-bit unsigned integers are natively supported only on the Evergreen family of devices and later. Signed 24-bit integers are supported only on the Northern Island family of devices and later. The use of OpenCL built-in functions for `mul24` and `mad24` is encouraged. Note that `mul24` can be useful for array indexing operations.

Packed 16-bit and 8-bit operations are not natively supported; however, in cases where it is known that no overflow will occur, some algorithms may be able to effectively pack 2 to 4 values into the 32-bit registers natively supported by the hardware.

The MAD instruction is an IEEE-compliant multiply followed by an IEEE-compliant add; it has the same accuracy as two separate MUL/ADD operations. No special compiler flags are required for the compiler to convert separate MUL/ADD operations to use the MAD instruction.

Table 6.10 shows the throughput for each stream processing core. To obtain the peak throughput for the whole device, multiply the number of stream cores and the engine clock (see Appendix D, "Device Parameters"). For example, according to Table 6.10, a Cypress device can perform two double-precision ADD operations/cycle in each stream core. From Appendix D, "Device Parameters," a ATI Radeon™ HD 5870 GPU has 320 Stream Cores and an engine clock of 850 MHz, so the entire GPU has a throughput rate of (2*320*850 MHz) = 544 GFlops for double-precision adds.

6.8.2 AMD Media Instructions

AMD provides a set of media instructions for accelerating media processing. Notably, the sum-of-absolute differences (SAD) operation is widely used in motion estimation algorithms. For a brief listing and description of the AMD media operations, see the third bullet in Section A.8, "AMD Vendor-Specific Extensions," page A-4.

6.8.3 Math Libraries

OpenCL supports two types of math library operation: `native_function()` and `function()`. Native_functions are generally supported in hardware and can run substantially faster, although at somewhat lower accuracy. The accuracy for the non-native functions is specified in section 7.4 of the *OpenCL Specification*. The accuracy for the native functions is implementation-defined. Developers are encouraged to use the native functions when performance is more important than precision. Table 6.11 lists the native speedup factor for certain functions.

Table 6.11 Native Speedup Factor

Function	Native Speedup Factor
sin()	27.1x
cos()	34.2x
tan()	13.4x
exp()	4.0x
exp2()	3.4x
exp10()	5.2x
log()	12.3x
log2()	11.3x
log10()	12.8x
sqrt()	1.8x
rsqrt()	6.4x
powr()	28.7x
divide()	4.4x

6.8.4 VLIW and SSE Packing

Each stream core in the AMD GPU is programmed with a five-wide (or four-wide, depending on the GPU type) VLIW instruction. Efficient use of the GPU hardware requires that the kernel contain enough parallelism to fill all five processing elements; serial dependency chains are scheduled into separate instructions. A classic technique for exposing more parallelism to the compiler is loop unrolling. To assist the compiler in disambiguating memory addresses so that loads can be combined, developers should cluster load and store operations. In particular, re-ordering the code to place stores in adjacent code lines can improve performance. Figure 6.7 shows an example of unrolling a loop and then clustering the stores.

```
__kernel void loopKernel1A(int loopCount,
                           global float *output,
                           global const float * input)
{
    uint gid = get_global_id(0);

    for (int i=0; i<loopCount; i+=1) {
        float Velm0 = (input[i] * 6.0 + 17.0);
        output[gid+i] = Velm0;
    }
}
```

Figure 6.7 Unmodified Loop

Figure 6.8 is the same loop unrolled 4x.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

```
__kernel void loopKernel2A(int loopCount,
                         global float * output,
                         global const float * input)
{
    uint gid = get_global_id(0);

    for (int i=0; i<loopCount; i+=4) {
        float Velm0 = (input[i] * 6.0 + 17.0);
        output[gid+i] = Velm0;

        float Velm1 = (input[i+1] * 6.0 + 17.0);
        output[gid+i+1] = Velm1;

        float Velm2 = (input[i+2] * 6.0 + 17.0);
        output[gid+i+2] = Velm2;

        float Velm3 = (input[i+3] * 6.0 + 17.0);
        output[gid+i+3] = Velm3;
    }
}
```

Figure 6.8 Kernel Unrolled 4X

Figure 6.9 shows and example of an unrolled loop with clustered stores.

```
__kernel void loopKernel3A(int loopCount,
                         global float *output,
                         global const float * input)
{
    uint gid = get_global_id(0);

    for (int i=0; i<loopCount; i+=4) {
        float Velm0 = (input[i] * 6.0 + 17.0);
        float Velm1 = (input[i+1] * 6.0 + 17.0);
        float Velm2 = (input[i+2] * 6.0 + 17.0);
        float Velm3 = (input[i+3] * 6.0 + 17.0);

        output[gid+i+0] = Velm0;
        output[gid+i+1] = Velm1;
        output[gid+i+2] = Velm2;
        output[gid+i+3] = Velm3;
    }
}
```

Figure 6.9 Unrolled Loop with Stores Clustered

Unrolling the loop to expose the underlying parallelism typically allows the GPU compiler to pack the instructions into the slots in the VLIW word. For best results, unrolling by a factor of at least 5 (perhaps 8 to preserve power-of-two factors) may deliver best performance. Unrolling increases the number of required registers, so some experimentation may be required.

The CPU back-end requires the use of vector types (float4) to vectorize and generate packed SSE instructions. To vectorize the loop above, use float4 for the

array arguments. Obviously, this transformation is only valid in the case where the array elements accessed on each loop iteration are adjacent in memory. The explicit use of float4 can also improve the GPU performance, since it clearly identifies contiguous 16-byte memory operations that can be more efficiently coalesced.

Figure 6.10 is an example of an unrolled kernel that uses float4 for vectorization.

```
__kernel void loopKernel4(int loopCount,
                          global float4 *output,
                          global const float4 * input)
{
    uint gid = get_global_id(0);

    for (int i=0; i<loopCount; i+=1) {
        float4 Velm = input[i] * 6.0 + 17.0;

        output[gid+i] = Velm;
    }
}
```

Figure 6.10 Unrolled Kernel Using float4 for Vectorization

6.8.5 Compiler Optimizations

The OpenCL compiler currently recognizes a few patterns and transforms them into a single instruction. By following these patterns, a developer can generate highly efficient code. The currently accepted patterns are:

- Bitfield extract on signed/unsigned integers.

  `(A >> B) & C ==> [u]bit_extract`

  where
  - B and C are compile time constants,
  - A is a 8/16/32bit integer type, and
  - C is a mask.
- Bitfield insert on signed/unsigned integers

  `((A & B) << C) | ((D & E) << F ==> ubit_insert`

  where
  - B and E have no conflicting bits (`B^E == 0`),
  - B, C, E, and F are compile-time constants, and
  - B and E are masks.
  - The first bit set in B is greater than the number of bits in E plus the first bit set in E, or the first bit set in E is greater than the number of bits in B plus the first bit set in B.
  - If B, C, E, or F are equivalent to the value 0, this optimization is also supported.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

6.9 Clause Boundaries

AMD GPUs groups instructions into clauses. These are broken at control-flow boundaries when:

- the instruction type changes (for example, from FETCH to ALU), or
- if the clause contains the maximum amount of operations (the maximum size for an ALU clause is 128 operations).

ALU and LDS access instructions are placed in the same clause. FETCH, ALU/LDS, and STORE instructions are placed into separate clauses.

The GPU schedules a pair of wavefronts (referred to as the “even” and “odd” wavefront). The even wavefront executes for four cycles (each cycle executes a quarter-wavefront); then, the odd wavefront executes for four cycles. While the odd wavefront is executing, the even wavefront accesses the register file and prepares operands for execution. This fixed interleaving of two wavefronts allows the hardware to efficiently hide the eight-cycle register-read latencies.

With the exception of the special treatment for even/odd wavefronts, the GPU scheduler only switches wavefronts on clause boundaries. Latency within a clause results in stalls on the hardware. For example, a wavefront that generates an LDS bank conflict stalls on the compute unit until the LDS access completes; the hardware does not try to hide this stall by switching to another available wavefront.

ALU dependencies on memory operations are handled at the clause level. Specifically, an ALU clause can be marked as dependent on a FETCH clause. All FETCH operations in the clause must complete before the ALU clause begins execution.

Switching to another clause in the same wavefront requires approximately 40 cycles. The hardware immediately schedules another wavefront if one is available, so developers are encouraged to provide multiple wavefronts/compute unit. The cost to switch clauses is far less than the memory latency; typically, if the program is designed to hide memory latency, it hides the clause latency as well.

The address calculations for FETCH and STORE instructions execute on the same hardware in the compute unit as do the ALU clauses. The address calculations for memory operations consumes the same executions resources that are used for floating-point computations.

- The ISA dump shows the clause boundaries. See the example shown below.

For more information on clauses, see the *AMD Evergreen-Family ISA Microcode And Instructions (v1.0b)* and the *AMD R600/R700/Evergreen Assembly Language Format* documents.

The following is an example disassembly showing clauses. There are 13 clauses in the kernel. The first clause is an ALU clause and has 6 instructions.

```
00 ALU_PUSH_BEFORE: ADDR(32) CNT(13) KCACHE0(CB1:0-15) KCACHE1(CB0:0-15)
   0  x: MOV          R3.x,  KC0[0].x
      y: MOV          R2.y,  KC0[0].y
      z: MOV          R2.z,  KC0[0].z
      w: MOV          R2.w,  KC0[0].w
   1  x: MOV          R4.x,  KC0[2].x
      y: MOV          R2.y,  KC0[2].y
      z: MOV          R2.z,  KC0[2].z
      w: MOV          R2.w,  KC0[2].w
      t: SETGT_INT    R5.x,  PV0.x,  0.0f
   2  t: MULLO_INT    ____,  R1.x,  KC1[1].x
   3  y: ADD_INT      ____,  R0.x,  PS2
   4  x: ADD_INT      R0.x,  PV3.y,  KC1[6].x
   5  x: PREDNE_INT   ____,  R5.x,  0.0f      UPDATE_EXEC_MASK UPDATE_PRED
01 JUMP  POP_CNT(1) ADDR(12)
02 ALU: ADDR(45) CNT(5) KCACHE0(CB1:0-15)
   6  z: LSHL         ____,  R0.x,  (0x00000002, 2.802596929e-45f).x
   7  y: ADD_INT      ____,  KC0[1].x,  PV6.z
   8  x: LSHR         R1.x,  PV7.y,  (0x00000002, 2.802596929e-45f).x
03 LOOP_DX10 i0 FAIL_JUMP_ADDR(11)
04 ALU: ADDR(50) CNT(4)
   9  x: ADD_INT      R3.x,  -1,  R3.x
      y: LSHR         R0.y,  R4.x,  (0x00000002, 2.802596929e-45f).x
      t: ADD_INT      R4.x,  R4.x,  (0x00000004, 5.605193857e-45f).y
05 WAIT_ACK:  Outstanding_acks <= 0
06 TEX: ADDR(64) CNT(1)
   10  VFETCH R0.x___, R0.y, fc156  MEGA(4)
       FETCH_TYPE(NO_INDEX_OFFSET)
07 ALU: ADDR(54) CNT(3)
   11  x: MULADD_e   R0.x,  R0.x,  (0x40C00000, 6.0f).y,  (0x41880000, 17.0f).x
      t: SETE_INT    R2.x,  R3.x,  0.0f
08 MEM_RAT_CACHELESS_STORE_RAW_ACK: RAT(1)[R1].x___, R0, ARRAY_SIZE(4)  MARK  VPM
09 ALU_BREAK: ADDR(57) CNT(1)
   12  x: PREDE_INT   ____,  R2.x,  0.0f      UPDATE_EXEC_MASK UPDATE_PRED
10 ENDLOOP i0 PASS_JUMP_ADDR(4)
11 POP (1) ADDR(12)
12 NOP NO_BARRIER
END_OF_PROGRAM
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

6.10 Additional Performance Guidance

This section is a collection of performance tips for GPU compute and AMD-specific optimizations.

6.10.1 Loop Unroll pragma

The compiler directive #pragma unroll *<unroll-factor>* can be placed immediately prior to a loop as a hint to the compiler to unroll a loop. *<unroll-factor>* must be a positive integer, 1 or greater. When *<unroll-factor>* is 1, loop unrolling is disabled. When *<unroll-factor>* is 2 or greater, the compiler uses this as a hint for the number of times the loop is to be unrolled.

Examples for using this loop follow.

No unrolling example:

```
#pragma unroll 1
for (int i = 0; i < n; i++) {
...
}
```

Partial unrolling example:

```
#pragma unroll 4
for (int i = 0; i < 128; i++) {
...
}
```

Currently, the unroll pragma requires that the loop boundaries can be determined at compile time. Both loop bounds must be known at compile time. If *n* is not given, it is equivalent to the number of iterations of the loop when both loop bounds are known. If the unroll-factor is not specified, and the compiler can determine the loop count, the compiler fully unrolls the loop. If the unroll-factor is not specified, and the compiler cannot determine the loop count, the compiler does no unrolling.

6.10.2 Memory Tiling

There are many possible physical memory layouts for images. AMD Accelerated Parallel Processing devices can access memory in a tiled or in a linear arrangement.

- Linear – A linear layout format arranges the data linearly in memory such that element addresses are sequential. This is the layout that is familiar to CPU programmers. This format must be used for OpenCL buffers; it can be used for images.
- Tiled – A tiled layout format has a pre-defined sequence of element blocks arranged in sequential memory addresses (see Figure 6.11 for a conceptual illustration). A microtile consists of ABIJ; a macrotile consists of the top-left 16 squares for which the arrows are red. Only images can use this format. Translating from user address space to the tiled arrangement is transparent to the user. Tiled memory layouts provide an optimized memory access

pattern to make more efficient use of the RAM attached to the GPU compute device. This can contribute to lower latency.

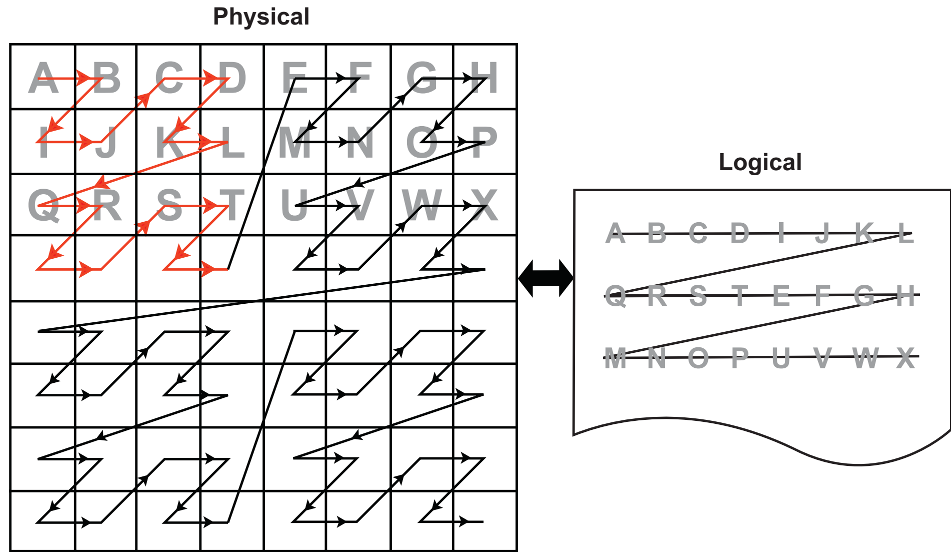

Figure 6.11 One Example of a Tiled Layout Format

Memory Access Pattern –

Memory access patterns in compute kernels are usually different from those in the pixel shaders. Whereas the access pattern for pixel shaders is in a hierarchical, space-filling curve pattern and is tuned for tiled memory performance (generally for textures), the access pattern for a compute kernel is linear across each row before moving to the next row in the global id space. This has an effect on performance, since pixel shaders have implicit blocking, and compute kernels do not. If accessing a tiled image, best performance is achieved if the application tries to use workgroups as a simple blocking strategy.

6.10.3 General Tips

- Avoid declaring global arrays on the kernel's stack frame as these typically cannot be allocated in registers and require expensive global memory operations.
- Use predication rather than control-flow. The predication allows the GPU to execute both paths of execution in parallel, which can be faster than attempting to minimize the work through clever control-flow. The reason for this is that if no memory operation exists in a `?:` operator (also called a ternary operator), this operation is translated into a single `cmov_logical` instruction, which is executed in a single cycle. An example of this is:

```
If (A>B) {
 C += D;
} else {
 C -= D;
}
```

  Replace this with:

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

```
int factor = (A>B) ? 1:-1;
C += factor*D;
```

In the first block of code, this translates into an IF/ELSE/ENDIF sequence of CF clauses, each taking ~40 cycles. The math inside the control flow adds two cycles if the control flow is divergent, and one cycle if it is not. This code executes in ~120 cycles.

In the second block of code, the `?:` operator executes in an ALU clause, so no extra CF instructions are generated. Since the instructions are sequentially dependent, this block of code executes in three cycles, for a ~40x speed improvement. To see this, the first cycle is the (A>B) comparison, the result of which is input to the second cycle, which is the `cmov_logical` factor, bool, 1, -1. The final cycle is a MAD instruction that: mad C, factor, D, C. If the ratio between CF clauses and ALU instructions is low, this is a good pattern to remove the control flow.

- Loop Unrolling
  - OpenCL kernels typically are high instruction-per-clock applications. Thus, the overhead to evaluate control-flow and execute branch instructions can consume a significant part of resource that otherwise can be used for high-throughput compute operations.
  - The AMD Accelerated Parallel Processing OpenCL compiler performs simple loop unrolling optimizations; however, for more complex loop unrolling, it may be beneficial to do this manually.
- If possible, create a reduced-size version of your data set for easier debugging and faster turn-around on performance experimentation. GPUs do not have automatic caching mechanisms and typically scale well as resources are added. In many cases, performance optimization for the reduced-size data implementation also benefits the full-size algorithm.
- When tuning an algorithm, it is often beneficial to code a simple but accurate algorithm that is retained and used for functional comparison. GPU tuning can be an iterative process, so success requires frequent experimentation, verification, and performance measurement.
- The profiler and analysis tools report statistics on a per-kernel granularity. To narrow the problem further, it might be useful to remove or comment-out sections of code, then re-run the timing and profiling tool.
- Writing code with dynamic pointer assignment should be avoided on the GPU. For example:

```
kernel void dyn_assign(global int* a, global int* b, global int* c)
{
        global int* d;
        size_t idx = get_global_id(0);
        if (idx & 1) {
                d = b;
        } else {
                d = c;
        }
        a[idx] = d[idx];
}
```

This is inefficient because the GPU compiler must know the base pointer that every load comes from and in this situation, the compiler cannot determine what ‘d’ points to. So, both B and C are assigned to the same GPU resource, removing the ability to do certain optimizations.

- If the algorithm allows changing the work-group size, it is possible to get better performance by using larger work-groups (more work-items in each work-group) because the workgroup creation overhead is reduced. On the other hand, the OpenCL CPU runtime uses a task-stealing algorithm at the work-group level, so when the kernel execution time differs because it contains conditions and/or loops of varying number of iterations, it might be better to increase the number of work-groups. This gives the runtime more flexibility in scheduling work-groups to idle CPU cores. Experimentation might be needed to reach optimal work-group size.
- Since the AMD OpenCL runtime supports only in-order queuing, using `clFinish()` on a queue and queuing a blocking command gives the same result. The latter saves the overhead of another API command.

  For example:

```
clEnqueueWriteBuffer(myCQ, buff, CL_FALSE, 0, buffSize, input, 0, NULL, NULL);
clFinish(myCQ);
```

  is equivalent, for the AMD OpenCL runtime, to:

```
clEnqueueWriteBuffer(myCQ, buff, CL_TRUE, 0, buffSize, input, 0, NULL, NULL);
```

6.10.4 Guidance for CUDA Programmers Using OpenCL

- Porting from CUDA to OpenCL is relatively straightforward. Multiple vendors have documents describing how to do this, including AMD:

http://developer.amd.com/documentation/articles/pages/OpenCL-and-the-ATI-Stream-v2.0-Beta.aspx#four

- Some specific performance recommendations which differ from other GPU architectures:
  – Use a workgroup size that is a multiple of 64. CUDA code can use a workgroup size of 32; this uses only half the available compute resources on an ATI Radeon™ HD 5870 GPU.
  – Vectorization can lead to substantially greater efficiency. The `ALUPacking` counter provided by the Profiler can track how well the kernel code is using the five-wide (or four-wide, depending on the GPU type) VLIW unit. Values below 70 percent may indicate that dependencies are preventing the full use of the processor. For some kernels, vectorization can be used to increase efficiency and improve kernel performance.
  – AMD GPUs have a very high single-precision flops capability (2.72 teraflops in a single ATI Radeon™ HD 5870 GPU). Algorithms that benefit from such throughput can deliver excellent performance on AMD Accelerated Parallel Processing hardware.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

6.10.5 Guidance for CPU Programmers Using OpenCL to Program GPUs

OpenCL is the industry-standard toolchain for programming GPUs and parallel devices from many vendors. It is expected that many programmers skilled in CPU programming will program GPUs for the first time using OpenCL. This section provides some guidance for experienced programmers who are programming a GPU for the first time. It specifically highlights the key differences in optimization strategy.

- Study the local memory (LDS) optimizations. These greatly affect the GPU performance. Note the difference in the organization of local memory on the GPU as compared to the CPU cache. Local memory is shared by many work-items (64 on Cypress). This contrasts with a CPU cache that normally is dedicated to a single work-item. GPU kernels run well when they collaboratively load the shared memory.
- GPUs have a large amount of raw compute horsepower, compared to memory bandwidth and to “control flow” bandwidth. This leads to some high-level differences in GPU programming strategy.
  – A CPU-optimized algorithm may test branching conditions to minimize the workload. On a GPU, it is frequently faster simply to execute the workload.
  – A CPU-optimized version can use memory to store and later load pre-computed values. On a GPU, it frequently is faster to recompute values rather than saving them in registers. Per-thread registers are a scarce resource on the CPU; in contrast, GPUs have many available per-thread register resources.
- Use `float4` and the OpenCL built-ins for vector types (`vload`, `vstore`, etc.). These enable the AMD Accelerated Parallel Processing OpenCL implementation to generate efficient, packed SSE instructions when running on the CPU. Vectorization is an optimization that benefits both the AMD CPU and GPU.

6.10.6 Optimizing Kernel Code

6.10.6.1 Using Vector Data Types

The CPU contains a vector unit, which can be efficiently used if the developer is writing the code using vector data types.

For architectures before Bulldozer, the instruction set is called SSE, and the vector width is 128 bits. For Bulldozer, there the instruction set is called AVX, for which the vector width is increased to 256 bits.

Using four-wide vector types (int4, float4, etc.) is preferred, even with Bulldozer.

6.10.6.2 Local Memory

The CPU does not benefit much from local memory; sometimes it is detrimental to performance. As local memory is emulated on the CPU by using the caches,

accessing local memory and global memory are the same speed, assuming the information from the global memory is in the cache.

6.10.6.3 Using Special CPU Instructions

The Bulldozer family of CPUs supports FMA4 instructions, exchanging instructions of the form `a*b+c` with `fma(a,b,c)` or `mad(a,b,c)` allows for the use of the special hardware instructions for multiplying and adding.

There also is hardware support for OpenCL functions that give the new hardware implementation of rotating.

For example:

```
sum.x += tempA0.x * tempB0.x + tempA0.y * tempB1.x + tempA0.z * tempB2.x +
tempA0.w * tempB3.x;
```

can be written as a composition of mad instructions which use fused multiple add (FMA):

```
sum.x += mad(tempA0.x, tempB0.x, mad(tempA0.y, tempB1.x, mad(tempA0.z,
tempB2.x, tempA0.w*tempB3.x)));
```

6.10.6.4 Avoid Barriers When Possible

Using barriers in a kernel on the CPU causes a significant performance penalty compared to the same kernel without barriers. Use a barrier only if the kernel requires it for correctness, and consider changing the algorithm to reduce barriers usage.

6.10.7 Optimizing Kernels for Evergreen and 69XX-Series GPUs

6.10.7.1 Clauses

The architecture for the 69XX series of GPUs is clause-based. A clause is similar to a basic block, a sequence of instructions that execute without flow control or I/O. Processor efficiency is determined in large part by the number of instructions in a clause, which is determined by the frequency of branching and I/O at the source-code level. An efficient kernel averages at least 16 or 32 instructions per clause.

The AMD APP KernelAnalyzer assembler listing lets you view clauses. Try the optimizations listed here from inside the AMD APP KernelAnalyzer to see the improvements in performance.

6.10.7.2 Remove Conditional Assignments

A conditional of the form “if-then-else” generates branching and thus generates one or more clauses. Use the `select()` function to replace these structures with conditional assignments that do not cause branching. For example:

```
if(x==1) r=0.5;
if(x==2) r=1.0;
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

becomes

```
r = select(r, 0.5, x==1);
r = select(r, 1.0, x==2);
```

Note that if the body of the `if` statement contains an I/O, the `if` statement cannot be eliminated.

6.10.7.3 Bypass Short-Circuiting

A conditional expression with many terms can compile into a number of clauses due to the C-language requirement that expressions must short circuit. To prevent this, move the expression out of the control flow statement. For example:

```
if(a&&b&&c&&d){…}
```

becomes

```
bool cond = a&&b&&c&&d;
if(cond){…}
```

The same applies to conditional expressions used in loop constructs (`do`, `while`, `for`).

6.10.7.4 Unroll Small Loops

If the loop bounds are known, and the loop is small (less than 16 or 32 instructions), unrolling the loop usually increases performance.

6.10.7.5 Avoid Nested `if`s

Because the GPU is a Vector ALU architecture, there is a cost to executing an if-then-else block because both sides of the branch are evaluated, then one result is retained while the other is discarded. When `if` blocks are nested, the results are twice as bad; in general, if blocks are nested k levels deep, there 2^k clauses are generated. In this situation, restructure the code to eliminate nesting.

6.10.7.6 Experiment With `do`/`while`/`for` Loops

`for` loops can generate more clauses than equivalent `do` or `while` loops. Experiment with these different loop types to find the one with best performance.

6.10.7.7 Do I/O With 4-Word Data

The native hardware I/O transaction size is four words (float4, int4 types). Avoid I/Os with smaller data, and rewrite the kernel to use the native size data. Kernel performance increases, and only 25% as many work items need to be dispatched.

Chapter 7 OpenCL Static C++ Programming Language

7.1 Overview

This extension defines the OpenCL Static C++ kernel language, which is a form of the ISO/IEC Programming languages C++ specification[1]. This language supports overloading and templates that can be resolved at compile time (hence static), while restricting the use of language features that require dynamic/runtime resolving. The language also is extended to support most of the features described in Section 6 of OpenCL spec: new data types (vectors, images, samples, etc.), OpenCL Built-in functions, and more.

7.1.1 Supported Features

The following list contains the major C++ features supported by this extension.

- Kernel and function overloading.
- Inheritance:
  - Strict inheritance.
  - Friend classes.
  - Multiple inheritance.
- Templates:
  - Kernel templates.
  - Member templates.
  - Template default argument.
  - Limited class templates (the `virtual.` keyword is not exposed).
  - Partial template specialization
- Namespaces.
- References.
- `this` operator.

Note that supporting templates and overloading highly improve the efficiency of writing code: it allows developers to avoid replication of code when not necessary.

1. Programming languages C++. International Standard ISO/IEC 14881, 1998.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Using kernel template and kernel overloading requires support from the runtime API as well. AMD provides a simple extension to `clCreateKernel`, which enables the user to specify the desired kernel.

7.1.2 Unsupported Features

C++ features not supported by this extension are:

- Virtual functions (methods marked with the `virtual` keyword).
- Abstract classes (a class defined only of pure virtual functions).
- Dynamic memory allocation (non-placement new/delete support is not provided).
- Exceptions (no support for throw/catch).
- The :: operator.
- STL and other standard C++ libraries.
- The language specified in this extension can be easily expanded to support these features.

7.1.3 Relations with ISO/IEC C++

This extension focuses on documenting the differences between the OpenCL Static C++ kernel language and the ISO/IEC Programming languages C++ specification. Where possible, this extension leaves technical definitions to the ISO/IEC specification.

7.2 Additions and Changes to Section 5 - The OpenCL C Runtime

7.2.1 Additions and Changes to Section 5.7.1 - Creating Kernel Objects

In the static C++ kernel language, a kernel can be overloaded, templated, or both. The syntax explaining how to do it is defined in Sections 7.3.4 and 7.3.5, below.

To support these cases, the following error codes were added; these can be returned by `clCreateKernel`.

- `CL_INVALID_KERNEL_TEMPLATE_TYPE_ARGUMENT_AMD` if a kernel template argument is not a valid type (is neither a valid OpenCL C type or a user defined type in the same source file).
- `CL_INVALID_KERNEL_TYPE_ARGUMENT_AMD` if a kernel type argument, used for overloading resolution, is not a valid type (is neither a valid OpenCL C type or user-defined type in the same source program).

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

7.2.2 Passing Classes between Host and Device

This extension allows a developer to pass classes between the host and the device. The mechanism used to pass the class to the device and back are the existing buffer object APIs. The class that is passed maintains its state (public and private members), and the compiler implicitly changes the class to use either the host-side or device-side methods.

On the host side, the application creates the class and an equivalent memory object with the same size (using the `sizeof` function). It then can use the class methods to set or change values of the class members. When the class is ready, the application uses a standard buffer API to move the class to the device (either Unmap or Write), then sets the buffer object as the appropriate kernel argument and enqueues the kernel for execution. When the kernel finishes the execution, the application can map back (or read) the buffer object into the class and continue working on it.

7.3 Additions and Changes to Section 6 - The OpenCL C Programming Language

7.3.1 Building C++ Kernels

To compile a program that contains C++ kernels and functions, the application must add the following compile option to `clBuildProgramWithSource`:

```
-x language
```

where `language` is defined as one of the following:

- `clc` – the source language is considered to be OpenCL C, as defined in the The OpenCL Programming Language version 1.2[1].
- `clc++` - the source language is considered to be OpenCL C++, as defined in the following sections of the this document.

7.3.2 Classes and Derived Classes

OpenCL C is extended to support classes and derived classes as per Sections 9 and 10 of the C++ language specification, with the limitation that virtual functions and abstracts classes are not supported. The virtual keyword is reserved, and the OpenCL C++ compiler is required to report a compile time error if it is used in the input program.

This limitation restricts class definitions to be fully statically defined. There is nothing prohibiting a future version of OpenCL C++ from relaxing this restriction, pending performance implications.

A class definition can not contain any address space qualifier, either for members or for methods:

1. *The OpenCL Programming Language 1.2. Rev15*, Khronos 2011.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

```
class myClass{
   public:
      int myMethod1(){ return x;}
      void __local myMethod2(){x = 0;}
   private:
      int x;
      __local y; // illegal
};
```

The class invocation inside a kernel, however, can be either in private or local address space:

```
__kernel void myKernel()
{
   myClass c1;
   __local myClass c2;
   ...
}
```

Classes can be passed as arguments to kernels, by defining a buffer object at the size of the class, and using it. The device invokes the adequate device-specific methods, and accesses the class members passed from the host.

OpenCL C kernels (defined with `__kernel`) may not be applied to a class constructor, destructor, or method, except in the case that the class method is defined static and thus does not require object construction to be invoked.

7.3.3 Namespaces

Namespaces are support without change as per [1].

7.3.4 Overloading

As defined in of the C++ language specification, when two or more different declarations are specified for a single name in the same scope, that name is said to be overloaded. By extension, two declarations in the same scope that declare the same name but with different types are called overloaded declarations. Only kernel and function declarations can be overloaded, not object and type declarations.

As per of the C++ language specification, there are a number of restrictions as to how functions can be overloaded; these are defined formally in Section 13 of the C++ language specification. Note that kernels and functions cannot be overloaded by return type.

Also, the rules for well-formed programs as defined by Section 13 of the C++ language specification are lifted to apply to both kernel and function declarations.

The overloading resolution is per Section 13.1 of the C++ language specification, but extended to account for vector types. The algorithm for “best viable function”, Section 13.3.3 of the C++ language specification, is extended for vector types by inducing a partial-ordering as a function of the partial-ordering of its elements. Following the existing rules for vector types in the OpenCL 1.2 specification, explicit conversion between vectors is not allowed. (This reduces the number of

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

possible overloaded functions with respect to vectors, but this is not expected to be a particular burden to developers because explicit conversion can always be applied at the point of function evocation.)

For overloaded kernels, the following syntax is used as part of the kernel name:

```
foo(type1,...,typen)
```

where $type_1, ..., type_n$ must be either an OpenCL scalar or vector type, or can be a user-defined type that is allocated in the same source file as the kernel `foo`.

To allow overloaded kernels, use the following syntax:

```
__attribute__((mangled_name(myMangledName)))
```

The kernel `mangled_name` is used as a parameter to pass to the `clCreateKernel()` API. This mechanism is needed to allow overloaded kernels without changing the existing OpenCL kernel creation API.

7.3.5 Templates

OpenCL C++ provides unrestricted support for C++ templates, as defined in Section 14 of the C++ language specification. The arguments to templates are extended to allow for all OpenCL base types, including vectors and pointers qualified with OpenCL C address spaces (i.e. `__global`, `__local`, `__private`, and `__constant`).

OpenCL C++ kernels (defined with `__kernel`) can be templated and can be called from within an OpenCL C (C++) program or as an external entry point (from the host).

For kernel templates, the following syntax is used as part of the kernel name (assuming a kernel called `foo`):

```
foo<type1,...,typen>
```

where $type_1, ..., type_n$ must be either OpenCL scalar or vector type, or can be a user-defined type that is allocated in the same source file as the kernel `foo`. In this case a kernel is both overloaded and templated:

```
foo<type1,...,typen>(typen+1,...,typem)
```

Note that here overloading resolution is done by first matching non-templated arguments in order of appearance in the definition, then substituting template parameters. This allows intermixing of template and non-template arguments in the signature.

To support template kernels, the same mechanism for kernel overloading is used. Use the following syntax:

```
__attribute__((mangled_name(myMangledName)))
```

The kernel `mangled_name` is used as a parameter to passed to t he `clCreateKernel()` API. This mechanism is needed to allow template kernels

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

without changing the existing OpenCL kernel creation API. An implementation is not required to detect name collision with the user-specified `kernel_mangled` names involved.

7.3.6 Exceptions

Exceptions, as per Section 15 of the C++ language specification, are not supported. The keywords `try`, `catch`, and `throw` are reserved, and the OpenCL C++ compiler must produce a static compile time error if they are used in the input program.

7.3.7 Libraries

Support for the general utilities library, as defined in Sections 20-21 of the C++ language specification, is not provided. The standard C++ libraries and STL library are not supported.

7.3.8 Dynamic Operation

Features related to dynamic operation are not supported:

- the `virtual` modifier.
  OpenCL C++ prohibits the use of the `virtual` modifier. Thus, virtual member functions and virtual inheritance are not supported.
- Dynamic cast that requires runtime check.
- Dynamic storage allocation and deallocation.

7.4 Examples

7.4.1 Passing a Class from the Host to the Device and Back

The class definition must be the same on the host code and the device code, besides the members' type in the case of vectors. If the class includes vector data types, the definition must conform to the table that appears on Section 6.1.2 of the OpenCL Programming Specification 1.2, Corresponding API type for OpenCL Language types.

Example Kernel Code

```
Class Test
{
   setX (int value);
   private:
   int x;
}

__kernel foo (__global Test* InClass, ...)
{
   If (get_global_id(0) == 0)
      InClass->setX(5);
}
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Example Host Code

```
Class Test
{
    setX (int value);
    private:
    int x;
}


MyFunc ()
{
    tempClass = new(Test);
    ... // Some OpenCL startup code – create context, queue, etc.
                        cl_mem classObj = clCreateBuffer(context,
                        CL_USE_HOST_PTR, sizeof(Test),&tempClass,
                        event);
    clEnqueueMapBuffer(...,classObj,...);
    tempClass.setX(10);
    clEnqueueUnmapBuffer(...,classObj,...); //class is passed to the Device
    clEnqueueNDRange(..., fooKernel, ...);
    clEnqueueMapBuffer(...,classObj,...); //class is passed back to the Host

}
```

7.4.2 Kernel Overloading

This example shows how to define and use `mangled_name` for kernel overloading, and how to choose the right kernel from the host code. Assume the following kernels are defined:

```
__attribute__((mangled_name(testAddFloat4))) kernel void
testAdd(global float4 * src1, global float4 * src2, global float4 * dst)
{
    int tid = get_global_id(0);
    dst[tid] = src1[tid] + src2[tid];

}

__attribute__((mangled_name(testAddInt8))) kernel void
    testAdd(global int8 * src1, global int8 * src2, global int8 * dst)
{
    int tid = get_global_id(0);
    dst[tid] = src1[tid] + src2[tid];

}
```

The names `testAddFloat4` and `testAddInt8` are the external names for the two kernel instants. When calling `clCreateKernel`, passing one of these kernel names leads to the correct overloaded kernel.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

7.4.3 Kernel Template

This example defines a kernel template, `testAdd`. It also defines two explicit instants of the kernel template, `testAddFloat4` and `testAddInt8`. The names `testAddFloat4` and `testAddInt8` are the external names for the two kernel template instants that must be used as parameters when calling to the `clCreateKernel` API.

```
template <class T>
    kernel void testAdd(global T * src1, global T * src2, global T * dst)
{
    int tid = get_global_id(0);
    dst[tid] = src1[tid] + src2[tid];
}

template __attribute__((mangled_name(testAddFloat4))) kernel void
    testAdd(global float4 * src1, global float4 * src2, global float4 *
dst);

template __attribute__((mangled_name(testAddInt8))) kernel void
    testAdd(global int8 * src1, global int8 * src2, global int8 * dst);
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Appendix A
OpenCL Optional Extensions

The OpenCL extensions are associated with the devices and can be queried for a specific device. Extensions can be queried for platforms also, but that means that all devices in the platform support those extensions.

Table A.1, on page A-15, lists the supported extensions for the Evergreen-family of devices, as well as for the RV770 and x86 CPUs.

A.1 Extension Name Convention

The name of extension is standardized and must contain the following elements without spaces in the name (in lower case):

- cl_khr_<extension_name> - for extensions approved by Khronos Group. For example: cl_khr_fp64.
- cl_ext_<extension_name> - for extensions provided collectively by multiple vendors. For example: cl_ext_device_fission.
- cl_*<vendor_name>*_*<extension_name>* – for extension provided by a specific vendor. For example: cl_amd_media_ops.

The OpenCL Specification states that all API functions of the extension must have names in the form of cl*<FunctionName>*KHR, cl<FunctionName>EXT, or cl*<FunctionName><VendorName>*. All enumerated values must be in the form of CL_*<enum_name>*_KHR, CL_*<enum_name>*_EXT, or CL_*<enum_name>*_*<VendorName>*.

A.2 Querying Extensions for a Platform

To query supported extensions for the OpenCL platform, use the clGetPlatformInfo() function, with the param_name parameter set to the enumerated value CL_PLATFORM_EXTENSIONS. This returns the extensions as a character string with extension names separated by spaces. To find out if a specific extension is supported by this platform, search the returned string for the required substring.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

A.3 Querying Extensions for a Device

To get the list of devices to be queried for supported extensions, use one of the following:

- Query for available platforms using `clGetPlatformIDs()`. Select one, and query for a list of available devices with `clGetDeviceIDs()`.
- For a specific device type, call `clCreateContextFromType()`, and query a list of devices by calling `clGetContextInfo()` with the param_name parameter set to the enumerated value `CL_CONTEXT_DEVICES`.

After the device list is retrieved, the extensions supported by each device can be queried with function call `clGetDeviceInfo()` with parameter `param_name` being set to enumerated value `CL_DEVICE_EXTENSIONS`.

The extensions are returned in a char string, with extension names separated by a space. To see if an extension is present, search the string for a specified substring.

A.4 Using Extensions in Kernel Programs

There are special directives for the OpenCL compiler to enable or disable available extensions supported by the OpenCL implementation, and, specifically, by the OpenCL compiler. The directive is defined as follows.

```
#pragma OPENCL EXTENSION <extention_name> : <behavior>
#pragma OPENCL EXTENSION all: <behavior>
```

The *<extension_name>* is described in Section A.1, "Extension Name Convention.". The second form allows to address all extensions at once.

The *<behavior>* token can be either:

- `enable` - the extension is enabled if it is supported, or the error is reported if the specified extension is not supported or token "all" is used.
- `disable` - the OpenCL implementation/compiler behaves as if the specified extension does not exist.
- `all` - only core functionality of OpenCL is used and supported, all extensions are ignored. If the specified extension is not supported then a warning is issued by the compiler.

The order of directives in `#pragma OPENCL EXTENSION` is important: a later directive with the same extension name overrides any previous one.

The initial state of the compiler is set to ignore all extensions as if it was explicitly set with the following directive:

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


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

This means that the extensions must be explicitly enabled to be used in kernel programs.

Each extension that affects kernel code compilation must add a defined macro with the name of the extension. This allows the kernel code to be compiled differently, depending on whether the extension is supported and enabled, or not. For example, for extension `cl_khr_fp64` there should be a `#define` directive for macro `cl_khr_fp64`, so that the following code can be preprocessed:

```
#ifdef cl_khr_fp64
    // some code
#else
    // some code
#endif
```

A.5 Getting Extension Function Pointers

Use the following function to get an extension function pointer.

```
void* clGetExtensionFunctionAddress(const char* FunctionName).
```

This returns the address of the extension function specified by the `FunctionName` string. The returned value must be appropriately cast to a function pointer type, specified in the extension spec and header file.

A return value of NULL means that the specified function does not exist in the CL implementation. A non-NULL return value does not guarantee that the extension function actually exists – queries described in sec. 2 or 3 must be done to ensure the extension is supported.

The `clGetExtensionFunctionAddress()` function cannot be used to get core API function addresses.

A.6 List of Supported Extensions that are Khronos-Approved

Supported extensions approved by the Khronos Group are:

- `cl_khr_global_int32_base_atomics` – basic atomic operations on 32-bit integers in global memory.
- `cl_khr_global_int32_extended_atomics` – extended atomic operations on 32-bit integers in global memory.
- `cl_khr_local_int32_base_atomics` – basic atomic operations on 32-bit integers in local memory.
- `cl_khr_local_int32_extended_atomics` – extended atomic operations on 32-bit integers in local memory.
- `cl_khr_int64_base_atomics` – basic atomic operations on 64-bit integers in both global and local memory.
- `cl_khr_int64_extended_atomics` – extended atomic operations on 64-bit integers in both global and local memory.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

- `cl_khr_3d_image_writes` – supports kernel writes to 3D images.
- `cl_khr_byte_addressable_store` – this eliminates the restriction of not allowing writes to a pointer (or array elements) of types less than 32-bit wide in kernel program.
- `cl_khr_gl_sharing` – allows association of OpenGL context or share group with CL context for interoperability.
- `cl_khr_icd` – the OpenCL Installable Client Driver (ICD) that lets developers select from multiple OpenCL runtimes which may be installed on a system. This extension is automatically enabled as of SDK v2 for AMD Accelerated Parallel Processing.
- `cl_khr_d3d10_sharing` - allows association of D3D10 context or share group with CL context for interoperability.

A.7 cl_ext Extensions

- `cl_ext_device_fission` - Support for device fission in OpenCL™. For more information about this extension, see:

  http://www.khronos.org/registry/cl/extensions/ext/cl_ext_device_fission.txt

A.8 AMD Vendor-Specific Extensions

This section describes the AMD vendor-specific extensions.

A.8.1 cl_amd_fp64

Before using double data types, double-precision floating point operators, and/or double-precision floating point routines in OpenCL™ C kernels, include the `#pragma OPENCL EXTENSION cl_amd_fp64 : enable` directive. See Table A.1 for a list of supported routines.

A.8.2 cl_amd_vec3

This extension adds support for vectors with three elements: float3, short3, char3, etc. This data type was added to OpenCL 1.1 as a core feature. For more details, see section 6.1.2 in the OpenCL 1.1 or OpenCL 1.2 spec.

A.8.3 cl_amd_device_persistent_memory

This extension adds support for the new buffer and image creation flag `CL_MEM_USE_PERSISTENT_MEM_AMD`. Buffers and images allocated with this flag reside in host-visible device memory. This flag is mutually exclusive with the flags `CL_MEM_ALLOC_HOST_PTR` and `CL_MEM_USE_HOST_PTR`.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

A.8.4 cl_amd_device_attribute_query

This extension provides a means to query AMD-specific device attributes. To enable this extension, include the `#pragma OPENCL EXTENSION cl_amd_device_attribute_query : enable` directive. Once the extension is enabled, and the `clGetDeviceInfo` parameter `<param_name>` is set to `CL_DEVICE_PROFILING_TIMER_OFFSET_AMD`, the offset in nano-seconds between an event timestamp and Epoch is returned.

A.8.5 cl_amd_device_profiling_timer_offset

This query enables the developer to get the offset between event timestamps in nano-seconds. To use it, compile the kernels with the `#pragma OPENCL EXTENSION cl_amd_device_attribute_query : enable directive`. For kernels complied with this pragma, calling `clGetDeviceInfo` with <*param_name*> set to `CL_DEVICE_PROFILING_TIMER_OFFSET_AMD` returns the offset in nano-seconds between event timestamps.

A.8.6 cl_amd_device_topology

This query enables the developer to get a description of the topology used to connect the device to the host. Currently, this query works only in Linux. Calling `clGetDeviceInfo` with <*param_name*> set to `CL_DEVICE_TOPOLOGY_AMD` returns the following 32-bytes union of structures.

```
typedef union
{
    struct { cl_uint type; cl_uint data[5]; } raw;
    struct { cl_uint type; cl_char unused[17]; cl_char bus; cl_char
device; cl_char function; } pcie; } cl_device_topology_amd;
```

The type of the structure returned can be queried by reading the first unsigned int of the returned data. The developer can use this type to cast the returned union into the right structure type.

Currently, the only supported type in the structure above is PCIe (type value = 1). The information returned contains the PCI Bus/Device/Function of the device, and is similar to the result of the `lspci` command in Linux. It enables the developer to match between the OpenCL device ID and the physical PCI connection of the card.

A.8.7 cl_amd_device_board_name

This query enables the developer to get the name of the GPU board and model of the specific device. Currently, this is only for GPU devices.

Calling `clGetDeviceInfo` with <*param_name*> set to `CL_DEVICE_BOARD_NAME_AMD` returns a 128-character value.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

A.8.8 cl_amd_compile_options

This extension adds the following options, which are not part of the OpenCL specification.

- `-g` — This is an experimental feature that lets you use the GNU project debugger, GDB, to debug kernels on x86 CPUs running Linux or cygwin/minGW under Windows. For more details, see Chapter 3, "Debugging OpenCL." This option does not affect the default optimization of the OpenCL code.
- `-O0` — Specifies to the compiler not to optimize. This is equivalent to the OpenCL standard option `-cl-opt-disable`.
- `-f[no-]bin-source` — Does [not] generate OpenCL source in the `.source` section. For more information, see Appendix E, "OpenCL Binary Image Format (BIF) v2.0."
- `-f[no-]bin-llvmir` — Does [not] generate LLVM IR in the `.llvmir` section. For more information, see Appendix E, "OpenCL Binary Image Format (BIF) v2.0."
- `-f[no-]bin-amdil` — Does [not] generate AMD IL in the `.amdil` section. For more information, see Appendix E, "OpenCL Binary Image Format (BIF) v2.0."
- `-f[no-]bin-exe` — Does [not] generate the executable (ISA) in .text section. For more information, see Appendix E, "OpenCL Binary Image Format (BIF) v2.0."

To avoid source changes, there are two environment variables that can be used to change CL options during the runtime.

- `AMD_OCL_BUILD_OPTIONS` — Overrides the CL options specified in `clBuildProgram()`.
- `AMD_OCL_BUILD_OPTIONS_APPEND` — Appends options to the options specified in `clBuildProgram()`.

A.8.9 cl_amd_offline_devices

To generate binary images offline, it is necessary to access the compiler for every device that the runtime supports, even if the device is currently not installed on the system. When, during context creation, `CL_CONTEXT_OFFLINE_DEVICES_AMD` is passed in the context properties, all supported devices, whether online or offline, are reported and can be used to create OpenCL binary images.

A.8.10 cl_amd_event_callback

This extension provides the ability to register event callbacks for states other than `cl_complete`. The full set of event states are allowed: `cl_queued`, `cl_submitted`, and `cl_running`. This extension is enabled automatically and

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

does not need to be explicitly enabled through `#pragma` when using the SDK v2 of AMD Accelerated Parallel Processing.

A.8.11 `cl_amd_popcnt`

This extension introduces a “population count” function called `popcnt`. This extension was taken into core OpenCL 1.2, and the function was renamed `popcount`. The core 1.2 `popcount` function (documented in section 6.12.3 of the *OpenCL Specification*) is identical to the AMD extension `popcnt` function.

A.8.12 `cl_amd_media_ops`

This extension adds the following built-in functions to the OpenCL language. Note: For OpenCL scalar types, *n* = 1; for vector types, it is {2, 4, 8, or 16}.

Note: in the following, ***n*** denotes the size, which can be 1, 2, 4, 8, or 16; [***i***] denotes the indexed element of a vector, designated 0 to ***n***-1.

Built-in function: `amd_pack`

```
uint amd_pack(float4 src)
```

Return value

```
((((uint)src[0]) & 0xFF) <<  0) +
((((uint)src[1]) & 0xFF) <<  8) +
((((uint)src[2]) & 0xFF) << 16) +
((((uint)src[3]) & 0xFF) << 24)
```

Built-in function: `amd_unpack0`

```
floatn   amd_unpack0 (uintn src)
```

Return value for each vector component

```
(float)(src[i] & 0xFF)
```

Built-in function: `amd_unpack1`

```
floatn   amd_unpack1 (uintn src)
```

Return value for each vector component

```
(float)((src[i] >> 8) & 0xFF)
```

Built-in function: `amd_unpack2`

```
floatn   amd_unpack2 (uintn src)
```

Return value for each vector component

```
(float)((src[i] >> 16) & 0xFF)
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Built-in function: amd_unpack3

```
floatn  amd_unpack3(uintn src)
```

Return value for each vector component

```
(float)((src[i] >> 24) & 0xFF)
```

Built-in function: amd_bitalign

```
uintn  amd_bitalign (uintn src0, uintn src1, uintn src2)
```

Return value for each vector component

```
(uint) (((((long)src0[i]) << 32) | (long)src1[i]) >> (src2[i] & 31))
```

Built-in function: amd_bytealign

```
uintn  amd_bytealign (uintn src0, uintn src1, uintn src2)
```

Return value for each vector component

```
(uint) (((((long)src0[i]) << 32) | (long)src1[i]) >> ((src2[i] & 3)*8))
```

Built-in function: amd_lerp

```
uintn  amd_lerp (uintn src0, uintn src1, uintn src2)
```

Return value for each vector component

```
((((src0[i] >>  0) & 0xFF) + ((src1[i] >>  0) & 0xFF) + ((src2[i] >>  0) & 1)) >> 1) <<  0) +
((((src0[i] >>  8) & 0xFF) + ((src1[i] >>  8) & 0xFF) + ((src2[i] >>  8) & 1)) >> 1) <<  8) +
((((src0[i] >> 16) & 0xFF) + ((src1[i] >> 16) & 0xFF) + ((src2[i] >> 16) & 1)) >> 1) << 16) +
((((src0[i] >> 24) & 0xFF) + ((src1[i] >> 24) & 0xFF) + ((src2[i] >> 24) & 1)) >> 1) << 24) ;
```

Built-in function: amd_sad

```
uintn  amd_sad (uintn src0, uintn src1, uintn src2)
```

Return value for each vector component

```
src2[i] +
abs(((src0[i] >>  0) & 0xFF) - ((src1[i] >>  0) & 0xFF)) +
abs(((src0[i] >>  8) & 0xFF) - ((src1[i] >>  8) & 0xFF)) +
abs(((src0[i] >> 16) & 0xFF) - ((src1[i] >> 16) & 0xFF)) +
abs(((src0[i] >> 24) & 0xFF) - ((src1[i] >> 24) & 0xFF));
```

Built-in function: amd_sad4

```
uint amd_sad4 (uint4 a, uint4 b, uint c)
```

Return value for each vector component

```
src2[i] +
abs(((src0[i] >>  0) & 0xFF) - ((src1[i] >>  0) & 0xFF)) +
abs(((src0[i] >>  8) & 0xFF) - ((src1[i] >>  8) & 0xFF)) +
abs(((src0[i] >> 16) & 0xFF) - ((src1[i] >> 16) & 0xFF)) +
abs(((src0[i] >> 24) & 0xFF) - ((src1[i] >> 24) & 0xFF));
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Built-in function: `amd_sadhi`

```
uintn  amd_sadhi (uintn src0, uintn src1, uintn src2)
```

Return value for each vector component

```
src2[i] +
(abs(((src0[i] >>  0) & 0xFF) - ((src1[i] >>  0) & 0xFF)) << 16) +
(abs(((src0[i] >>  8) & 0xFF) - ((src1[i] >>  8) & 0xFF)) << 16) +
(abs(((src0[i] >> 16) & 0xFF) - ((src1[i] >> 16) & 0xFF)) << 16) +
(abs(((src0[i] >> 24) & 0xFF) - ((src1[i] >> 24) & 0xFF)) << 16);
```

For more information, see:
http://www.khronos.org/registry/cl/extensions/amd/cl_amd_media_ops.txt

A.8.13 cl_amd_media_ops2

This extension adds further built-in functions to those of `cl_amd_media_ops`. When enabled, it adds the following built-in functions to the OpenCL language.

Note: type***n*** denotes an open scalar type { ***n*** = 1 } and vector types { ***n*** = 2, 4, 8, 16 }.

Built-in Function: `uintn amd_msad (uintn src0, uintn src1, uintn src2)`

Description:

```
uchar4 src0u8 = as_uchar4(src0.s0);
uchar4 src1u8 = as_uchar4(src1.s0);
dst.s0 = src2.s0 +
   ((src1u8.s0 == 0) ? 0 : abs(src0u8.s0 - src1u8.s0)) +
   ((src1u8.s1 == 0) ? 0 : abs(src0u8.s1 - src1u8.s1)) +
   ((src1u8.s2 == 0) ? 0 : abs(src0u8.s2 - src1u8.s2)) +
   ((src1u8.s3  == 0) ? 0  : abs(src0u8.s3  - src1u8.s3));
```

A similar operation is applied to other components of the vectors.

Built-in Function: `ulongn amd_qsad (ulongn src0, uintn src1, ulongn src2)`

Description:

```
uchar8 src0u8 = as_uchar8(src0.s0);
ushort4 src2u16 = as_ushort4(src2.s0);
ushort4 dstu16;
dstu16.s0 = amd_sad(as_uint(src0u8.s0123), src1.s0, src2u16.s0);
dstu16.s1 = amd_sad(as_uint(src0u8.s1234), src1.s0, src2u16.s1);
dstu16.s2 = amd_sad(as_uint(src0u8.s2345), src1.s0, src2u16.s2);
dstu16.s3 = amd_sad(as_uint(src0u8.s3456), src1.s0, src2u16.s3);
dst.s0     = as_uint2(dstu16);
```

A similar operation is applied to other components of the vectors.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Built-in Function:

```
ulongn amd_mqsad (ulongn src0, uintn src1, ulongn src2)
```

Description:

```
uchar8 src0u8 = as_uchar8(src0.s0);
ushort4 src2u16 = as_ushort4(src2.s0);
ushort4 dstu16;
dstu16.s0 = amd_msad(as_uint(src0u8.s0123), src1.s0, src2u16.s0);
dstu16.s1 = amd_msad(as_uint(src0u8.s1234), src1.s0, src2u16.s1);
dstu16.s2 = amd_msad(as_uint(src0u8.s2345), src1.s0, src2u16.s2);
dstu16.s3 = amd_msad(as_uint(src0u8.s3456), src1.s0, src2u16.s3);
dst.s0     = as_uint2(dstu16);
```

A similar operation is applied to other components of the vectors.

Built-in Function: `uintn amd_sadw (uintn src0, uintn src1, uintn src2)`

Description:

```
ushort2 src0u16 = as_ushort2(src0.s0);
ushort2 src1u16 = as_ushort2(src1.s0);
dst.s0 = src2.s0 +
   abs(src0u16.s0 - src1u16.s0) +
   abs(src0u16.s1 - src1u16.s1);
```

A similar operation is applied to other components of the vectors.

Built-in Function: `uintn amd_sadd (uintn src0, uintn src1, uintn src2)`

Description:

```
dst.s0 = src2.s0 +  abs(src0.s0 - src1.s0);
```

A similar operation is applied to other components of the vectors.

Built-in Function: `uintn amd_bfm (uintn src0, uintn src1)`

Description:

```
dst.s0 = ((1 << (src0.s0 & 0x1f)) - 1) << (src1.s0 & 0x1f);
```

A similar operation is applied to other components of the vectors.

Built-in Function: `uintn amd_bfe (uintn src0, uintn src1, uintn src2)`

Description:

NOTE: The **>>** operator represents a logical right shift.

```
offset = src1.s0 & 31;
width = src2.s0 & 31;
if width = 0
   dst.s0 = 0;
else if (offset + width) < 32
   dst.s0 = (src0.s0 << (32 - offset - width)) >> (32 - width);
else
   dst.s0 = src0.s0 >> offset;
```

A similar operation is applied to other components of the vectors.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Built-in Function: `intn amd_bfe (intn src0, uintn src1, uintn src2)`

Description:

NOTE: operator **>>** represent an arithmetic right shift.

```
offset = src1.s0 & 31;
width = src2.s0 & 31;
if width = 0
   dst.s0 = 0;
else if (offset + width) < 32
   dst.s0 = src0.s0 << (32-offset-width) >> 32-width;
else
   dst.s0 = src0.s0 >> offset;
```

A similar operation is applied to other components of the vectors.

Built-in Function:

```
intn amd_median3 (intn src0, intn src1, intn src2)
uintn amd_median3 (uintn src0, uintn src1, uintn src2)
floatn amd_median3 (floatn src0, floatn src1, floatn src2)
```

Description:

Returns median of src0, src1, and src2.

Built-in Function:

```
intn amd_min3 (intn src0, intn src1, intn src2)
uintn amd_min3 (uintn src0, uintn src1, uintn src2)
floatn amd_min3 (floatn src0, floatn src1, floatn src2)
```

Description:

Returns min of src0, src1, and src2.

Built-in Function:

```
intn amd_max3 (intn src0, intn src1, intn src2)
uintn amd_max3 (uintn src0, uintn src1, uintn src2)
floatn amd_max3 (floatn src0, floatn src1, floatn src2)
```

Description:

Returns max of src0, src1, and src2.

For more information, see:
http://www.khronos.org/registry/cl/extensions/amd/cl_amd_media_ops2.txt

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

A.8.14 cl_amd_printf

The OpenCL™ Specification 1.1 adds support for the optional AMD extension `cl_amd_printf`, which provides printf capabilities to OpenCL C programs. To use this extension, an application first must include
`#pragma OPENCL EXTENSION cl_amd_printf : enable`.

Built-in function:
`printf(__constant char * restrict format, …);`

This function writes output to the `stdout` stream associated with the host application. The `format` string is a character sequence that:

- is null-terminated and composed of zero and more directives,
- ordinary characters (i.e. not %), which are copied directly to the output stream unchanged, and
- conversion specifications, each of which can result in fetching zero or more arguments, converting them, and then writing the final result to the output stream.

The `format` string must be resolvable at compile time; thus, it cannot be dynamically created by the executing program. (Note that the use of variadic arguments in the built-in `printf` does not imply its use in other built-ins; more importantly, it is not valid to use `printf` in user-defined functions or kernels.)

The OpenCL C printf closely matches the definition found as part of the C99 standard. Note that conversions introduced in the `format` string with `%` are supported with the following guidelines:

- A 32-bit floating point argument is not converted to a 64-bit double, unless the extension `cl_khr_fp64` is supported and enabled, as defined in section 9.3 of the *OpenCL Specification 1.1*. This includes the double variants if `cl_khr_fp64` is supported and defined in the corresponding compilation unit.
- 64-bit integer types can be printed using `%ld / %lx / %lu`.
- `%lld / %llx / %llu` are not supported and reserved for 128-bit integer types (long long).
- All OpenCL vector types (section 6.1.2 of the *OpenCL Specification 1.1*) can be explicitly passed and printed using the modifier `vn`, where `n` can be 2, 3, 4, 8, or 16. This modifier appears before the original conversion specifier for the vector's component type (for example, to print a `float4 %v4f`). Since `vn` is a conversion specifier, it is valid to apply optional flags, such as `field width` and `precision`, just as it is when printing the component types. Since a vector is an aggregate type, the comma separator is used between the components:
`0:1, … , n-2:n-1`.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

A.9 cl_amd_predefined_macros

The following macros are predefined when compiling OpenCL™ C kernels. These macros are defined automatically based on the device for which the code is being compiled.

GPU devices:

```
__WinterPark__
__BeaverCreek__
__Turks__
__Caicos__
__Tahiti__
__Pitcairn__
__Capeverde__
__Cayman__
__Barts__
__Cypress__
__Juniper__
__Redwood__
__Cedar__
__ATI_RV770__
__ATI_RV730__
__ATI_RV710__
__Loveland__
__GPU__
```

CPU devices:

```
__CPU__
__X86__
__X86_64__
```

Note that `__GPU__` or `__CPU__` are predefined whenever a GPU or CPU device is the compilation target.

An example kernel is provided below.

```
#pragma OPENCL EXTENSION cl_amd_printf : enable
const char* getDeviceName() {
#ifdef __Cayman__
        return "Cayman";
#elif __Barts__
        return "Barts";
#elif __Cypress__
        return "Cypress";
#elif defined(__Juniper__)
        return "Juniper";
#elif defined(__Redwood__)
        return "Redwood";
#elif defined(__Cedar__)
        return "Cedar";
#elif defined(__ATI_RV770__)
        return "RV770";
#elif defined(__ATI_RV730__)
        return "RV730";
#elif defined(__ATI_RV710__)
        return "RV710";
#elif defined(__Loveland__)
        return "Loveland";
#elif defined(__GPU__)
        return "GenericGPU";
#elif defined(__X86__)
        return "X86CPU";
#elif defined(__X86_64__)
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

```
        return "X86-64CPU";
#elif defined(__CPU__)
        return "GenericCPU";
#else
        return "UnknownDevice";
#endif
}
kernel void test_pf(global int* a)
{
        printf("Device Name: %s\n", getDeviceName());
}
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

A.10 Supported Functions for `cl_amd_fp64` / `cl_khr_fp64`

AMD OpenCL is now `cl_khr_fp64`-compliant on devices compliant with OpenCL 1.1 and greater (every GPU later than 7xx, and all CPUs). Thus, `cl_amd_fp64` is now a synonym for `cl_khr_fp64` on all supported devices.

A.11 Extension Support by Device

Table A.1 and Table A.2 list the extension support for selected devices.

Table A.1 Extension Support for AMD GPU Devices 1

	A M D APUs			A M D Radeon™ H D				
Extension	**Brazos**	**Llano**	**Trinity**	**Tahiti[1], Pitcairn[2], Cape Verde[3]**	**Turks[4]**	**Cayman[5]**	**Barts[6]**	**Cypress[7]**
cl_khr_*_atomics (32-bit)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
cl_ext_atomic_counters_32	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
cl_khr_gl_sharing	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
cl_khr_byte_addressable_store	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
cl_ext_device_fission	CPU only	CPU only	CPU only	No	No	No	No	No
cl_amd_device_attribute_query	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
cl_khr_fp64	CPU only	CPU only	CPU only	Yes	Yes	Yes	No	Yes
cl_amd_fp64	CPU only	CPU only	CPU only	Yes	Yes	Yes	No	Yes
cl_amd_vec3	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Images	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
cl_khr_d3d10_sharing	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
cl_amd_media_ops	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
cl_amd_printf	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
cl_amd_popcnt	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
cl_khr_3d_image_writes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Platform Extensions								
cl_khr_icd	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
cl_amd_event_callback	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
cl_amd_offline_devices	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

1. AMD Radeon™ HD 79XX series.
2. AMD Radeon™ HD 78XX series.
3. AMD Radeon™ HD 77XX series.
4. AMD Radeon™ HD 75XX series and AMD Radeon™ HD 76XX series.
5. AMD Radeon™ HD 69XX series.
6. AMD Radeon™ HD 68XX series.
7. ATI Radeon™ HD 59XX series and 58XX series, AMD FirePro™ V88XX series and V87XX series.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Note that an atomic counter is a device-level counter that can be added / decremented by different work-items, where the atomicity of the operation is guaranteed. The access to the counter is done only through add/dec built-in functions; thus, no two work-items have the same value returned in the case that a given kernel only increments or decrements the counter. (Also see http://www.khronos.org/registry/cl/extensions/ext/cl_ext_atomic_counters_32.txt.)

Table A.2 Extension Support for Older AMD GPUs and CPUs

Extension	Juniper[1]	Redwood[2]	Cedar[3]	x86 CPU with SSE2 or later
cl_khr_*_atomics	Yes	Yes	Yes	Yes
cl_ext_atomic_counters_32	Yes	Yes	Yes	No
cl_khr_gl_sharing	Yes	Yes	Yes	Yes
cl_khr_byte_addressable_store	Yes	Yes	Yes	Yes
cl_ext_device_fission	No	No	No	Yes
cl_amd_device_attribute_query	Yes	Yes	Yes	Yes
cl_khr_fp64	No	No	No	Yes
cl_amd_fp64[4]	No	No	No	Yes
cl_amd_vec3	Yes	Yes	Yes	Yes
Images	Yes	Yes	Yes	Yes[5]
cl_khr_d3d10_sharing	Yes	Yes	Yes	Yes
cl_amd_media_ops	Yes	Yes	Yes	Yes
cl_amd_media_ops2	Yes	Yes	Yes	Yes
cl_amd_printf	Yes	Yes	Yes	Yes
cl_amd_popcnt	Yes	Yes	Yes	Yes
cl_khr_3d_image_writes	Yes	Yes	Yes	No
Platform Extensions				
cl_khr_icd	Yes	Yes	Yes	Yes
cl_amd_event_callback	Yes	Yes	Yes	Yes
cl_amd_offline_devices	Yes	Yes	Yes	Yes

1. ATI Radeon™ HD 5700 series, AMD Mobility Radeon™ HD 5800 series, AMD FirePro™ V5800 series, AMD Mobility FirePro™ M7820.
2. ATI Radeon™ HD 5600 Series, ATI Radeon™ HD 5600 Series, ATI Radeon™ HD 5500 Series, AMD Mobility Radeon™ HD 5700 Series, AMD Mobility Radeon™ HD 5600 Series, AMD FirePro™ V4800 Series, AMD FirePro™ V3800 Series, AMD Mobility FirePro™ M5800
3. ATI Radeon™ HD 5400 Series, AMD Mobility Radeon™ HD 5400 Series
4. Available on all devices that have double-precision, including all Southern Island devices.
5. Environment variable `CPU_IMAGE_SUPPORT` must be set.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Appendix B
The OpenCL Installable Client Driver (ICD)

The OpenCL Installable Client Driver (ICD) is part of the AMD Accelerated Parallel Processing software stack. Code written prior to SDK v2.0 must be changed to comply with OpenCL ICD requirements.

B.1 Overview

The ICD allows multiple OpenCL implementations to co-exist; also, it allows applications to select between these implementations at runtime.

In releases prior to SDK v2.0, functions such as `clGetDeviceIDs()` and `clCreateContext()` accepted a NULL value for the platform parameter. Releases from SDK v2.0 no longer allow this; the platform must be a valid one, obtained by using the platform API. The application now must select which of the OpenCL platforms present on a system to use.

Use the `clGetPlatformIDs()` and `clGetPlatformInfo()` functions to see the list of available OpenCL implementations, and select the one that is best for your requirements. It is recommended that developers offer their users a choice on first run of the program or whenever the list of available platforms changes.

A properly implemented ICD and OpenCL library is transparent to the end-user.

B.2 Using ICD

Sample code that is part of the SDK contains examples showing how to query the platform API and call the functions that require a valid platform parameter.

This is a pre-ICD code snippet.

```
context = clCreateContextFromType(
            0,
            dType,
            NULL,
            NULL,
            &status);
```

The ICD-compliant version of this code follows.

```
/*
     * Have a look at the available platforms and pick either
     * the AMD one if available or a reasonable default.
     */
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

```
    cl_uint numPlatforms;
    cl_platform_id platform = NULL;
    status = clGetPlatformIDs(0, NULL, &numPlatforms);
    if(!sampleCommon->checkVal(status,
                               CL_SUCCESS,
                               "clGetPlatformIDs failed."))
    {
        return SDK_FAILURE;
    }
    if (0 < numPlatforms)
    {
        cl_platform_id* platforms = new cl_platform_id[numPlatforms];
        status = clGetPlatformIDs(numPlatforms, platforms, NULL);
        if(!sampleCommon->checkVal(status,
                                   CL_SUCCESS,
                                   "clGetPlatformIDs failed."))
        {
            return SDK_FAILURE;
        }
        for (unsigned i = 0; i < numPlatforms; ++i)
        {
            char pbuf[100];
            status = clGetPlatformInfo(platforms[i],
                                       CL_PLATFORM_VENDOR,
                                       sizeof(pbuf),
                                       pbuf,
                                       NULL);


            if(!sampleCommon->checkVal(status,
                                       CL_SUCCESS,
                                       "clGetPlatformInfo failed."))
            {
                return SDK_FAILURE;
            }


            platform = platforms[i];
            if (!strcmp(pbuf, "Advanced Micro Devices, Inc."))
            {
                break;
            }
        }
        delete[] platforms;
    }


    /*
     * If we could find our platform, use it. Otherwise pass a NULL and
get whatever the
     * implementation thinks we should be using.
     */


    cl_context_properties cps[3] =
    {
        CL_CONTEXT_PLATFORM,
        (cl_context_properties)platform,
        0
    };
    /* Use NULL for backward compatibility */
    cl_context_properties* cprops = (NULL == platform) ? NULL : cps;


    context = clCreateContextFromType(
                  cprops,
                  dType,
                  NULL,
                  NULL,
                  &status);
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Another example of a pre-ICD code snippet follows.

```
status = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_DEFAULT, 0, NULL,
&numDevices);
```

The ICD-compliant version of the code snippet is:

```
status = clGetDeviceIDs(platform, CL_DEVICE_TYPE_DEFAULT, 0, NULL,
&numDevices);
```

NOTE: It is recommended that the host code look at the platform vendor string when searching for the desired OpenCL platform, instead of using the platform name string. The platform name string might change, whereas the platform vendor string remains constant for a particular vendor's implementation.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Appendix C Compute Kernel

C.1 Differences from a Pixel Shader

Differences between a *pixel shader* and a compute kernel include: location indexing, amount of resources used on the GPU compute device, memory access patterns, cache behavior, work-item spawn rate, creation of wavefronts and groups, and newly exposed hardware features such as Local Data Store and Shared Registers. Many of these changes are based on the spawn/dispatch pattern of a compute kernel. This pattern is linear; for a pixel shader, it is a hierarchical-Z pattern. The following sections discuss the effects of this change. at the IL level.

C.2 Indexing

A primary difference between a compute kernel and a pixel shader is the indexing mode. In a pixel shader, indexing is done through the *vWinCoord* register and is directly related to the output domain (frame buffer size and geometry) specified by the user space program. This domain is usually in the Euclidean space and specified by a pair of coordinates. In a compute kernel, however, this changes: the indexing method is switched to a linear index between one and three dimensions, as specified by the user. This gives the programmer more flexibility when writing kernels.

Indexing is done through the *vaTid* register, which stands for absolute work-item id. This value is linear: from 0 to N-1, where N is the number of work-items requested by the user space program to be executed on the GPU compute device. Two other indexing variables, *vTid* and *vTgroupid*, are derived from settings in the kernel and *vaTid*.

In SDK 1.4 and later, new indexing variables are introduced for either 3D spawn or 1D spawn. The 1D indexing variables are still valid, but replaced with *vAbsTidFlat*, *vThreadGrpIdFlat*, and *vTidInGrpFlat*, respectively. The 3D versions are *vAbsTid*, *vThreadGrpId*, and *vTidInGrp*. The 3D versions have their respective positions in each dimension in the x, y, and z components. The w component is not used. If the group size for a dimension is not specified, it is an implicit 1. The 1D version has the dimension replicated over all components.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

C.3 Performance Comparison

To show the performance differences between a compute kernel and a pixel shader, the following subsection show a matrix transpose program written in three ways:

1. A naïve pixel shader of matrix transpose.
2. The compute kernel equivalent.
3. An optimized matrix transpose using LDS to further improve performance.

C.4 Pixel Shader

The traditional naïve matrix transpose reads in data points from the (j,i)th element of input matrix in sampler and writes out at the current (i,j)th location, which is implicit in the output write. The kernel is structured as follows:

```
il_ps_2_0
dcl_input_position_interp(linear_noperspective) vWinCoord0.xy__
dcl_output_generic o0
dcl_resource_id(0)_type(2d,unnorm)_fmtx(float)_fmty(float)_fmtz(float)_fmtw(float)
sample_resource(0)_sampler(0) o0, vWinCoord0.yx
end
```

Figure C.1 shows the performance results of using a pixel shader for this matrix transpose.

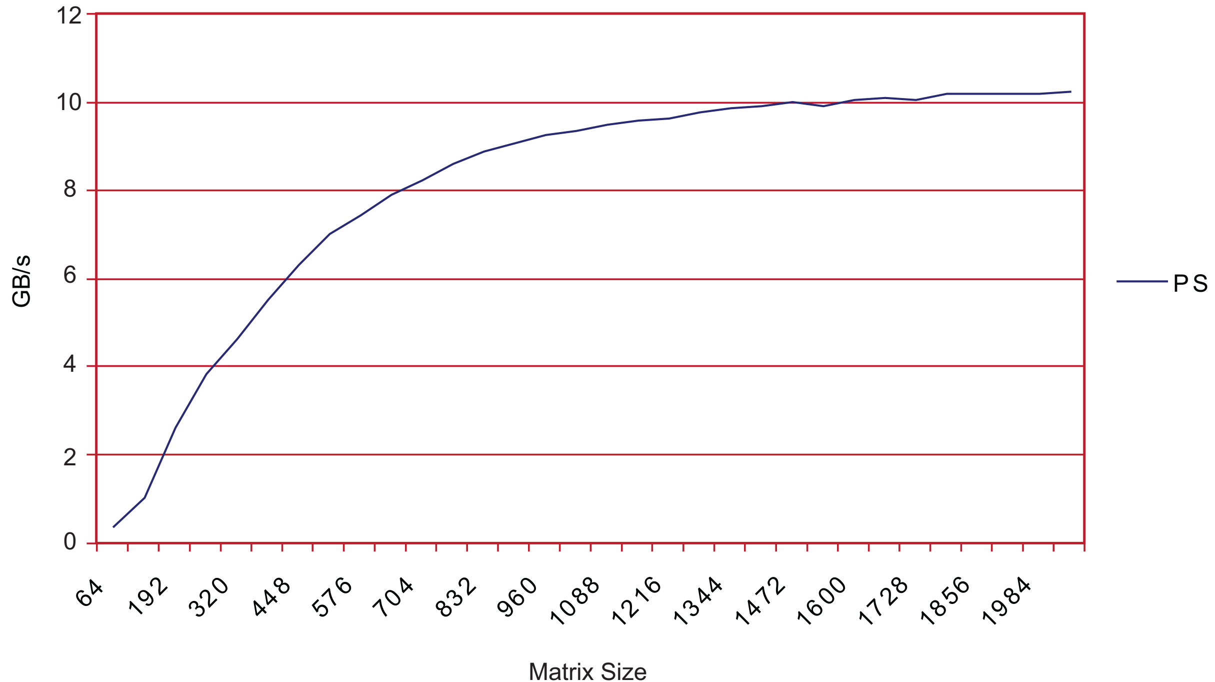

Figure C.1 Pixel Shader Matrix Transpose

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

C.5 Compute Kernel

For the compute kernel, the kernel is structured as follows:

```
il_cs_2_0
dcl_num_threads_per_group 64
dcl_cb cb0[1]
dcl_resource_id(0)_type(2d,unnorm)_fmtx(float)_fmty(float)_fmtz(float)_fmtw(float)
umod r0.x, vAbsTidFlat.x, cb0[0].x
udiv r0.y, vAbsTidFlat.x, cb0[0].x
sample_resource(0)_sampler(0) r1, r0.yx
mov g[vAbsTidFlat.x], r1
end
```

Figure C.2 shows the performance results using a compute kernel for this matrix transpose.

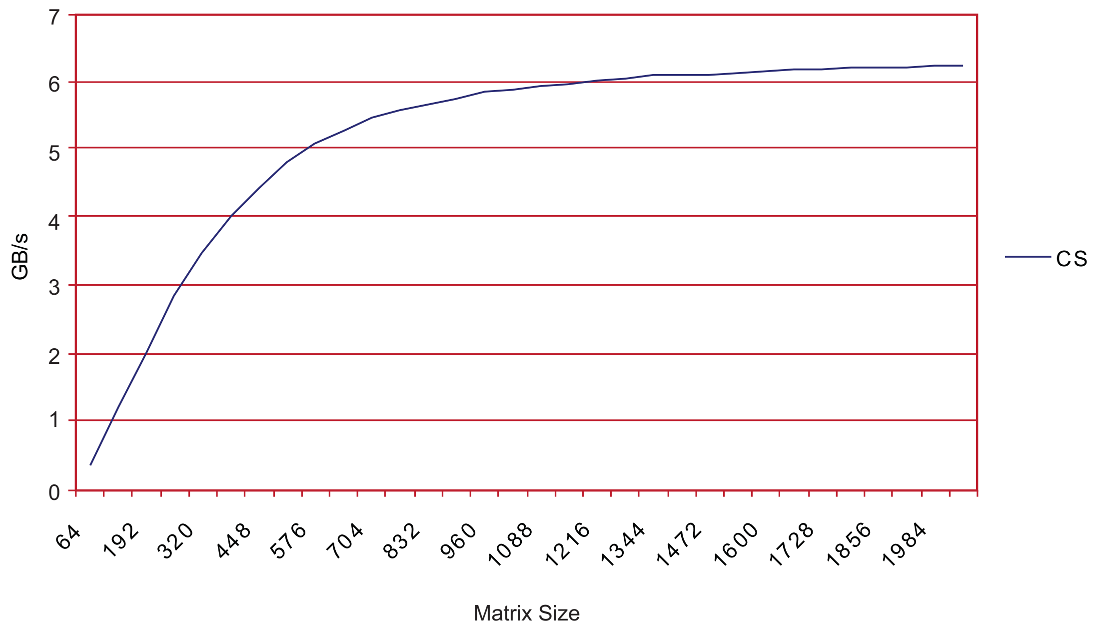

Figure C.2 Compute Kernel Matrix Transpose

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

C.6 LDS Matrix Transpose

Figure C.3 shows the performance results using the LDS for this matrix transpose.

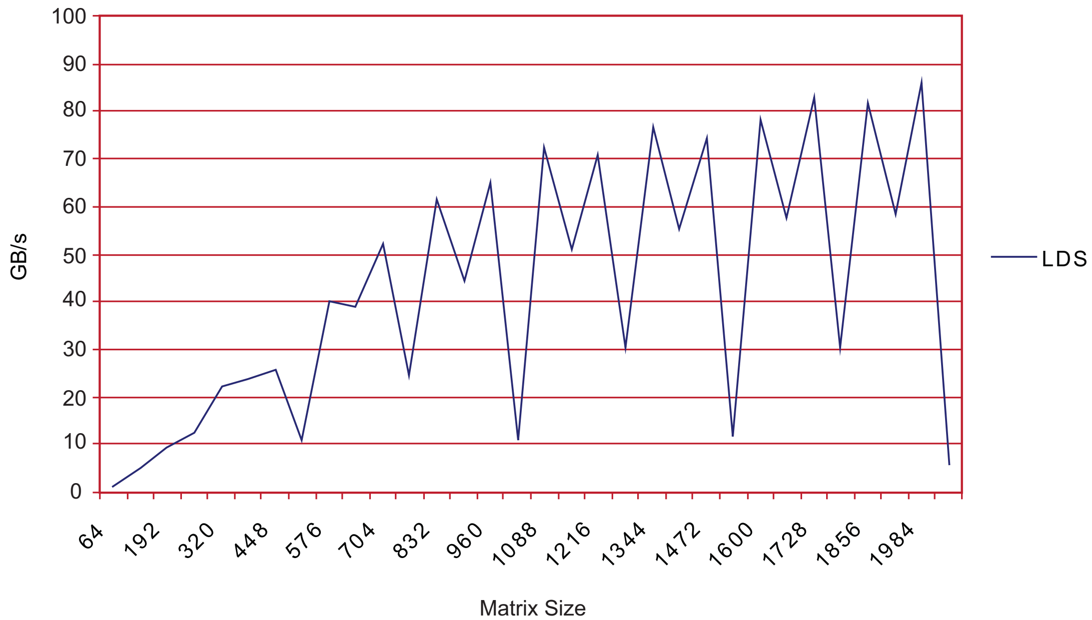

Figure C.3 LDS Matrix Transpose

C.7 Results Comparison

Based on the graphs above, it can be seen that in most cases using the LDS to do on-chip transpose outperforms the similar pixel shader and compute kernel versions; however, a direct porting of the transpose algorithm from a pixel shader to a compute kernel does not immediately increase performance. This is because of the differences mentioned above between the pixel shader and the compute kernel. Taking advantage of the compute kernel features such as LDS can lead to a large performance gain in certain programs.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Appendix D
Device Parameters

On the following pages, Table D.2 through Table D.7 provide device-specific information for AMD GPUs.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Table D.1 Parameters for 7xxx Devices

	Verde PRO	Verde XT	Pitcairn PRO	Pitcairn XT	Tahiti PRO	Tahiti XT
Product Name (AMD Radeon™ HD)	7750	7770	7850	7870	7950	7970
Engine Speed (MHz)	800	1000	860	1000	800	925
Compute Resources						
Compute Units	8	10	16	20	28	32
Processing Elements	512	640	1024	1280	1792	2048
Peak Gflops	819	1280	1761	2560	2867	3789
Cache and Register Sizes						
# of 32b Vector Registers/CU	65536	65536	65536	65536	65536	65536
Size of Vector Registers/CU	256 kB	256 kB	256 kB	256 kB	256 kB	256 kB
LDS Size/ CU	64 kB	64 kB	64 kB	64 kB	64 kB	64 kB
LDS Banks / CU	32	32	32	32	32	32
Constant Cache / GPU	64 kB	64 kB	128 kB	128 kB	128 kB	128 kB
Max Constants / 4 CUs	16 kB	16 kB	16 kB	16 kB	16 kB	16 kB
L1 Cache Size / CU	16 kB	16 kB	16 kB	16 kB	16 kB	16 kB
L2 Cache Size / GPU	512 kB	512 kB	512 kB	512 kB	768 kB	768 kB
Peak GPU Bandwidths						
Register Read (GB/s)	4915	7680	10568	15360	17203	22733
LDS Read (GB/s)	819	1280	1761	2560	2867	3789
Constant Cache Read (GB/s)	102	160	220	320	358	474
L1 Read (GB/s)	410	640	881	1280	1434	1894
L2 Read (GB/s)	205	256	440	512	614	710
Global Memory (GB/s)	72	72	154	154	240	264
Global Limits						
Max Wavefronts / GPU	320	400	640	800	1120	1280
Max Wavefronts / CU (avg)	40	40	40	40	40	40
Max Work-Items / GPU	20480	25600	40960	51200	71680	81920
Memory						
Memory Channels	4	4	8	8	12	12
Memory Bus Width (bits)	128	128	256	256	384	384
Memory Type and Speed (MHz)	GDDR5 1125	GDDR5 1125	GDDR5 1200	GDDR5 1200	GDDR5 1250	GDDR5 1375
Frame Buffer	1 GB	1 GB	2 GB	1 GB or 2 GB	3 GB	3 GB

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Table D.2 Parameters for 68xx and 69xx Devices

	Barts PRO	Barts XT	Blackcomb PRO	Cayman PRO	Cayman XT	Cayman Gemini
Product Name (AMD Radeon™ HD)	6850	6870	6950M	6950	6970	6990
Engine Speed (MHz)	775	900	580	800	880	830 (BIOS 1, default) 880 (BIOS 2)
Compute Resources						
Compute Units	12	14	12	22	24	48
Stream Cores	192	224	192	352	384	768
Processing Elements	960	1120	960	1408	1536	3072
Peak Gflops	1488	2016	1113.6	2252.8	2703.36	5100-5407
Cache and Register Sizes						
# of Vector Registers/CU	16384	16384	16384	16384	16384	16384
Size of Vector Registers/CU	256 kB	256 kB	256 kB	256 kB	256 kB	256 kB
LDS Size/ CU	32 kB	32 kB	32 kB	32 kB	32 kB	32 kB
LDS Banks / CU	32	32	32	32	32	32
Constant Cache / GPU	32 kB	32 kB	32 kB	48 kB	48 kB	48 kB
Max Constants / CU	8 kB	8 kB	8 kB	8 kB	8 kB	8 kB
L1 Cache Size / CU	8 kB	8 kB	8 kB	8 kB	8 kB	8 kB
L2 Cache Size / GPU	512 kB	512 kB	512 kB	512 kB	512 kB	512 kB
Peak GPU Bandwidths						
Register Read (GB/s)	7142	9677	5345	13517	16220	30597-32440
LDS Read (GB/s)	1190	1613	891	2253	2703	5100-5407
Constant Cache Read (GB/s)	2381	3226	1782	4506	5407	10199-10813
L1 Read (GB/s)	595	806	445	1126	1352	2550-2703
L2 Read (GB/s)	397	461	297	410	451	850-901
Global Memory (GB/s)	128	134	115	160	176	320
Global Limits						
Max Wavefronts / GPU	496	496	496	512	512	512
Max Wavefronts / CU (avg)	41.3	35.4	41.3	23.3	21.3	21.3
Max Work-Items / GPU	31744	31744	31744	32768	32768	32768
Memory						
Memory Channels	8	8	8	8	8	8
Memory Bus Width (bits)	256	256	256	256	256	512
Memory Type and Speed (MHz)	GDDR5 1000	GDDR5 1050	GDDR5 900	GDDR5 1250	GDDR5 1375	GDDR5 1250
Frame Buffer	1 GB	1 GB	1 GB	1 GB or 2 GB	2 GB	4 GB total

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Table D.3 Parameters for 65xx, 66xx, and 67xx Devices

	Turks PRO	Turks XT	Whistler LP	Whistler PRO	Whistler XT	Barts LE
Product Name (AMD Radeon™ HD)	6570	6670	6730M	6750M	6770M	6790
Engine Speed (MHz)	650	800	485	600	725	840
Compute Resources						
Compute Units	6	6	6	6	6	10
Stream Cores	96	96	96	96	96	160
Processing Elements	480	480	480	480	480	800
Peak Gflops	624	768	465.6	576	696	1344
Cache and Register Sizes						
# of Vector Registers/CU	16384	16384	16384	16384	16384	16384
Size of Vector Registers/CU	256 kB	256 kB	256 kB	256 kB	256 kB	256 kB
LDS Size/ CU	32 kB	32 kB	32 kB	32 kB	32 kB	32 kB
LDS Banks / CU	32	32	32	32	32	32
Constant Cache / GPU	16 kB	16 kB	16 kB	16 kB	16 kB	32 kB
Max Constants / CU	8 kB	8 kB	8 kB	8 kB	8 kB	8 kB
L1 Cache Size / CU	8 kB	8 kB	8 kB	8 kB	8 kB	8 kB
L2 Cache Size / GPU	256 kB	256 kB	256 kB	256 kB	256 kB	512 kB
Peak GPU Bandwidths						
Register Read (GB/s)	2995	3686	2235	2765	3341	6451
LDS Read (GB/s)	499	614	372	461	557	1075
Constant Cache Read (GB/s)	998	1229	745	922	1114	2150
L1 Read (GB/s)	250	307	186	230	278	538
L2 Read (GB/s)	166	205	124	154	186	430
Global Memory (GB/s)	29 to 64	64	26 to 38	29 to 58	29 to 58	134
Global Limits						
Max Wavefronts / GPU	248	248	248	248	248	496
Max Wavefronts / CU (avg)	41.3	41.3	41.3	41.3	41.3	49.6
Max Work-Items / GPU	15872	15872	15872	15872	15872	31744
Memory						
Memory Channels	4	4	4	4	4	8
Memory Bus Width (bits)	128	128	128	128	128	256
Memory Type and Speed (MHz)	GDDR5, 1000; or DDR3, 900	GDDR5 1000	GDDR5, 600; or DDR3, 800	GDDR5, 800; or DDR3, 900	GDDR5, 900; or DDR3, 900	GDDR5 1050
Frame Buffer	512 MB or 1 GB for GDDR5; 1 or 2 GB for DDR3	512 MB or 1 GB	256 MB	1 GB	1 GB or 2 GB	1 GB

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Table D.4 Parameters for 64xx Devices

	Seymour LP	Seymour PRO	Caicos	Seymour XT	Seymour XTX
Product Name (AMD Radeon™ HD)	6430M	6450M	6450	6470M	6490M
Engine Speed (MHz)	480	600	625 to 750	700 to 750	800
Compute Resources					
Compute Units	2	2	2	2	2
Stream Cores	32	32	32	32	32
Processing Elements	160	160	160	160	160
Peak Gflops	153.6	192	200 to 240	224 to 240	256
Cache and Register Sizes					
# of Vector Registers/CU	16384	16384	16384	16384	16384
Size of Vector Registers/CU	256 kB	256 kB	256 kB	256 kB	256 kB
LDS Size/ CU	32 kB	32 kB	32 kB	32 kB	32 kB
LDS Banks / CU	32	32	32	32	32
Constant Cache / GPU	4 kB	4 kB	4 kB	4 kB	4 kB
Max Constants / CU	8 kB	8 kB	8 kB	8 kB	8 kB
L1 Cache Size / CU	8 kB	8 kB	8 kB	8 kB	8 kB
L2 Cache Size / GPU	128 kB	128 kB	128 kB	128 kB	128 kB
Peak GPU Bandwidths					
Register Read (GB/s)	737	922	960 to 1152	1075 to 1152	1229
LDS Read (GB/s)	123	154	160 to 192	179 to 192	205
Constant Cache Read (GB/s)	246	307	320 to 384	358 to 384	410
L1 Read (GB/s)	61	77	80 to 96	90 to 96	102
L2 Read (GB/s)	61	77	80 to 96	90 to 96	102
Global Memory (GB/s)	13	13	13 to 29	14 to 26	14 to 26
Global Limits					
Max Wavefronts / GPU	192	192	192	192	192
Max Wavefronts / CU (avg)	96.0	96.0	96.0	96.0	96.0
Max Work-Items / GPU	12288	12288	12288	12288	12288
Memory					
Memory Channels	2	2	2	2	2
Memory Bus Width (bits)	64	64	64	64	64
Memory Type and Speed (MHz)	DDR3 800	DDR3 800	GDDR5, 800 - 900; or DDR3, 800	GDDR5, 800; or DDR3, 900	GDDR5, 800; or DDR3, 900
Frame Buffer	512 MB	1 GB	512 MB or 1 GB	512 MB or 1 GB	512 MB or 1 GB

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Table D.5 Parameters for Zacate and Ontario Devices

	Ontario	Ontario	Zacate	Zacate
Product Name (AMD Radeon™ HD)	C-30	C-50	E-240	E-350
Engine Speed (MHz)	277	276	500	492
Compute Resources				
Compute Units	2	2	2	2
Stream Cores	16	16	16	16
Processing Elements	80	80	80	80
Peak Gflops	44.32	44.16	80	78.72
Cache and Register Sizes				
# of Vector Registers/CU	8192	8192	8192	8192
Size of Vector Registers/CU	128 kB	128 kB	128 kB	128 kB
LDS Size/ CU	32 kB	32 kB	32 kB	32 kB
LDS Banks / CU	16	16	16	16
Constant Cache / GPU	4 kB	4 kB	4 kB	4 kB
Max Constants / CU	8 kB	8 kB	8 kB	8 kB
L1 Cache Size / CU	8 kB	8 kB	8 kB	8 kB
L2 Cache Size / GPU	64 kB	64 kB	64 kB	64 kB
Peak GPU Bandwidths				
Register Read (GB/s)	213	212	384	378
LDS Read (GB/s)	35	35	64	63
Constant Cache Read (GB/s)	71	71	128	126
L1 Read (GB/s)	35	35	64	63
L2 Read (GB/s)	35	35	64	63
Global Memory (GB/s)	9	9	9	9
Global Limits				
Max Wavefronts / GPU	192	192	192	192
Max Wavefronts / CU (avg)	96.0	96.0	96.0	96.0
Max Work-Items / GPU	6144	6144	6144	6144
Memory				
Memory Channels	2	2	2	2
Memory Bus Width (bits)	64	64	64	64
Memory Type and Speed (MHz)	DDR3 533	DDR3 533	DDR3 533	DDR3 533
Frame Buffer	Shared Memory	Shared Memory	Shared Memory	Shared Memory

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Table D.6 Parameters for 56xx, 57xx, 58xx, Eyfinity6, and 59xx Devices

	Redwood XT	Juniper LE	Juniper XT	Cypress LE	Cypress PRO	Cypress XT	Hemlock
Product Name (ATI Radeon™ HD)	5670	5750	5770	5830	5850	5870	5970
Engine Speed (MHz)	775	700	850	800	725	850	725
Compute Resources							
Compute Units	5	9	10	14	18	20	40
Stream Cores	80	144	160	224	288	320	640
Processing Elements	400	720	800	1120	1440	1600	3200
Peak Gflops	620	1008	1360	1792	2088	2720	4640
Cache and Register Sizes							
# of Vector Registers/CU	16384	16384	16384	16384	16384	16384	16384
Size of Vector Registers/CU	256 kB	256 kB	256 kB	256 kB	256 kB	256 kB	256 kB
LDS Size/ CU	32 k	32 k	32 k	32 k	32 k	32 k	32 k
LDS Banks / CU	16	32	32	32	32	32	32
Constant Cache / GPU	16 k	24 k	24 k	32 k	40 k	48 k	96 k
Max Constants / CU	8 k	8 k	8 k	8 k	8 k	8 k	8 k
L1 Cache Size / CU	8 k	8 k	8 k	8 k	8 k	8 k	8 k
L2 Cache Size / GPU	128 k	256 k	256 k	512 k	512 k	512 k	2 x 512 k
Peak GPU Bandwidths							
Register Read (GB/s)	2976	4838	6528	8602	10022	13056	22272
LDS Read (GB/s)	248	806	1088	1434	1670	2176	3712
Constant Cache Read (GB/s)	992	1613	2176	2867	3341	4352	7424
L1 Read (GB/s)	248	403	544	717	835	1088	1856
L2 Read (GB/s)	198	179	218	410	371	435	742
Global Memory (GB/s)	64	74	77	128	128	154	256
Global Limits							
Max Wavefronts / GPU	248	248	248	496	496	496	992
Max Wavefronts / CU (avg)	49.6	27.6	24.8	35.4	27.6	24.8	24.8
Max Work-Items / GPU	15872	15872	15872	31744	31744	31744	63488
Memory							
Memory Channels	4	4	4	8	8	8	2 x 8
Memory Bus Width (bits)	128	128	128	256	256	256	2 x 256
Memory Type and Speed (MHz)	GDDR5 1000	GDDR5 1150	GDDR5 1200	GDDR5 1000	GDDR5 1000	GDDR5 1200	GDDR5 1000
Frame Buffer	1 GB / 512 MB	1 GB / 512 MB	1 GB	1 GB	1GB	1 GB	2 x 1 GB

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Table D.7 Parameters for Exxx, Cxx, 54xx, and 55xx Devices

	Zacate	Zacate	Ontario	Ontario	Cedar	Redwood PRO2	Redwood PRO
Product Name (ATI Radeon™ HD)	E-350	E-240	C-50	C-30	5450	5550	5570
Engine Speed (MHz)	492	500	276	277	650	550	650
Compute Resources							
Compute Units	2	2	2	2	2	4	5
Stream Cores	16	16	16	16	16	64	80
Processing Elements	80	80	80	80	80	320	400
Peak Gflops	78.72	80	44.16	44.32	104	352	520
Cache and Register Sizes							
# of Vector Registers/CU	8192	8192	8192	8192	8192	16384	16384
Size of Vector Registers/CU	128 kB	128 kB	128 kB	128 kB	128 kB	256 kB	256 kB
LDS Size/ CU	32 kB	32 kB	32 kB	32 kB	32k	32k	32k
LDS Banks / CU	16	16	16	16	16	16	16
Constant Cache / GPU	4 kB	4 kB	4 kB	4 kB	4k	16k	16k
Max Constants / CU	4 kB	4 kB	4 kB	4 kB	4k	8k	8k
L1 Cache Size / CU	8 kB	8 kB	8 kB	8 kB	8k	8k	8k
L2 Cache Size / GPU	64 kB	64 kB	64 kB	64 kB	64k	128k	128k
Peak GPU Bandwidths							
Register Read (GB/s)	378	384	212	213	499	1690	2496
LDS Read (GB/s)	63	64	35	35	83	141	208
Constant Cache Read (GB/s)	126	128	71	71	166	563	832
L1 Read (GB/s)	63	64	35	35	83	141	208
L2 Read (GB/s)	63	64	35	35	83	141	166
Global Memory (GB/s)	9	9	9	9	13	26	29
Global Limits							
Max Wavefronts / GPU	192	192	192	192	192	248	248
Max Wavefronts / CU (avg)	96.0	96.0	96.0	96.0	96.0	62.0	49.6
Max Work-Items / GPU	6144	6144	6144	6144	6144	15872	15872
Memory							
Memory Channels	2	2	2	2	2	4	4
Memory Bus Width (bits)	64	64	64	64	64	128	128
Memory Type and Speed (MHz)	DDR3 533	DDR3 533	DDR3 533	DDR3 533	DDR3 800	DDR3 800	DDR3 900
Frame Buffer	Shared Memory	Shared Memory	Shared Memory	Shared Memory	1 GB / 512 MB	1 GB / 512 MB	1 GB / 512 MB

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Appendix E OpenCL Binary Image Format (BIF) v2.0

E.1 Overview

OpenCL Binary Image Format (BIF) 2.0 is in the ELF format. BIF2.0 allows the OpenCL binary to contain the OpenCL source program, the LLVM IR, AMD IL, and the executable. The BIF defines the following special sections:

- `.source`: for storing the OpenCL source program.
- `.llvmir`: for storing the OpenCL immediate representation (LLVM IR).
- `.amdil`: for storing the AMD IL that is generated from the OpenCL source program.
- `.comment`: for storing the OpenCL version and the driver version that created the binary.

The BIF can have other special sections for debugging, etc. It also contains several ELF special sections, such as `.text` for storing the executable, `.rodata` for storing the OpenCL runtime control data, and other ELF special sections required for forming an ELF (for example: `.strtab`, `.symtab`, `.shstrtab`).

By default, OpenCL generates a binary that has LLVM IR, AMD IL, and the executable for the GPU (,`.llvmir`, `.amdil`, and `.text` sections), as well as LLVM IR and the executable for the CPU (`.llvmir` and `.text` sections). The BIF binary always contains a `.comment` section, which is a readable C string. The default behavior can be changed with the BIF options described in Section E.2, "BIF Options," page E-3.

The LLVM IR enables recompilation from LLVM IR to the target. When a binary is used to run on a device for which the original program was not generated and the original device is feature-compatible with the current device, OpenCL recompiles the LLVM IR to generate a new code for the device. Note that the LLVM IR is only universal within devices that are feature-compatible in the same device type, not across different device types. This means that the LLVM IR for the CPU is not compatible with the LLVM IR for the GPU. The LLVM IR for a GPU works only for GPU devices that have equivalent feature sets.

BIF2.0 is supported since Stream SDK 2.2.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

E.1.1 Executable and Linkable Format (ELF) Header

For the ELF binary to be considered valid, the AMD OpenCL runtime expects certain values to be specified. The following header fields must be set for all binaries that are created outside of the OpenCL framework.

Table E.1 ELF Header Fields

Field	Value	Description
e_ident[EI_CLASS]	ELFCLASS32, ELFCLASS64	BIF can be either 32-bit ELF or 64bit ELF.
e_ident[EI_DATA]	ELFDATA2LSB	BIF is stored in little Endian order.
e_ident[EI_OSABI]	ELFOSABI_NONE	Not used.
e_ident[EI_ABIVERSION]	0	Not used.
e_type	ET_NONE	Not used.
e_machine	oclElfTargets Enum	CPU/GPU machine ID.
E_version	EV_CURRENT	Must be EV_CURRENT.
e_entry	0	Not used.
E_phoff	0	Not used.
e_flags	0	Not used.
E_phentsize	0	Not used.
E_phnum	0	Not used.

The fields not shown in Table E.1 are given values according to the *ELF Specification*. The e_machine value is defined as one of the oclElfTargets enumerants; the values for these are:

$$e_machine = \begin{cases} 1001 + \text{CaltargetEnum} & : \text{GPU} \\ 2002 & : \text{CPU generic without SSE3} \\ 2003 & : \text{CPU generic with SSE3} \end{cases}$$

```
typedef enum CALtargetEnum {
    CAL_TARGET_600 = 0,           /**< R600 GPU ISA */
    CAL_TARGET_610 = 1,           /**< RV610 GPU ISA */
    CAL_TARGET_630 = 2,           /**< RV630 GPU ISA */
    CAL_TARGET_670 = 3,           /**< RV670 GPU ISA */
    CAL_TARGET_7XX = 4,           /**< R700 class GPU ISA */
    CAL_TARGET_770 = 5,           /**< RV770 GPU ISA */
    CAL_TARGET_710 = 6,           /**< RV710 GPU ISA */
    CAL_TARGET_730 = 7,           /**< RV730 GPU ISA */
    CAL_TARGET_CYPRESS = 8,       /**< CYPRESS GPU ISA */
    CAL_TARGET_JUNIPER = 9,       /**< JUNIPER GPU ISA */
    CAL_TARGET_REDWOOD = 10,      /**< REDWOOD GPU ISA */
    CAL_TARGET_CEDAR= 11,         /**< CEDAR GPU ISA */
    CAL_TARGET_SUMO = 12,         /**< SUMO GPU ISA */
    CAL_TARGET_SUPERSUMO =13,     /**< SUPERSUMO GPU ISA */
    CAL_TARGET_WRESTLER = 14,     /**< WRESTLER GPU ISA */
    CAL_TARGET_CAYMAN =15,        /**< CAYMAN GPU ISA */
    CAL_TARGET_KAUAI = 16,        /**< KAUAI GPU ISA */
    CAL_TARGET_BARTS = 17 ,       /**< BARTS GPU ISA */
    CAL_TARGET_TURKS = 18 ,       /**< TURKS GPU ISA */
    CAL_TARGET_CAICOS  = 19,      /**< CAICOS GPU ISA */
};
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

E.1.2 Bitness

The BIF can be either 32-bit ELF format or a 64-bit ELF format. For the GPU, OpenCL generates a 32-bit BIF binary; it can read either 32-bit BIF or 64-bit BIF binary. For the CPU, OpenCL generates and reads only 32-bit BIF binaries if the host application is 32-bit (on either 32-bit OS or 64-bit OS). It generates and reads only 64-bit BIF binary if the host application is 64-bit (on 64-bit OS).

E.2 BIF Options

OpenCL provides the following options to control what is contained in the binary.

`-f[no-]bin-source` — [not] generate OpenCL source in `.source` section.

`-f[no-]bin-llvmir` — [not] generate LLVM IR in `.llvmir` section.

`-f[no-]bin-amdil` — [not] generate AMD IL in `.amdil` section.

`-f[no-]bin-exe` — [not] generate the executable (ISA) in `.text` section.

The option syntax follows the GCC option syntax. Note that AMD IL is only valid for the GPU. It is ignored if the device type is CPU.

By default, OpenCL generates the `.llvmir` section, `.amdil` section, and `.text` section. The following are examples for using these options:

Example 1: Generate executable for execution:

```
clBuildProgram(program, 0, NULL, "-fno-bin-llvmir -fno-bin-amdil", NULL,
NULL);
```

Example 2: Generate only LLVM IR:

```
clBuildProgram(program, 0, NULL, "-fno-bin-exe -fno-bin-amdil", NULL,
NULL);
```

This binary can recompile for all the other devices of the same device type.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Appendix F
Open Decode API Tutorial

F.1 Overview

This section provides a basic tutorial for using the sample program for Open Decode. The Open Decode API provides the ability to access the hardware for fixed-function decoding using the AMD Unified Video Decoder block on the GPU for decoding H.264 video.

The AMD sample Open Video Decode, provided at http://developer.amd.com/zones/OpenCLZone/pages/openclappexamples.aspx shows how to read in compressed H.264 video elementary stream frames and supporting parameters, then call the hardware decoder to decompress the video.

The following is an introduction for the Open CL programmer to start using UVD hardware; it shows how to perform a decode using the Open Video Decode API.

Open Decode allows the decompression to take place on the GPU, where the Open CL buffers reside. This lets applications perform post-processing operations on the decompressed data on the GPU prior to rendering the frames.

Figure F.1 diagrams an example of an optional Gaussian Blur operation on raw decoded-data. The video is operated on using the GPU compute shader units. The finished video then is displayed using the OpenGL rendering pipeline. Note that OpenCL post-processing takes place on buffers in GPU memory; these buffers do not have to be copied to the CPU address space.

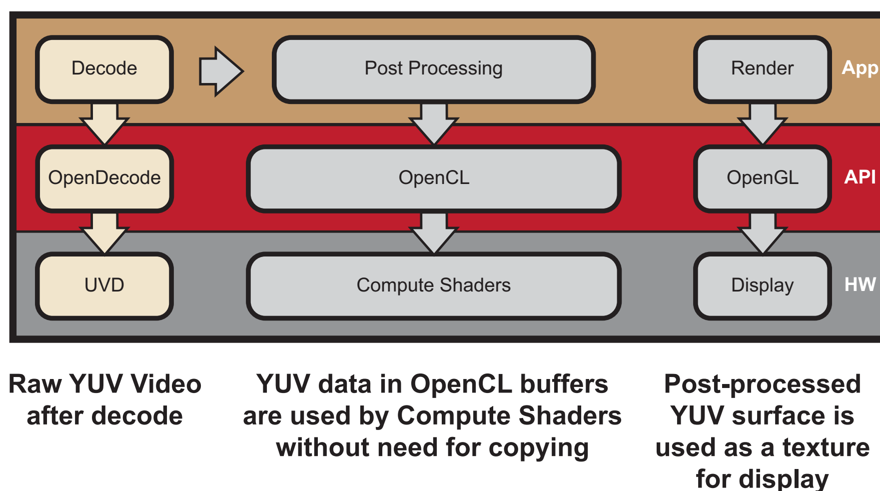

Figure F.1 Open Decode with Optional Post-Processing

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

The five-step process consists of initialization, context creation, session creation, decode execution, and finally session and context destruction.

F.2 Initializing

The first step in using the Open Decode is to get the *Device Info and capabilities* through `OVDecodeGetDeviceInfo` and `OVDecodeGetDeviceCap`. `OVDecodeGetDeviceInfo` obtains the information about the device(s) and initializes the UVD hardware and firmware. As a result of the call to `OVDecodeGetDeviceCaps`, the `deviceInfo` data structure provides the supported output format and the compression profile. The application then can verify that these values support the requested decode. The following code snippet shows the use of OVDecodeGetDeviceInfo and OVDecodeGetDeviceCap.

```
ovdecode_device_info *deviceInfo = new ovdecode_device_info[numDevices];
    status = OVDecodeGetDeviceInfo(&numDevices, deviceInfo);

    unsigned int ovDeviceID = 0;
    for(unsigned int i = 0; i < numDevices; i++)
    {
        ovdecode_cap *caps = new
        ovdecode_cap[deviceInfo[i].decode_cap_size];
        status = OVDecodeGetDeviceCap(deviceInfo[i].device_id,
                                      deviceInfo[i].decode_cap_size,
                                      caps);

        for(unsigned int j = 0; j < deviceInfo[i].decode_cap_size; j++)
        {
            if(caps[j].output_format == OVD_NV12_INTERLEAVED_AMD &&
               caps[j].profile == OVD_H264_HIGH_41)
            {
       ovDeviceID = deviceInfo[i].device_id;
                break;
            }
        }
    }
```

F.3 Creating the Context

The second step is to create the context for the decode session within Open CL using **`clCreateContext`** (see following code snippet). The context creation (tied to OpenCL queue creation) initializes the Open Decode function pointers, callbacks, etc. Context creation also allocates buffers, such as timestamps and/or synchronization buffers needed for the decode. (See Example Code 1 on page 1-21 for `clCreateContext` usage.)

```
intptr_t properties[] =
    {
     CL_CONTEXT_PLATFORM, (cl_context_properties)platform,
           0
};

ovdContext = clCreateContext(properties,
                             1,
                             &clDeviceID,
                             0,
                             0,
                             &err);
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

F.4 Creating the Session

The third step is to create the decode session using **OVDecodeCreateSession**. This function, as shown in the following code snippet, creates the decode driver session and performs the internal resource allocation.

From the capabilities that you have confirmed above in step 1, you now specify the decode profile (H.264) and the output format (NV12 Interleaved). The height and width also are specified. This can be obtained by parsing the data from the input stream.

```
ovdecode_profile profile = OVD_H264_HIGH_41;
ovdecode_format  oFormat = OVD_NV12_INTERLEAVED_AMD;
oWidth  = video_width;
oHeight = video_height;

session = OVDecodeCreateSession(
     ovdContext,
     ovDeviceID,
     profile,
     oFormat,
     oWidth,
     oHeight);
```

F.5 Decoding

Decode execution goes through OpenCL and starts the UVD decode function.

For each OpenCL command sent to the Command Queue, a unique Event_Object is returned. These Event_Objects are useful for forming synchronization points and can be placed in Event Wait Lists by the programmer. If the programmer creates an Event Wait List, each Event_Object in the list must be completed, in list order, before this command is executed.

The Create Command Queue call, `clCreateCommandQueue`, sets up a queue into which, later in the program, OpenCL commands are sent (for example, to run the various OpenCL kernels).

```
cl_cmd_queue = clCreateCommandQueue((cl_context)ovdContext,
                 clDeviceID,
                 0,
                 &err);


This section demonstrates how the Frame info set up can be done with the
information read in from the video frame parameters.


    slice_data_control_size = sliceNum * sizeof(ovd_slice_data_control);
    slice_data_control      =
(ovd_slice_data_control*)malloc(slice_data_control_size);
    pic_parameter_2_size    = sizeof(H264_picture_parameter_2);
    num_event_in_wait_list  = 0;
    bitstream_data_max_size = video_width*video_height*3/2;
    bitstream_data          =
(ovd_bitstream_data)malloc(bitstream_data_max_size);
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

```
    // Size of NV12 format
    int host_ptr_size = oHeight * video_pitch * 3/2;
    host_ptr = malloc(host_ptr_size);

      Create output buffer:

    output_surface = clCreateBuffer((cl_context)ovdContext,
                 CL_MEM_READ_WRITE,
                 host_ptr_size,
                 NULL,
                 &err);

     The sample demonstrates how data can be read to provide Open Decode
with the information needed. Details can be obtained by reviewing the
sample routine 'ReadPictureData' to fill in the values needed to send into
the OvDecodePicture.

ReadPictureData(iFramesDecoded,
                 &picture_parameter,
                          &pic_parameter_2,
                          pic_parameter_2_size,
                          bitstream_data,
                          &bitstream_data_read_size,
                          bitstream_data_max_size,
                          slice_data_control,
                          slice_data_control_size);

     This OVDecoePicture call performs the operation of decoding the frame
and placing the output in the output surface buffer. The OVDecodepicture
is called in a loop until the end of the input stream is reached.

     OPEventHandle eventRunVideoProgram;
     OVresult res = OVDecodePicture(session,
                 &picture_parameter,
                 &pic_parameter_2,
                 pic_parameter_2_size,
                 &bitstream_data,
                 bitstream_data_read_size,
                 slice_data_control,
                 slice_data_control_size,
                 output_surface,
                 num_event_in_wait_list,
                 NULL,
                 &eventRunVideoProgram,
                 0);

Wait until the Decode session completes:

err = clWaitForEvents(1, (cl_event *)&(eventRunVideoProgram));
if(err != CL_SUCCESS)
{
    std::cout <<
    return false;
}
```

F.6 Destroying Session and Context

The final step is to release the resources and close the session. This is done by releasing all the allocated memory and structures, as well as calling `OVDecodeDestroySession` and `clReleaseContext`. These functions cause the decode session to free allocation of resources needed for the session. This frees

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

driver session and all internal resources; it also sets the UVD clock to idle state. The following code snippet shows how this is done.

```
err = clReleaseMemObject((cl_mem)output_surface);

bool ovdErr = OVDecodeDestroySession(session);

err = clReleaseContext((cl_context)ovdContext);
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Appendix G OpenCL-OpenGL Interoperability

This chapter explains how to establish an association between GL context and CL context.

Please note the following guidelines.

1. All devices used to create the OpenCL context associated with `command_queue` must support acquiring shared CL/GL objects. This constraint is enforced at context-creation time.
2. `clCreateContext` and `clCreateContextFromType` fail context creation if the device list passed in cannot interoperate with the GL context. `clCreateContext` only permits GL-friendly device(s). `clCreateFromContextType` can only include GL-friendly device(s).
3. Use `clGetGLContextInfoKHR` to determine GL-friendly device(s) from the following parameters:
   a. `CL_CURRENT_DEVICE_FOR_GL_CONTEXT_KHR` only returns the device that can interoperate with the GL context.
   b. `CL_DEVICES_FOR_GL_CONTEXT_KHR` includes all GL-context interoperable devices.
4. While it is possible to create as many GL contexts on a GPU, do not create concurrently two GL contexts for two GPUs from the same process.

G.1 Under Windows

This sections discusses CL-GL interoperability for single and multiple GPU systems running under Windows. Please note the following guidelines.

1. All devices used to create the OpenCL context associated with `command_queue` must support acquiring shared CL/GL objects. This constraint is enforced at context-creation time.
2. `clCreateContext` and `clCreateContextFromType` fail context creation if the device list passed in cannot interoperate with the GL context. `clCreateContext` only permits GL-friendly device(s). `clCreateFromContextType` can only include GL-friendly device(s).
3. Use `clGetGLContextInfoKHR` to determine GL-friendly device(s) from the following parameters:
   a. `CL_CURRENT_DEVICE_FOR_GL_CONTEXT_KHR` only returns the device that can interoperate with the GL context.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

b. CL_DEVICES_FOR_GL_CONTEXT_KHR includes all GL-context interoperable devices.

G.1.1 Single GPU Environment

G.1.1.1 Creating CL Context from a GL Context

Use **GLUT** windowing system or Win32 API for event handling.

Using GLUT

1. Use glutInit to initialize the GLUT library and negotiate a session with the windowing system. This function also processes the command line options, depending on the windowing system.
2. Use wglGetCurrentContext to get the current rendering GL context (HGLRC) of the calling thread.
3. Use wglGetCurrentDC to get the device context (HDC) that is associated with the current OpenGL rendering context of the calling thread.
4. Use the clGetGLContextInfoKHR *(*See Section 9.7 of the *OpenCL Specification 1.1)* function and the CL_CURRENT_DEVICE_FOR_GL_CONTEXT_KHR parameter to get the device ID of the CL device associated with OpenGL context.
5. Use clCreateContext *(*See Section 4.3 of the *OpenCL Specification 1.1)* to create the CL context (of type cl_context).

The following code snippet shows you how to create an interoperability context using GLUT on single GPU system.

```
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
glutInitWindowSize(WINDOW_WIDTH, WINDOW_HEIGHT);
glutCreateWindow("OpenCL SimpleGL");

HGLRC glCtx = wglGetCurrentContext();

Cl_context_properties cpsGL[] =
{CL_CONTEXT_PLATFORM,(cl_context_properties)platform,
CL_WGL_HDC_KHR, (intptr_t) wglGetCurrentDC(),

CL_GL_CONTEXT_KHR, (intptr_t) glCtx, 0};

status = clGetGLContextInfoKHR(cpsGL,
                CL_CURRENT_DEVICE_FOR_GL_CONTEXT_KHR,
                sizeof(cl_device_id),
                &interopDevice,
                NULL);

// Create OpenCL context from device's id
                context = clCreateContext(cpsGL,
                1,
                &interopDevice,
                0,
                0,
                &status);
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Using Win32 API

1. Use `CreateWindow` for window creation and get the device handle (HWND).
2. Use `GetDC` to get a handle to the device context for the client area of a specific window, or for the entire screen (OR). Use `CreateDC` function to create a device context (HDC) for the specified device.
3. Use `ChoosePixelFormat` to match an appropriate pixel format supported by a device context and to a given pixel format specification.
4. Use `SetPixelFormat` to set the pixel format of the specified device context to the format specified.
5. Use `wglCreateContext` to create a new OpenGL rendering context from device context (HDC).
6. Use `wglMakeCurrent` to bind the GL context created in the above step as the current rendering context.
7. Use `clGetGLContextInfoKHR` function (see Section 9.7 of the *OpenCL Specification 1.1*) and parameter `CL_CURRENT_DEVICE_FOR_GL_CONTEXT_KHR` to get the device ID of the CL device associated with OpenGL context.
8. Use `clCreateContext` function (see Section 4.3 of the *OpenCL Specification 1.1*) to create the CL context (of type `cl_context`).

The following code snippet shows how to create an interoperability context using WIN32 API for windowing. (Users also can refer to the SimpleGL sample in the AMD APP SDK samples.)

```
int  pfmt;
PIXELFORMATDESCRIPTOR  pfd;
pfd.nSize            = sizeof(PIXELFORMATDESCRIPTOR);
pfd.nVersion         = 1;
pfd.dwFlags          = PFD_DRAW_TO_WINDOW |
                           PFD_SUPPORT_OPENGL | PFD_DOUBLEBUFFER ;
pfd.iPixelType       = PFD_TYPE_RGBA;
pfd.cColorBits       = 24;
pfd.cRedBits         = 8;
pfd.cRedShift        = 0;
pfd.cGreenBits       = 8;
pfd.cGreenShift      = 0;
pfd.cBlueBits        = 8;
pfd.cBlueShift       = 0;
pfd.cAlphaBits       = 8;
pfd.cAlphaShift      = 0;
pfd.cAccumBits       = 0;
pfd.cAccumRedBits    = 0;
pfd.cAccumGreenBits  = 0;
pfd.cAccumBlueBits   = 0;
pfd.cAccumAlphaBits  = 0;
pfd.cDepthBits       = 24;
pfd.cStencilBits     = 8;
pfd.cAuxBuffers      = 0;
pfd.iLayerType       = PFD_MAIN_PLANE;
pfd.bReserved        = 0;
pfd.dwLayerMask      = 0;
pfd.dwVisibleMask    = 0;
pfd.dwDamageMask     = 0;

ZeroMemory(&pfd, sizeof(PIXELFORMATDESCRIPTOR));

WNDCLASS windowclass;
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

```
windowclass.style = CS_OWNDC;
windowclass.lpfnWndProc = WndProc;
windowclass.cbClsExtra = 0;
windowclass.cbWndExtra = 0;
windowclass.hInstance = NULL;
windowclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
windowclass.hCursor = LoadCursor(NULL, IDC_ARROW);
windowclass.hbrBackground = (HBRUSH)GetStockObject(BLACK_BRUSH);
windowclass.lpszMenuName = NULL;
windowclass.lpszClassName = reinterpret_cast<LPCSTR>("SimpleGL");
         RegisterClass(&windowclass);

gHwnd = CreateWindow(reinterpret_cast<LPCSTR>("SimpleGL"),
            reinterpret_cast<LPCSTR>("SimpleGL"),
            WS_CAPTION | WS_POPUPWINDOW | WS_VISIBLE,
            0,
            0,
            screenWidth,
            screenHeight,
            NULL,
            NULL,
            windowclass.hInstance,
            NULL);

hDC = GetDC(gHwnd);

pfmt = ChoosePixelFormat(hDC, &pfd);

ret = SetPixelFormat(hDC, pfmt, &pfd);

hRC = wglCreateContext(hDC);

ret = wglMakeCurrent(hDC, hRC);

cl_context_properties properties[] =
{
    CL_CONTEXT_PLATFORM,
    (cl_context_properties) platform,
    CL_GL_CONTEXT_KHR,    (cl_context_properties) hRC,
    CL_WGL_HDC_KHR,       (cl_context_properties) hDC,
    0
};

status = clGetGLContextInfoKHR(properties,
            CL_CURRENT_DEVICE_FOR_GL_CONTEXT_KHR,
            sizeof(cl_device_id),
            &interopDevice,
            NULL);

// Create OpenCL context from device's id
context = clCreateContext(properties,
            1,
            &interopDevice,
            0,
            0,
            &status);
```

G.1.2 Multi-GPU Environment

G.1.2.1 Creating CL context from a GL context

Do not to use the GLUT windowing system in multi-GPU environment because it always creates a GL context on the primary display, and it is not possible to specify which display device to select for a GL context.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

To use Win32 API for windowing in multi-GPU environment:

1. Detect each display by using `EnumDisplayDevices` function. This function lets you obtain the information about display devices in the current session.
2. To query all display devices in the current session, call this function in a loop, starting with `DevNum` set to 0, and incrementing `DevNum` until the function fails. To select all display devices in the desktop, use only the display devices that have the `DISPLAY_DEVICE_ATTACHED_TO_DESKTOP` flag in the `DISPLAY_DEVICE` structure.
3. To get information on the display adapter, call `EnumDisplayDevices` with `lpDevice` set to NULL. For example, `DISPLAY_DEVICE.`**`DeviceString`** contains the adapter name.
4. Use `EnumDisplaySettings` to get DEVMODE. `dmPosition.x` and `dmPosition.y` are used to get the x coordinate and y coordinate of the current display.
5. Try to find the first OpenCL device (winner) associated with the OpenGL rendering context by using the loop technique of 2., above**.**
6. Inside the loop:
   a. Create a window on a specific display by using the `CreateWindow` function. This function returns the window handle (HWND).
   b. Use `GetDC` to get a handle to the device context for the client area of a specific window, or for the entire screen (OR). Use the `CreateDC` function to create a device context (HDC) for the specified device.
   c. Use `ChoosePixelFormat` to match an appropriate pixel format supported by a device context to a given pixel format specification.
   d. Use `SetPixelFormat` to set the pixel format of the specified device context to the format specified.
   e. Use `wglCreateContext` to create a new OpenGL rendering context from device context (HDC).
   f. Use `wglMakeCurrent` to bind the GL context created in the above step as the current rendering context.
   g. Use `clGetGLContextInfoKHR` (See Section 9.7 of the *OpenCL Specification 1.1)* and `CL_CURRENT_DEVICE_FOR_GL_CONTEXT_KHR` parameter to get the number of GL associated devices for CL context creation. If the number of devices is zero go to the next display in the loop. Otherwise, use `clGetGLContextInfoKHR` (See Section 9.7 of the *OpenCL Specification 1.1)* and the `CL_CURRENT_DEVICE_FOR_GL_CONTEXT_KHR` parameter to get the device ID of the CL device associated with OpenGL context.
   h. Use `clCreateContext` (See Section 4.3 of the *OpenCL Specification 1.1)* to create the CL context (of type `cl_context`).

The following code demonstrates how to use WIN32 Windowing API in CL-GL interoperability on multi-GPU environment.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

```
int xCoordinate = 0;
int yCoordinate = 0;

for (deviceNum = 0; EnumDisplayDevices(NULL,
                    deviceNum,
                    &dispDevice,
                    0); deviceNum++)
{

    if (dispDevice.StateFlags &
            DISPLAY_DEVICE_MIRRORING_DRIVER)
    {
        continue;
    }

    DEVMODE deviceMode;

    EnumDisplaySettings(dispDevice.DeviceName,
            ENUM_CURRENT_SETTINGS,
            &deviceMode);

    xCoordinate = deviceMode.dmPosition.x;
    yCoordinate = deviceMode.dmPosition.y;
    WNDCLASS windowclass;

    windowclass.style = CS_OWNDC;
    windowclass.lpfnWndProc = WndProc;
    windowclass.cbClsExtra = 0;
    windowclass.cbWndExtra = 0;
    windowclass.hInstance = NULL;
    windowclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    windowclass.hCursor = LoadCursor(NULL, IDC_ARROW);
    windowclass.hbrBackground = (HBRUSH)GetStockObject(BLACK_BRUSH);
    windowclass.lpszMenuName = NULL;
    windowclass.lpszClassName = reinterpret_cast<LPCSTR>("SimpleGL");
                RegisterClass(&windowclass);
    gHwnd = CreateWindow(
    reinterpret_cast<LPCSTR>("SimpleGL"),
                reinterpret_cast<LPCSTR>(
                "OpenGL Texture Renderer"),
                WS_CAPTION | WS_POPUPWINDOW,
                xCoordinate,
                yCoordinate,
                screenWidth,
                screenHeight,
                NULL,
                NULL,
                windowclass.hInstance,
                NULL);
    hDC = GetDC(gHwnd);

    pfmt = ChoosePixelFormat(hDC, &pfd);

    ret = SetPixelFormat(hDC, pfmt, &pfd);

    hRC = wglCreateContext(hDC);

    ret = wglMakeCurrent(hDC, hRC);

    cl_context_properties properties[] =
    {
            CL_CONTEXT_PLATFORM,
            (cl_context_properties) platform,
            CL_GL_CONTEXT_KHR,
            (cl_context_properties) hRC,
            CL_WGL_HDC_KHR,
            (cl_context_properties) hDC,
            0
    };

    if (!clGetGLContextInfoKHR)
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

```
    {
        clGetGLContextInfoKHR = (clGetGLContextInfoKHR_fn)
        clGetExtensionFunctionAddress(
            "clGetGLContextInfoKHR");
    }

    size_t deviceSize = 0;
    status = clGetGLContextInfoKHR(properties,
                CL_CURRENT_DEVICE_FOR_GL_CONTEXT_KHR,
                0,
                NULL,
                &deviceSize);

    if (deviceSize == 0)
    {
            // no interopable CL device found, cleanup
            wglMakeCurrent(NULL, NULL);
            wglDeleteContext(hRC);
            DeleteDC(hDC);
            hDC = NULL;
            hRC = NULL;
            DestroyWindow(gHwnd);
            // try the next display
            continue;
    }
    ShowWindow(gHwnd, SW_SHOW);
    //Found a winner
    break;
}

cl_context_properties properties[] =
{
    CL_CONTEXT_PLATFORM,
    (cl_context_properties) platform,
    CL_GL_CONTEXT_KHR,
    (cl_context_properties) hRC,
    CL_WGL_HDC_KHR,
    (cl_context_properties) hDC,
    0
};

status = clGetGLContextInfoKHR( properties,
            CL_CURRENT_DEVICE_FOR_GL_CONTEXT_KHR,
            sizeof(cl_device_id),
            &interopDevice,
            NULL);

// Create OpenCL context from device's id
context = clCreateContext(properties,
            1,
            &interopDevice,
            0,
            0,
            &status);
```

G.1.3 Limitations

- It is recommended not to use GLUT in a multi-GPU environment.
- AMD currently supports CL-GL interoperability only in a single-GPU environment.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

G.2 Linux Operating System

G.2.1 Single GPU Environment

G.2.1.1 Creating CL Context from a GL Context

Using GLUT

1. Use `glutInit` to initialize the GLUT library and to negotiate a session with the windowing system. This function also processes the command-line options depending on the windowing system.
2. Use `glXGetCurrentContext` to get the current rendering context (GLXContext).
3. Use `glXGetCurrentDisplay` to get the display (Display *) that is associated with the current OpenGL rendering context of the calling thread.
4. Use `clGetGLContextInfoKHR` (see Section 9.7 of the *OpenCL Specification 1.1*) and the `CL_CURRENT_DEVICE_FOR_GL_CONTEXT_KHR` parameter to get the device ID of the CL device associated with the OpenGL context.
5. Use `clCreateContext` (see Section 4.3 of the *OpenCL Specification 1.1*) to create the CL context (of type `cl_context`).

The following code snippet shows how to create an interoperability context using GLUT in Linux.

```
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
glutInitWindowSize(WINDOW_WIDTH, WINDOW_HEIGHT);
glutCreateWindow("OpenCL SimpleGL");

gGLXContext glCtx = glXGetCurrentContext();

Cl_context_properties cpsGL[] =
{
    CL_CONTEXT_PLATFORM,

    (cl_context_properties)platform,

    CL_GLX_DISPLAY_KHR,

    (intptr_t) glXGetCurrentDisplay(),

    CL_GL_CONTEXT_KHR,

(   intptr_t) glCtx, 0};

status = clGetGLContextInfoKHR(cpsGL,
        CL_CURRENT_DEVICE_FOR_GL_CONTEXT_KHR,
        sizeof(cl_device_id),
        &interopDevice,
        NULL);

// Create OpenCL context from device's id
context = clCreateContext(cpsGL,
            1,
            &interopDevice,
            0,
            0,
            &status);
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Using X Window System

1. Use `XOpenDisplay` to open a connection to the server that controls a display.
2. Use `glXChooseFBConfig` to get a list of GLX frame buffer configurations that match the specified attributes.
3. Use glXChooseVisual to get a visual that matches specified attributes.
4. Use `XCreateColormap` to create a color map of the specified visual type for the screen on which the specified window resides and returns the colormap ID associated with it. Note that the specified window is only used to determine the screen.
5. Use `XCreateWindow` to create an unmapped sub-window for a specified parent window, returns the window ID of the created window, and causes the X server to generate a CreateNotify event. The created window is placed on top in the stacking order with respect to siblings.
6. Use `XMapWindow` to map the window and all of its sub-windows that have had map requests. Mapping a window that has an unmapped ancestor does not display the window, but marks it as eligible for display when the ancestor becomes mapped. Such a window is called unviewable. When all its ancestors are mapped, the window becomes viewable and is visible on the screen if it is not obscured by another window.
7. Use `glXCreateContextAttribsARB` to initialize the context to the initial state defined by the OpenGL specification, and returns a handle to it. This handle can be used to render to any GLX surface.
8. Use `glXMakeCurrent` to make argrument3 (GLXContext) the current GLX rendering context of the calling thread, replacing the previously current context if there was one, and attaches argument3 (GLXcontext) to a GLX drawable, either a window or a GLX pixmap.
9. Use `clGetGLContextInfoKHR` to get the OpenCL-OpenGL interoperability device corresponding to the window created in step 5.
10. Use `clCreateContext` to create the context on the interoperable device obtained in step 9.

The following code snippet shows how to create a CL-GL interoperability context using the X Window system in Linux.

```
Display *displayName = XOpenDisplay(0);

int nelements;
GLXFBConfig *fbc = glXChooseFBConfig(displayName,
DefaultScreen(displayName), 0, &nelements);
      static int attributeList[] = { GLX_RGBA,
                                     GLX_DOUBLEBUFFER,
                                     GLX_RED_SIZE,
                                     1,
                                     GLX_GREEN_SIZE,
                                     1,
                                     GLX_BLUE_SIZE,
                                     1,
                                     None
                                   };
    XVisualInfo *vi = glXChooseVisual(displayName,
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

```
                            DefaultScreen(displayName),
                            attributeList);

    XSetWindowAttributes swa;
    swa.colormap = XCreateColormap(displayName,
                        RootWindow(displayName, vi->screen),
                        vi->visual,
                        AllocNone);
    swa.border_pixel = 0;
    swa.event_mask = StructureNotifyMask;

    Window win = XCreateWindow(displayName,
RootWindow(displayName, vi->screen),
                                  10,
                                  10,
                                  WINDOW_WIDTH,
                                  WINDOW_HEIGHT,
                                  0,
                                  vi->depth,
                                  InputOutput,
                                  vi->visual,
                                  CWBorderPixel|CWColormap|CWEventMask,
                                  &swa);

    XMapWindow (displayName, win);

    std::cout  << "glXCreateContextAttribsARB "
               << (void*) glXGetProcAddress((const
    GLubyte*)"glXCreateContextAttribsARB")
              << std::endl;

    GLXCREATECONTEXTATTRIBSARBPROC glXCreateContextAttribsARB =
    (GLXCREATECONTEXTATTRIBSARBPROC)
                                glXGetProcAddress((const
                                GLubyte*)"glXCreateContextAttribsARB");

    int attribs[] = {
                    GLX_CONTEXT_MAJOR_VERSION_ARB, 3,
                    GLX_CONTEXT_MINOR_VERSION_ARB, 0,
                    0
                    };

    GLXContext ctx = glXCreateContextAttribsARB(displayName,
                            *fbc,
                            0,
                            true,
                            attribs);
    glXMakeCurrent (displayName,

        win,

        ctx);

cl_context_properties cpsGL[] = {
CL_CONTEXT_PLATFORM,(cl_context_properties)platform,
CL_GLX_DISPLAY_KHR, (intptr_t) glXGetCurrentDisplay(),
CL_GL_CONTEXT_KHR, (intptr_t) gGlCtx, 0
                                  };
status = clGetGLContextInfoKHR( cpsGL,
                                    CL_CURRENT_DEVICE_FOR_GL_CONTEXT_KHR,
                                    sizeof(cl_device_id),
                                    &interopDeviceId,
                                    NULL);

// Create OpenCL context from device's id
context = clCreateContext(cpsGL,
           1,
           &interopDeviceId,
           0,
           0,
           &status);
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

G.2.2 Multi-GPU Configuration

G.2.2.1 Creating CL Context from a GL Context

Using X Window System

1. Use `XOpenDisplay` to open a connection to the server that controls a display.
2. Use `ScreenCount` to get the number of available screens.
3. Use `XCloseDisplay` to close the connection to the X server for the display specified in the Display structure and destroy all windows, resource IDs (Window, Font, Pixmap, Colormap, Cursor, and GContext), or other resources that the client created on this display.
4. Use a FOR loop to enumerate the displays. To change the display, change the value of the environment variable DISPLAY.
5. Inside the loop:
   a. Use `putenv` to set the environment variable DISPLAY with respect to the display number.
   b. Use `OpenDisplay` to open a connection to the server that controls a display.
   c. Use `glXChooseFBConfig` to get a list of GLX frame buffer configurations that match the specified attributes.
   d. Use `glXChooseVisual` to get a visual that matches specified attributes.
   e. Use `XCreateColormap` to create a color map of the specified visual type for the screen on which the specified window resides and returns the colormap ID associated with it. Note that the specified window is only used to determine the screen.
   f. Use `XCreateWindow` to create an unmapped sub-window for a specified parent window, returns the window ID of the created window, and causes the X server to generate a CreateNotify event. The created window is placed on top in the stacking order with respect to siblings.
   g. Use `XMapWindow` to map the window and all of its sub-windows that have had map requests. Mapping a window that has an unmapped ancestor does not display the window but marks it as eligible for display when the ancestor becomes mapped. Such a window is called unviewable. When all its ancestors are mapped, the window becomes viewable and is visible on the screen, if it is not obscured by another window.
   h. Use `glXCreateContextAttribsARB` function to initialize the context to the initial state defined by the OpenGL specification and return a handle to it. This handle can be used to render to any GLX surface.
   i. Use `glXMakeCurrent` to make argrument3 (GLXContext) the current GLX rendering context of the calling thread, replacing the previously current context, if there was one, and to attach argument3 (GLXcontext) to a GLX drawable, either a window or a GLX pixmap.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

j. Use clGetGLContextInfoKHR to get the number of OpenCL-OpenGL interoperability devices corresponding to the window created in f, above.

k. If the number of interoperable devices is zero, use glXDestroyContext to destroy the context created at step h, and go to step a; otherwise, exit from the loop (an OpenCL-OpenGL interoperable device has been found).

6. Use clGetGLContextInfoKHR to get the OpenCL-OpenGL interoperable device id.

7. Use clCreateContext to create the context on the interoperable device obtained in the previous step.

The following code segment shows how to create an OpenCL-OpenGL interoperability context on a system with multiple GPUs.

```
displayName = XOpenDisplay(NULL);
int screenNumber = ScreenCount(displayName);
XCloseDisplay(displayName);

for (int i = 0; i < screenNumber; i++)
{
    if (isDeviceIdEnabled())
    {
        if (i < deviceId)
        {
            continue;
        }
    }
    char disp[100];
    sprintf(disp, "DISPLAY=:0.%d", i);
    putenv(disp);
    displayName = XOpenDisplay(0);
    int nelements;
    GLXFBConfig *fbc = glXChooseFBConfig(displayName,
                        DefaultScreen(displayName),
                        0,
                        &nelements);
    static int attributeList[] = { GLX_RGBA,
                                   GLX_DOUBLEBUFFER,
                                   GLX_RED_SIZE,
                                   1,
                                   GLX_GREEN_SIZE,
                                   1,
                                   GLX_BLUE_SIZE,
                                   1,
                                   None
                                 };

    XVisualInfo *vi = glXChooseVisual(displayName,
                        DefaultScreen(displayName),
                        attributeList);
    XSetWindowAttributes swa;
    swa.colormap = XCreateColormap(displayName,
                    RootWindow(displayName, vi->screen),
                    vi->visual,
                    AllocNone);
    swa.border_pixel = 0;
    swa.event_mask = StructureNotifyMask;

    win = XCreateWindow(displayName,
                RootWindow(displayName, vi->screen),
                10,
                10,
                width,
                height,
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

```
                0,
                vi->depth,
                InputOutput,
                vi->visual,
                CWBorderPixel|CWColormap|CWEventMask,
                &swa);

    XMapWindow (displayName, win);

    int attribs[] = {
                        GLX_CONTEXT_MAJOR_VERSION_ARB, 3,
                        GLX_CONTEXT_MINOR_VERSION_ARB, 0,
                        0
                    };

    GLXContext ctx = glXCreateContextAttribsARB(displayName,
                            *fbc,
                            0,
                            true,
                            attribs);
    glXMakeCurrent (displayName,
        win,
        ctx);

    gGlCtx = glXGetCurrentContext();
    properties cpsGL[] = {
    CL_CONTEXT_PLATFORM, (cl_context_properties)platform,
    CL_GLX_DISPLAY_KHR, (intptr_t) glXGetCurrentDisplay(),
    CL_GL_CONTEXT_KHR, (intptr_t) gGlCtx, 0
                          };

    size_t deviceSize = 0;
    status = clGetGLContextInfoKHR(cpsGL,
                        CL_CURRENT_DEVICE_FOR_GL_CONTEXT_KHR,
                        0,
                        NULL,
                        &deviceSize);
    int numDevices = (deviceSize / sizeof(cl_device_id));


    if(numDevices == 0)
    {
        glXDestroyContext(glXGetCurrentDisplay(), gGlCtx);
        continue;
        }
    else
        {
            //Interoperable device found
            std::cout<<"Interoperable device found "<<std::endl;
            break;
         }
}

status = clGetGLContextInfoKHR( cpsGL,
                CL_CURRENT_DEVICE_FOR_GL_CONTEXT_KHR,
                sizeof(cl_device_id),
                &interopDeviceId,
                NULL);

// Create OpenCL context from device's id
context = clCreateContext(cpsGL,
                1,
                &interopDeviceId,
                0,
                0,
                &status);
```


Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

Index

Symbols

.amdil
  generating . . . E-3
  storing AMD IL generated from OpenCL . E-1
.comment
  BIF binary . . . E-1
  storing OpenCL and driver versions that created the binary . . . E-1
.llvmir
  generating . . . E-3
  storing OpenCL immediate representation (LLVM IR) . . . E-1
.rodata
  storing OpenCL runtime control data . . . E-1
.shstrtab
  forming an ELF . . . E-1
.source
  storing OpenCL source program . . . E-1
.strtab
  forming an ELF . . . E-1
.symtab
  forming an ELF . . . E-1
.text
  generating . . . E-3
  storing the executable . . . E-1
_cdecl calling convention
  Linux . . . 2-7
_global atomics . . . 1-23
_local atomics . . . 1-23
_local syntax . . . 5-10, 6-18
_stdcall calling convention
  Windows . . . 2-7

Numerics

1D address . . . 1-13
1D copying
  bandwidth and ratio to peak bandwidth . . 6-4
1D indexing variables
  spawn . . . C-1
  vAbsTidFlat . . . C-1
  vThreadGrpIdFlat . . . C-1
  vTidInGrpFlat . . . C-1
1D indexing version position . . . C-1
2D
  address . . . 1-13
  work-groups
    four number identification . . . 5-7, 6-10
2D addresses
  reading and writing . . . 1-13
3D indexing variables
  spawn . . . C-1
  vAbsTid . . . C-1
  vThreadGrpId . . . C-1
  vTidInGrp . . . C-1
3D indexing versions positions . . . C-1
6900 series GPUs
  optimizing kernels . . . 6-53
79XX series devices . . . 1-2, 1-3

A

absolute work-item id
  indexing
    vaTid register . . . C-1
acceleration
  hardware . . . 5-12
access
  highest bandwidth through GPRs . . . 5-14
  instructions
    ALU . . . 6-46
    LDS . . . 6-46
  memory . . . 1-11, 1-13
    linear arrangement . . . 5-26, 6-48
    tiled arrangement . . . 5-26, 6-48
  patterns
    compute kernels . . . 5-27, 6-49
    controlling . . . 5-10, 6-17
    generating global and LDS memory references . . . 6-30
    inefficient . . . 5-6, 6-8
    pixel shaders . . . 5-27, 6-49, C-1
    preserving sequentially-increasing addressing of the original kernel . . . 6-30
    simple stride and large non-unit strides . . . 5-2, 6-6

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

serializing
  bank conflict 5-2, 6-2
  channel conflict. 5-2, 6-2
 the memory system
  quarter-wavefront units. 5-8, 6-12
 tiled image
  workgroup blocking strategy. . . 5-27, 6-49
access pattern
 efficient vs inefficient 4-36
 typical for each work-item. 5-11
accesses
 that map to same bank 5-9
accumulation operations
 NDRange 1-17
address
 1D 1-13
 2D 1-13
 calculation
  for FETCH instructions. 6-46
  for STORE instructions. 6-46
 normalized. 1-13
 un-normalized 1-13
addressing
 unique in HD 7900 series. 5-3
algorithm
 better performance by changing work-group
  size 5-29
 mapping to LDS 5-15
algorithms
 dynamic scheduling 4-32, 6-36
 simple static partitioning 4-32, 6-35
alignment
 adjusting 6-14
allocating
 buffers
  Open Decode F-2
 images
  OpenCL. 1-20
 memory
  selecting a device. 1-20
 memory buffer
  OpenCL program model. 1-20
ALU
 access instructions
  placed in the same clause 6-46
 clause
  marked as dependent. 6-46
 initiating LDS reads 5-10, 6-17
 instructions 5-17, 6-24, 6-46
 pipeline latency hiding 6-23
ALU/LDS
 instruction 6-46
ALUBusy performance counter. 5-17, 6-24
ALUFetchRatio counter
 reported in the APP Profiler 5-17, 6-24
ALUs
 arrangement of 1-3
 processing elements. 1-3
AMD Accelerated Parallel Processing
 accessing memory
  linear arrangement 5-26, 6-48
  tiled arrangement 5-26, 6-48
 implementation of OpenCL. 1-6
 open platform strategy 1-7
 optimization. 4-1
 performance 4-1, 5-1, 6-1
 programming model 1-7
 relationship of components. 1-6
 software. 4-9
  exposing IL 4-9
  exposing ISA 4-9
 software stack. 1-6
  Installable Client Driver (ICD) B-1
AMD APP KernelAnalyzer 1-7
 determining path used 6-5
 tools used to examine registers . . 5-18, 6-26
 viewing clauses. 6-53
AMD APP Profiler 4-1
 ALUFetchRatio counter. 5-17, 6-24
 CompletePath counter 6-5
 determining path used 6-4
 displaying LDS usage. 5-20, 6-28
 example profiler and bandwidth
  calculation. 4-13
 FastPath counter. 6-5
 Fetch counters 4-12
 Kernel Time metric 4-11
 PathUtilization counter 6-5
 performance counters
  for optimizing local memory . . . 5-10, 6-17
 recording execution time for a kernel. . . 4-11
 reporting dimensions of global
  NDRange 4-12
 reporting static number of register spills
  ScratchReg field 5-18, 6-27
 selecting an optimal value
  latency hiding 5-22, 6-32
 tools used to examine registers . . 5-18, 6-26
 Writer counters 4-12
AMD Core Math Library (ACML) 1-7
AMD GPU
 number of compute units 1-3
 See GPU
AMD media instructions 5-24, 6-42
AMD OpenCL
 See OpenCL

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

AMD Phenom II X4 processor
  performance characteristics 4-29, 6-33
AMD Radeon HD 68XX A-15
AMD Radeon HD 69XX A-15
AMD Radeon HD 75XX A-15
AMD Radeon HD 7770 . . . . 4-29, 4-30
AMD Radeon HD 77XX 5-5, 5-33, A-15
AMD Radeon HD 78XX 5-5, 5-33, A-15
AMD Radeon HD 7970 5-2, 5-5, 5-9, 5-15, 5-24, 5-29
AMD Radeon HD 7970 GPU 4-35
AMD Radeon HD 79XX 1-2, 1-3, 5-3, 5-33
AMD Radeon HD 79XX series. A-15
AMD Radeon HD 7XXX . . . 1-2, 1-3, 1-18, 4-7, 4-22, 5-5, 5-10
AMD supplemental compiler. A-6
  -g option A-6
AMD supplemental compiler option
  -f[n-]bin-source A-6
  -f[no-]bin-amdil A-6
  -f[no-]bin-exe. A-6
  -f[no-]bin-llvmir A-6
AMD tools to examine registers. . . . 5-18, 6-26
AMD Unified Video Decoder block
  decoding H.264 video F-1
AMD vendor-specific extensions A-4
amd_bitalign
  built-in function A-8
amd_bytealign
  built-in function A-8
amd_lerp
  built-in function A-8
AMD_OCL_BUILD_OPTIONS
  environment variables 2-5
AMD_OCL_BUILD_OPTIONS_APPEND
  environment variable 2-5
amd_pack
  built-in function A-7
amd_sad
  buillt-in function A-8
amd_sad4
  built-in function A-8
amd_sadhi
  built-in function A-9
amd_unpack0
  built-in function A-7
amd_unpack1
  built-in function A-7
amd_unpack2
  built-in function A-7
amd_unpack3
  built-in function A-8
AMD-specific optimizations
  performance tips. 5-25, 6-48
analyze
  API call input and output 4-5
API
  C++. 3-2
  call
    analyze input and output 4-5
  hot spots. 4-2
  naming extension functions A-1
  OpenCL 2-1
  platform
    Installable Client Driver (ICD) B-1
    querying B-1
  processing calls 1-14
  Trace View 4-5
API calls
  timing 4-3
  view and debug input parameters and
    output results 4-2
API commands
  three categories 2-6
application code
  developing Visual Studio 3-4
application kernels
  device-specific binaries 2-2
architectural registers
  CPU 6-25
arguments
  cb 4-19
  map_flags 4-19
  offset. 4-19
  operation
    buffer 4-15
    image 4-15
    ptr 4-15
  ptr 4-20
arrangement of ALUs 1-3
asynchronous launch
  scheduling process for GPUs . . . 4-33, 6-37
ATI Radeon HD 5000
  FastPath coalescing 6-12
  FastPath vs CompletePath performance. 6-3
  graphics processors memory paths
    CompletePath. 6-3
    FastPath 6-3
  interleave 6-7
  internal memory 6-11
  scheduling groups of work-items
    wavefronts 5-2, 6-1
ATI Radeon HD 5670
  performance characteristics 4-29, 6-33
  threading. 4-30, 6-34

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

ATI Radeon HD 5870
  bank selection bits 6-7
  channel selection 6-7
  delivering memory bandwidth. 5-9, 6-17
  eight channels. 5-5, 6-8
  eight memory controllers 6-2
  global limit of wavefronts 6-25
  hardware
    performance parameters 5-14, 6-20
  memory
    bandwidth 5-15, 6-22
    channels 5-15, 6-22
  running code 4-35, 6-39
atomic
  operation
    local work size 5-21, 6-29
  unit
    wavefront executing 5-11, 6-19
atomic operations 1-2
atomics
  _global. 1-23
  _local. 1-23

B

bandwidth and ratio to peak bandwidth
  1D copies 6-4
bandwidths
  calculating 4-12
  for different launch dimensions 6-8
  for float1 and float4. 6-12
  including coalesced writes 6-14
  including unaligned access. 6-15
  instruction throughput for GPUs. . 5-23, 6-41
  peak range 4-12
  performance 4-14
bandwith
  very high by embedding address into instruction 5-12
bank address
  LDS 5-9, 6-16
bank conflicts
  controlling bank bits 5-9, 6-16
  generating
    wavefront stalls on the compute unit. 6-17
  LDS examines requests 5-10, 6-17
  measuring
    LDSBankConflict performance
      counter 5-10, 6-17
  serializing the access 5-2, 6-2
  vs channel conflicts 5-2, 6-2
bank selection bits
  ATI Radeon HD 5870 GPU 6-7
barrier
  command-queue 1-2
barrier() instruction 6-18
barriers 5-34
  execution order 1-1
  removing using the compiler 5-11, 6-19
  usage and LDS. 5-16, 6-23
  using in kernel 5-31
  work-group 1-1
  work-items. 6-18
    encountering. 1-1
BIF
  .amdil
    storing AMD IL E-1
  .comment
    storing OpenCL and driver versions that
      created the binary E-1
  .llvmir
    storing immediate representation (LLVM
      IR). E-1
  .source
    storing OpenCL source program E-1
  binary
    .comment section E-1
  bitness. E-3
  changing default behavior E-1
  ELF special sections. E-1
  options to control what is contained in the
    binary E-3
  overview E-1
binary
  application kernels 2-2
  controlling
    BIF options E-3
  CPU. 2-1
  generating
    in OpenCL E-1
    LLVM AS 2-1
  GPU 2-1, 2-2
Binary Image Format (BIF)
  See BIF
bitness
  BIF. E-3
bottlenecks
  discovering 4-1
branch
  granularity
    work-item 1-10
  instructions 1-6
branching
  flow control 1-10
  replacing
    with conditional assignments . . 5-31, 6-53

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

breakpoint
- adding 3-1
- CL kernel function 3-3
- host code 3-3
- no breakpoint is set 3-2
- setting 3-2, 3-4
  - sample GDB debugging session 3-3
- setting a 3-2

buffer
- argument. 4-15
- color 1-11
- command queue. 1-14
- creating temporary runtime 4-21
- global 1-13
  - linear memory layout 1-14
  - source or destination for instruction . 1-13
  - storing writes to random memory locations 1-13
- host side zero copy 4-22
- OpenCL 4-22
- paths 4-21
- pre-pinned. 4-22
- querying the device for the maximum number of constant buffers . . . 5-13, 6-20
- read only
  - L1 and L2 caches 6-21
- regular device 4-22
- relationship
  - sample code. 1-20
- transfer options
  - BufferBandwidth code 4-25
- write combining
  - chip-set dependent. 4-15
- zero copy 4-23
  - available buffer types 4-23
  - calling 4-23
  - size limit per buffer 4-23

BufferBandwidth
- code sample 4-25
- transfer options. 4-25

buffers
- pre-pinned. 4-24
  - optimizing data transfers 4-22

built-in function
- amd_bitalign A-8
- amd_bytealign A-8
- amd_lerp. A-8
- amd_pack A-7
- amd_sad A-8
- amd_sad4 . . . . A-8
- amd_sadhi A-9
- amd_unpack0 . . . . A-7
- amd_unpack1 . . . . A-7
- amd_unpack2 . . . . A-7
- amd_unpack3 . . . . A-8

built-in functions
- for OpenCL language
  - cl_amd_media_ops A-7, A-9
- OpenCL C programs
  - cl_amd_printf A-12
- variadic arguments A-12
- writing output to the stdout stream A-12

burst cycles
- through all channels. 5-5, 6-7

burst write 1-14

C

C front-end
- compiler 2-2

C kernels
- predefined macros A-13

C program sample
- OpenCL 1-20

C programming
- OpenCL 1-18

C++ API 3-2

C++ extension
- unsupported features 7-2

C++ files
- compiling. 2-3

C++ kermel language 7-1

C++ kernels
- building 7-3

C++ language
- leveraging a CPU-targeted routine 4-36, 6-40

C++ templates 7-5

cache
- behavior
  - pixel shader C-1
- CPU vs GPU 4-31, 6-34
- GPU vs CPU 4-31
- L1 . . . . 1-14, 5-14, 6-1, 6-21
- L2 . . . . 1-14, 5-14, 6-1, 6-21
- LDS vs L1. . . . . 5-9, 5-15, 6-22
- memory
  - controlling access pattern 5-10, 6-17
- texture system 1-14

cache coherency protocol
- CPU 4-15

CAL
- completed IL passed to for compilation. . 2-2

calling convention
- Linux. 2-7
- Windows 2-7

caveats
- synchronization. 4-34, 6-37

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

cb argument 4-19
Cedar
  ASIC device 6-39
  different architecture characteristics. . . . 6-32
  optimizing 6-32
  reduced work size
    launching the kernel 6-32
channel
  burst cycles. 5-5, 6-7
  elements in 5-5
  processing serially 5-3, 6-6
  selection 5-5
  selection bits 5-5
channel conflicts
  avoiding
    GPU programming 5-3, 6-6
    work-group staggering 5-7, 6-10
  FastPath 5-8, 6-10
    conflict. 5-8, 6-10
  reading from the same address . . . 5-8, 6-10
  serializing the access 5-2, 6-2
  vs bank conflict 5-2, 6-2
channel selection
  ATI Radeon HD 5870 GPU 6-7
channels
  12 in HD 7900 series 5-3
character extensions A-1
  searching for substrings A-2
character sequence
  format string A-12
Cilk
  dynamic scheduling algorithms . . 4-32, 6-36
  multi-core runtimes 4-32, 6-36
CL context
  associate with GL context G-1
CL kernel function
  breakpoint 3-3
CL options
  change during runtime 2-5
cl_amd_device_attribute_query extension
  querying AMD-specific device attributes A-5
cl_amd_event_callback extension
  registering event callbacks for states. . . A-6
cl_amd_fp64 extension A-4
cl_amd_media_ops extension
  adding built-in functions to OpenCL language
    A-7, A-9
cl_amd_printf extension A-12
cl_ext extensions. A-4
cl_khr_fp64
  supported function A-15
CL_PROFILING_COMMAND_END
  OpenCL timestamp 4-10
CL_PROFILING_COMMAND_QUEUED
  OpenCL timestamp. 4-10
CL_PROFILING_COMMAND_START
  OpenCL timestamp. 4-10
CL_PROFILING_COMMAND_SUBMIT
  OpenCL timestamp. 4-10
CL_QUEUE_PROFILING_ENABLE
  setting the flag 4-10
classes
  passing between host and device 7-3
clause
  ALU
    marked as dependent. 6-46
  AMD GPUs
  architecture for the 6900 series GPUs. . 6-53
  boundaries
    ALU and LDS access instructions . . . 6-46
    broken at control-flow. 6-46
    FETCH, ALU/LDS, and STORE
      instructions 6-46
    ISA dump 6-46
    switching wavefronts 6-46
  conditional assignments 5-31, 6-53
  disassembly example 6-46
  FETCH 6-46
  latency hiding 6-46
  switching
    in the same wavefront 6-46
  viewing
    using APP KernelAnalyzer assembler 6-53
clBuildProgram
  debugging OpenCL program 3-2
clCreateCommandQueue
  sets up a queue
    OpenCL commands are sent. F-3
clCreateKernel
  C++extension 7-2
clDeviceInfo
  querying for device memory. 5-20, 6-29
clEnqueue call
  passing an event to be queried . . 4-33, 6-37
clEnqueue commands. 2-5
clEnqueueNDRangeKernel
  partitioning the workload. 5-20, 6-29
  setting breakpoint in the host code 3-3
clFinish
  blocking operation. 4-34, 6-38
clFinish()
  blocking the CPU 4-12
clFlush
  commands flushed and executed in
    parallel 4-34, 6-38
  flushing to the GPU 4-21

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

clGetPlatformIDs() function
  available OpenCL implementations B-1
clGetPlatformInfo() function
  available OpenCL implementations B-1
  querying supported extensions for OpenCL
    platform A-1
C-like language
  OpenCL 1-18
CLPerfMarker library. 4-2
clustering the stores
  assisting the compiler in disambiguating
    memory addresses. 6-43
  unrolled loop. 6-44
coalesce detection
  ignoring work-item that does not write. . 6-13
coalesced writes
  bandwidths 6-14
  processing quarter-wavefront units 6-12
  reordering your data. 6-12
code
  a simple and accurate algorithm
    performance tips 5-28, 6-50
  avoid writing with dynamic pointer
    assignment performance tips. . 5-28, 6-50
  basic programming steps. 1-20
  BufferBandwidth sample. 4-25
  example with two kernels. 6-12
  FastPath vs CompletePath sample 6-3
  generating IL 4-10
  generating ISA 4-10
  ICD-compliant version B-1, B-3
  parallel min() function. 1-23
  porting unchanged to GPU 5-2
  pre-ICD snippet B-1, B-3
  remove or comment-out
    performance tips 5-28, 6-50
  re-ordering
    improving performance 6-43
  restructuring
    to eliminate nesting 5-32, 6-54
  rewriting to employ array
    transpositions 5-3, 6-6
  running
    on ATI Radeon HD 5870 GPU 4-35, 6-39
    on Linux 2-7
    on Windows 2-6
  runtime steps 1-23
  sample for reading the current value of
    OpenCL timer clock 4-11
code requirements
  Installable Client Driver (ICD) B-1
color buffer 1-11
command processor
  transfer from system to GPU 1-14
command processors
  concurrent processing of command
    queues 1-3
command queue. 1-19
  associated with single device 1-19
  barrier
    enforce ordering within a single queue 1-2
  configured to execute in-order. . . 4-34, 6-38
  creating device-specific 1-20
  elements
    constants 1-14
    kernel execution calls 1-14
    kernels 1-14
    transfers between device and host . . 1-14
  executing kernels 1-19
  execution 1-14
  flushing to the GPU 4-21
  moving data 1-19
  no limit of the number pointing to the
    same device. 1-19
  OpenCL 1-20
  scheduling asynchronously from . 4-33, 6-37
command queue flushing 4-21
command queues 1-3
  multiple 2-6
command-queue barrier 1-2
commands
  API
    three categories 2-6
  buffer. 1-14
  clEnqueue. 2-5
  copy buffers and images 4-21
  driver layer issuing 1-14
  driver layer translating 1-14
  event. 2-6
  GDB 3-3
  kernel 2-6
  memory. 2-6
  non-blocking 4-21
  OpenCL API functions 2-5
  queue 1-15
  read buffers and images 4-21
  synchronizing
    begin executing in OpenCL. . . 4-34, 6-37
  write buffers and images 4-21
communication and data transfers between
  system and GPU
  PCIe 1-14
communication between the host (CPU)
  and the GPU 1-14

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

comparison results
  pixel shader, compute kernel and LDS . C-4
compilation
  GPU-specific binaries 2-2
compile time
  resolving format string A-12
compiler
  converting separate MUL/ADD operations
    to use MAD instruction. 5-24, 6-42
  disambiguating memory addresses
    clustering the stores 6-43
  exposing more parallelism to
    loop unrolling 6-43
  generating spill code. 5-18, 6-27
  LLVM framework. 2-2
  packing instructions into VLIW word
    slots. 6-44
  relying on to remove the barriers . 5-11, 6-19
  set to ignore all extensions A-2
  toolchain 2-1
    back-end 2-1
    OpenCL. 2-1
    sharing front-end 2-1
    sharing high-level transformations 2-1
  transformations 2-1
  using pragma
    unrolling a loop. 5-25, 6-48
  using standard C front-end. 2-2
compiler option
  -f[no-]bin-amdil 2-4
  -f[no-]bin-exe 2-4
  -f[no-]bin-llvmir 2-4
  -f[no-]bin-source 2-4
  -g 2-4
  -O0 2-4
  -save-temps 2-4
compiling
  an OpenCL application 2-2
  C++ files 2-3
  kernels. 2-1
  on Linux
    building 32-bit object files on a 64-bit
      system 2-3
    linking to a 32-bit library. 2-3
    linking to a 64-bit library. 2-3
  OpenCL on Linux 2-3
  OpenCL on Windows 2-2
    Intel C (C++) compiler 2-2
    setting project properties 2-2
    Visual Studio 2008 Professional
      Edition. 2-2
  the host program 2-2
CompletePath
  ATI Radeon HD 5000 graphics processors
    memory paths. 6-3
  counter
    AMD APP Profiler. 6-5
  kernels. 6-4
  MEM_RAT. 6-6
  performance
    ATI Radeon HD 5000 series hardware 6-3
  vs FastPath
    using float1 6-3
complex application
  investigating high-level structure 4-2
computation
  data-parallel model 1-1
compute device structure
  GPU 1-2, 1-4
compute devices
  program
    optimization. 4-1, 5-1, 6-1
    performance 4-1, 5-1, 6-1
compute kernel C-1, C-3
  data-parallel granularity 1-18
  definition 1-17
  differences from pixel shader. C-1
  indexing mode
    linear C-1
  kernel structure C-3
  linear pattern. C-1
  matrix transpose C-2
  performance comparison C-2
  performance results of matrix transpose C-3
  spawn/dispatch pattern. C-1
  strengths
    computationally intensive applications 1-17
  using LDS features for larger performance
    gain C-4
  wavefronts. 1-18
  workgroups 1-18
compute shader units
  video operation F-1
compute unit
  computing number of wavefronts per. . . 5-18
  containing processing elements 5-2
  contents of 5-2
  executing work-groups 5-2, 6-1
  GPU 5-2, 6-1
  LDS usage effects 5-19, 6-28
  mapping 1-7
  processing independent
    wavefronts. 5-16, 6-24
  registers shared among all active
    wavefronts. 6-25

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

scheduling available wavefronts . 5-17, 6-24
stream cores. 1-5
supporting a maximum of eight
work-groups 6-24
supporting up to 32 wavefronts
OpenCL 6-25
work-group availability 5-20, 6-29
compute unites
number in AMD GPU 1-3
compute units
79XX devices 1-2
independent operation 1-9
number in AMD GPUs 1-3
structured in AMD GPUs 1-3
conditional expression
bypassing short-circuiting . . 5-31, 5-32, 6-54
used in loop constructs 5-32, 6-54
constant address
compiler embedding into instruction . . . 5-12
constant buffers
in hardware. 5-13
querying a device when using 5-13
constant memory
domain 1-11
optimization. 5-12, 6-19
performance
same index. 5-12, 6-19
simple direct-addressing
patterns 5-12, 6-19
varying index 5-13, 6-19
constant memory optimization 5-12
constants
caching 1-14
command queue elements. 1-14
enabling
L1 and L2 caches 6-21
inline literal 5-12
constraints
of the current LDS model. 1-18
on in-flight wavefronts 5-17, 6-24
context
creating in OpenCL 4-36, 6-40
creating separate for each device 4-37
creation
allocating buffers F-2
OpenCL F-2
extend vs duplicate 4-36
placing devices in the same
context 4-37, 6-40
relationship
sample code. 1-20
contexts
associating CL and GL. G-1
control flow statement
moving a conditional expression out of
loop constructs. 5-32, 6-54
control-flow boundaries
clauses 6-46
copy map mode
runtime tracks the map location. 4-20
copy memory objects 4-18
transfer policy 4-20
copy performance
steps to improve. 6-16
summary. 6-16
copying data
implicit and explicit 1-12
copying processes 1-13
counters
Fetch. 4-12
Write 4-12
CPU
accessing pinned host memory 4-14
advantages
caches 4-31, 6-35
fast launch time 4-31, 6-35
low latency 4-31, 6-35
back-end
generating packed SSE instructions . 6-44
vectorizing 6-44
binaries 2-1
blocking with clFinish(). 4-12
cache coherency protocol 4-15
caching when accessing pinned host
memory 4-15
code
parallel min() function 1-23
communication between host and GPU 1-14
copying data in pieces and transferring to the
device
using GPU DMA engine 4-14
dedicating a core for scheduling
chores. 4-33, 6-37
each thread is assigned a fixed set of
architectural registers. 6-25
excelling at latency-sensitive tasks 4-30, 6-33
float4 vectorization 4-36, 6-40
kernels 4-35, 6-39
key performance characteristics . 4-29, 6-33
launch time tracking 4-11
leverage a targeted routine
C++ language. 4-36, 6-40
local memory mapping to same cacheable
DRAM used for global memory 4-36, 6-39
low-latency response
dedicated spin loop 4-35, 6-38

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

mapping uncached memory 4-16
memory domain 1-11
more on-chip cache than GPU . . . 4-31, 6-34
multi-core
dynamic scheduling algorithms 4-32, 6-36
no benefit from local memory 5-30
only supports small number of
threads 4-36, 6-40
optimization when programming 4-1, 5-1, 6-1
overlapping copies
double buffering 4-14
predefined macros A-13
processing 2-2
LLVM AS 2-2
OpenCL runtime 2-1
programming using OpenCL 5-29, 6-52
skip copying between host memory
and PCIe memory 1-13
SSE 4-30, 6-34
streaming writes performance 4-15
uncached memory 4-15
vs GPU
notable differences 4-35, 6-39
performance comparison 4-30, 6-34
running work-items 4-35, 6-39
threading 4-30, 6-34
vectorized types vs floating-point
hardware 4-35, 6-39
waiting for the GPU to become idle
by inserting calls 4-12
CPU cache 4-31
vs GPU 4-31
CPU timestamps
for host code 4-2
Creating CL context
from a GL Context G-11
crossbar load distribution 5-3
CUDA
code
workgroup size 5-29, 6-51
greater efficiency using vectorization . . . 6-51
guidance using OpenCL 5-29, 6-51
high single-precision flops
AMD GPU 5-29, 6-51
performance recommendations . . 5-29, 6-51
cygwin
GDB running 3-4
Cypress device 6-39

D

-D name
OpenCL supported options 2-4
data
available to device kernel access 4-18
computations
select a device 1-20
fetch units 1-13
in pinned host memory 4-15
location
scheduling process for GPUs . 4-34, 6-37
memory allocated and initialized 4-21
moving using corresponding command
queue 1-19
native hardware I/O transaction size
four word 6-54
optimizing movement
zero copy memory objects 4-19
parallelism
grouping 1-18
processing
staggered offsets 5-6, 6-9
set
performance tips 5-28, 6-50
structures
minimize bank conflicts 5-3, 6-6
transfer optimization 4-21
transfers
select a device 1-20
to the optimizer 2-2
data transfer
optimizing using pre-pinned buffers 4-22
data transfer execution
confirm efficiency of 4-4
dataflow between host (CPU) and GPU . . 1-12
data-parallel granularity
compute kernels 1-18
data-parallel programming model
executing non-graphic functions 1-7
debug
information
creating metadata structures 2-2
input parameters and output results 4-2
debugger
kernel symbols not visible 3-2
debugging
kernels 2-2, 3-2
OpenCL 3-1
kernels in Windows 3-4
OpenCL programs 3-1

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

session
GDB sample . . . 3-3
hello world kernel sample . . . 3-2
setting the environment . . . 3-2
decode
execution
Open Decode . . . F-3
specifying the output format
NV12 Interleaved . . . F-3
specifying the profile
H.264 . . . F-3
decode session
creating
Open Decode . . . F-3
releasing the resources and closing the session . . . F-4
default memory objects . . . 4-20
tracking . . . 4-20
deferred allocation definition . . . 4-22
definition
kernel . . . 1-17
NDRange . . . 1-17
wavefront . . . 1-10
derived classes . . . 7-3
device
AMD GPU parameters . . . D-1
balanced solution that runs well on CPU and GPU . . . 4-36, 6-40
Cedar ASIC . . . 6-39
creating context . . . 4-36, 6-40
Cypress . . . 6-39
dedicated memory
discrete GPU . . . 4-15
different performance characteristics . . . 4-35, 6-39
extension support listing . . . A-15
fission extension support in OpenCL . . . A-4
fusion . . . 4-15
GPU access is slower . . . 4-15
heterogeneous . . . 4-33, 6-36
kernels
copying between device memory . . . 4-21
list
function call query . . . A-2
memory
avoiding over-allocating . . . 4-16
transfers . . . 4-14, 4-15
multiple
creating a separate queue . . . 4-32, 6-35
when to use . . . 4-31, 6-35
no limit of number of command queues . . . 1-19
obtaining peak throughput . . . 5-24, 6-42
peak performances . . . 4-36, 6-40
placing in the same context . . . 4-37, 6-40
relationship
sample code . . . 1-20
scheduling
across both CPU and GPU . . . 4-32, 6-36
starving the GPU . . . 6-37
device fission extension
reserving a core for scheduling . . . 4-33, 6-37
device timestamps
retrieval . . . 4-2
device-optimal access pattern
threading . . . 1-23
devices
79XX series . . . 1-2, 1-3
device-specific
operations
kernel execution . . . 1-19
program compilation . . . 1-19
Direct Memory Access (DMA)
engine . . . 1-15
signaling transfer is completed . . . 1-15
transfers data to device memory . . . 4-14
transfers . . . 1-15
parallelization . . . 1-15
directives
extension name overrides . . . A-2
order . . . A-2
disassembly information
getting details after running the profiler . 4-10
discrete GPU
moving data . . . 4-34, 6-37
do loops
vs for loops . . . 5-32
domains
of synchronization . . . 1-1
command-queue . . . 1-1
work-items . . . 1-1
double buffering
overlapping CPU copies with DMA . . . 4-14
double copying
memory bandwidth . . . 1-13
double-precision
supported on all Southern Island devices . . . 5-23
double-precision floating-point
performing operations . . . 1-6
double-precision support . . . 5-33
drill down
kernel execution . . . 4-5
driver layer
issuing commands . . . 1-14
translating commands . . . 1-14

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

dynamic frequency scaling
  device performance 4-32
dynamic scheduling
  algorithms
    Cilk 4-32, 6-36
  heterogeneous workloads. 4-33, 6-36
  separate instructions. 1-2

E

efficiency
  of kernel and data transfer. 4-4
element
  work-item. 1-19
ELF
  .rodata
    storing OpenCL runtime control data. E-1
  .shstrtab
    forming an ELF. E-1
  .strtab
    forming an ELF. E-1
  .symtab
    forming an ELF. E-1
  .text
    storing the executable E-1
  format E-1
  forming E-1
  header fields E-2
  special sections
    BIF E-1
enforce ordering
  between or within queues
    events 1-2
  synchronizing a given event. 1-19
  within a single queue
    command-queue barrier 1-2
engine
  DMA 1-14
enqueuing
  commands in OpenCL 1-2
  multiple tasks
    parallelism. 1-1
  native kernels
    parallelism. 1-1
Enqueuing commands before flushing 4-34
environment variable
  AMD_OCL_BUILD_OPTIONS 2-5
  AMD_OCL_BUILD_OPTIONS_APPEND . 2-5
  setting to avoid source changes 3-2
Euclidean space
  output domain C-1
event
  commands. 2-6
  enforces ordering
    between queues 1-2
    within queues 1-2
  synchronizing 1-19
event commands. 2-6
Event Wait Lists
  placing Event_Objects F-3
Event_Object
  decoding execution
    OpenCL F-3
events
  forced ordering between. 1-2
Evergreen
  optimizing kernels 6-53
exceptions
  C++ 7-6
executing
  branch 1-10
  command-queues in-order 4-34, 6-38
  kernels. 1-1, 1-5, 1-19
    using corresponding command queue 1-19
  kernels for specific devices
    OpenCL programming model. 1-18
  loop 1-10
  non-graphic function
    data-parallel programming model. 1-7
  work-items
    on a single processing element . . 5-2, 6-1
execution
  command queue 1-14
  of a single instruction over all
    work-items. 1-18
  of GPU non-blocking kernel 4-12
  OpenCL applications. 2-6
  order
    barriers 1-1
  range
    balancing the workload. 5-16, 6-23
    optimization. 5-16, 6-23
  single stream core 1-15
execution dimensions
  guidelines 4-18
execution of work-items 1-16
explicit copying of data 1-12
extension
  cl_amd_popcnt A-7
  clCreateKernel 7-2
extension function pointers A-3
extension functions
  NULL and non-Null return values. A-3
extension support by device
  for devices 1 . . . . A-15
  for devices 2 and CPUs A-16

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

extensions
  all A-2
  AMD vendor-specific A-4
  approved by Khronos Group A-1
  approved list from Khronos Group. A-3
  character strings. A-1
  cl_amd_device_attribute_query A-5
  cl_amd_event_callback
    registering event callbacks for states . A-6
  cl_amd_fp64 . . . . A-4
  cl_amd_media_ops. A-7, A-9
  cl_amd_printf A-12
  cl_ext A-4
  compiler set to ignore. A-2
  device fission A-4
  disabling A-2
  enabling A-2, A-3
  FunctionName string A-3
  kernel code compilation
    adding defined macro A-3
  naming conventions A-1
  optional A-1
  provided by a specific vendor A-1
  provided collectively by multiple vendors A-1
  querying for a platform. A-1
  querying in OpenCL A-1
  same name overrides. A-2
  use in kernel programs A-2
external pins
  global memory bandwidth 5-15, 6-22

F

-f[n-]bin-source
  AMD supplemental compiler option A-6
-f[no-]bin-amdil
  AMD supplemental compiler option A-6
  compiler option 2-4
-f[no-]bin-exe
  AMD supplemental compiler option A-6
  compiler option 2-4
-f[no-]bin-llvmir
  AMD supplemental compiler option A-6
  compiler option 2-4
-f[no-]bin-source
  compiler option 2-4
false dependency 5-34
FastPath
  ATI Radeon HD 5000 graphics processors
    memory paths 6-3
  channel conflicts. 5-8, 6-10
  coalescing
    ATI Radeon HD 5000 devices. 6-12
counter
    AMD APP Profiler 6-5
  kernels 6-4
  MEM_RAT_CACHELESS 6-6
  OpenCL read-only images 6-4
  operations are used
    MEM_RAT_CACHELESS instruction . 6-5
  performance
    ATI Radeon HD 5000 series hardware 6-3
  reading from same address is a
    conflict 5-8, 6-10
  vs CompletePath
    using float1. 6-3
FETCH
  clause 6-46
  instruction 6-46
    address calculation 6-46
fetch unit
  loads 1-13
  processing 1-13
  stores 1-13
  streaming stores. 1-13
  transferring the work-item 1-13
fetches
  memory
    stalls 1-16
FetchInsts counters
  AMD APP Profiler. 4-12
five-way VLIW processor 6-1
float1
  bandwidths 6-12
  FastPath vs CompletePath 6-3
  unaligned access 6-15
float4
  bandwidths 6-12
  data types
    code example. 6-12
  eliminating conflicts 6-11
  format
    transferring data. 6-11
  using 5-30, 6-45, 6-52
  vectorization 4-36, 6-40, 6-45
  vectorizing the loop 6-44
float4 vs float1 formats
  performances 6-11
floating point operations
  double-precision 1-6
  single-precision. 1-6
flow control 1-10
  branching 1-10
  execution of a single instruction over
    all work-items 1-18

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

flushing
  command queue 4-21
FMA
  fused multipe add 5-33
FMA4 instructions 5-31
for loops
  vs do or while loops 5-32
forced ordering of events 1-2
format string A-12
  conversion guidelines A-12
  resolving compile time A-12
front-end
  ensuring the kernels meet OpenCL
    specification 2-2
  performs
    semantic checks 2-2
    syntactic checks 2-2
  standard C 2-2
  translating 2-2
front-end supports
  additional data-types
    float8 2-2
    int4 2-2
  additional keywords
    global 2-2
    kernel 2-2
  built-in functions
    barrier() 2-2
    get_global_id() 2-2
function call
  querying A-2
function names
  undecorated in Windows 2-7
FunctionName string
  address of extension A-3
fusion devices 4-15

G

-g
  compiler option 2-4
  experimental feature A-6
-g option
  passing to the compiler 3-2
gather/scatter model
  Local Data Store (LDS) 1-18
Gaussian Blur operation
  example F-1
gcc
  not supported 3-4
GCC option syntax E-3
GDB
  sample session 3-3
GDB (GNU project debugger). A-6
  command 3-3
  documentation 3-4
  running cygwin 3-4
  running minGW 3-4
gDEBugger 3-1
  downloading and installing 3-1
  limitations 3-1
generate binary images offline
  cl_amd_offline_devices A-6
get group ID
  changing launch order 5-7, 6-10
get group ID values
  are in ascending launch order 5-7, 6-10
GFLOPS
  kernel 1-13
  system 1-13
GL context
  associate with CL context. G-1
global buffer 1-13, 1-14
  characterstics 1-13
  linear memory layout 1-14
  source or destination for instruction 1-13
  storing writes to random memory
    locations 1-13
global ID values
  work-group order. 5-7, 6-10
global level for partitioning work. . . . 5-20, 6-28
global memory (VRAM). 1-13
global memory bandwidth
  external pins 5-15, 6-22
global memory domain 1-11
global resource constraints
  in-flight wavefronts 5-17, 6-24
global scope 1-2
global synchronization. 1-2
global work-size 5-20, 6-28
  defined 1-19
  dividing into work-groups 1-19
globally scoped constant arrays
  improving performance of OpenCL
    stack 5-13, 6-20
GlobalWorkSize field
  reporting dimensions of the NDRange. . 4-12
GLUT windowing system G-2
GNU project debugger
  GDB, description. 3-2
GNU project debugger (GDB)
  See GDB
GPR
  LDS usage 5-16, 6-23
  mapping private memory
    allocations to. 5-14, 6-21
  re-write the algorithm 5-18, 6-27

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

GPRs
  provide highest bandwidth access 5-14
GPU 1-6
  6900 series
    clause-based 6-53
    optimizing kernels 6-53
  accessing pinned host memory
    through PCIe bus 4-15
  adjusting alignment 6-14
  advantages
    high computation throughput . . 4-31, 6-35
    latency hiding 4-31, 6-35
  ATI Radeon HD 5670 threading . 4-30, 6-34
  binaries 2-1
  clause boundaries
  command queue flushing 4-21
  communication between host and GPU 1-14
  compiler
    packing instructions into VLIW word
      slots 6-44
  compute performance tips 5-25, 6-48
  constraints on in-flight wavefronts 5-17, 6-24
  copying data from host to GPU 1-12
  dataflow between host and GPU 1-12
  decompression
    Open Decode F-1
  determining local memory size . . . 5-9, 6-16
  device parameters D-1
  discrete
    existing in a separate address
      space 4-34, 6-37
  discrete device memory
    dedicated 4-15
    directly accessible by CPU 4-16
  divergent control-flow 4-31, 6-34
  excelling at high-throughput 4-29, 6-33
  execute the workload 5-30, 6-52
  exploiting performance
    specifying NDRange 5-16, 6-23
  float4 vectorization 4-36, 6-40
  flushing
    using clFlush 4-21
  fundamental unit of work
    is called wavefront 5-21, 6-29
  gather/scatter operation 4-30, 6-34
  global limit on the number of active
    wavefronts 6-25
  global memory system optimization. 5-1, 6-1
  high single-precision flops
    CUDA programmers guidance. 5-29, 6-51
  improving performance
    using float4 6-45
  Instruction Set Architecture (ISA). 4-10
  kernels 4-35, 6-39
  key performance characteristics . 4-29, 6-33
  launch time tracking 4-11
  loading constants into hardware cache . 6-19
  multiple compute units 5-2, 6-1
  new aspects to scheduling
    process 4-32, 6-36
  non-blocking kernel execution 4-12
  optimization when programming 4-1
  parallel min() function code 1-23
  parallelizing large numbers of
    work-items 1-15
  parameters
    56xx, 57xx, 58xx, Eyfinity6, and 59xx
      devices D-7
    64xx devices D-5, D-6
    65xx, 66xx, and 67xx devices D-4
    68xx and 69xx devices D-2, D-3
    Exxx, Cxx, 54xx, and 55xx devices. . . D-8
  performance
    LDS optimizations 5-30, 6-52
    when programming 4-1, 5-1, 6-1
  power efficiency 4-29, 6-33
  predefined macros A-13
  processing 2-2
    commands 1-14
    LLVM IR-to_CAL IL module. 2-2
    OpenCL runtime. 2-1
  programming. 1-7
    adjacent work-items read or write
      adjacent memory addresses . . 5-3, 6-6
    avoiding channel conflicts 5-3, 6-6
  programming strategy
    raw compute horsepower 5-30, 6-52
  re-computing values
    per-thread register resources . 5-30, 6-52
  registers 6-25
  reprocessing the wavefront 5-9, 6-17
  scheduling. 1-15
    asynchronous launch 4-33, 6-37
    data location. 4-34, 6-37
    even and odd wavefronts 6-46
    heterogeneous compute
      devices. 4-32, 6-36
    the work-items 6-1
  specific macros. 2-1
  starving the devices 6-37
  storing writes to random memory
    locations 1-13
  structure 1-2, 1-4
  thread single-cycle switching 4-30, 6-34
  threading. 5-16, 6-23
  throughput of instructions for 5-23, 6-41

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

transferring host memory to device
memory 4-14
pinning 4-14
transparent scheduled work 4-33, 6-37
using multiple devices 4-31, 6-35
video operation
compute shader units F-1
vs CPU
floating-point hardware vs vectorized types 4-35, 6-39
notable differences 4-35, 6-39
performance comparison 4-30, 6-34
running work-items 4-35, 6-39
wavefronts to hide latency 5-17, 6-24
write coalescing 6-13
Write Combine (WC) cache 5-1, 6-1
GPU cache
vs CPY 4-31
GPU memory system 5-1
GPUs
dedicating for specific operations 1-15
masking 1-15
granularity
branch 1-10
data-parallel 1-18
per-work-group allocation 5-19, 6-27
wavefront 1-10
graph
of limiting resource 4-6
guidance
for CPU programmers 5-29, 6-52
for CUDA programmers 5-29, 6-51
general tips 5-27, 6-49
guidelines for partitioning
global level 5-20, 6-28
local level 5-20, 6-28
work/kernel level 5-20, 6-28

H

H.264
specifying the decode profile F-3
H.264 video
decoding F-1
decompressing the video F-1
reading compressed elementary stream frames and supporting parameters F-1
hardware
overview 1-1
hardware acceleration 5-12
hardware constant buffers
taking advantage of 5-13
hardware limit
active work-items 1-15
hardware performance parameters
OpenCL memory resources 5-14, 6-20
HD 5000 series GPU
work-group dispatching 6-7
header fields
in ELF E-2
hello world sample kernel 3-2
heterogeneous devices
scheduler
balancing grain size 4-33, 6-36
conservative work allocation 4-33, 6-36
sending different workload sizes to different devices 4-33, 6-36
using only the fast device 4-33, 6-36
scheduling
process for GPUs 4-32, 6-36
situations 4-33, 6-36
hiding latency 4-34
how many wavefronts 5-17
hierarchical subdivision
OpenCL data-parallel programming model 1-1
hierarchical-Z pattern
pixel shader C-1
high-level structure
mode for investigating 4-2
host
application mapping 4-18
communication between host and GPU 1-14
copying data from host to GPU 1-12
dataflow between host and GPU 1-12
memory
device-visible 4-15
domain 1-11
Memcpy transfers 4-21
pinned staging buffers 4-14
pinning and unpinning 4-14
transferring to device memory 4-14
memory transfer methods 4-14
host to device 4-14
pinning and unpinning 4-14
runtime pinned host memory staging buffers 4-14
program
OpenCL 2-2
program compiling 2-2
host code
breakpoint 3-3
CPU and device timestamps 4-2
platform vendor string B-3
setting breakpoint 3-3
clEnqueueNDRangeKernel 3-3

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

host memory
  cost of pinning/unpinning 4-14
  faster than PCIe bus 4-16
  transfer costs 4-20
host side zero copy buffers 4-22
host/device architecture single platform
  consisting of a GPU and CPU. 1-19

I

-I dir
  OpenCL supported options 2-4
I/O transaction size
  four word 6-54
ID values
  global
    work-groups order 5-7, 6-10
idle stream cores 1-11
if blocks
  restructuring the code to eliminate
    nesting 5-32, 6-54
IL
  compiler
    using and IL shader or kernel 4-9
  complete 2-1
  description 4-9
  exposing 4-9
  incomplete 2-1
  passing the completed IL to the CAL
    compiler 2-2
  shader or kernel
    translating into hardware instructions or
      software emulation layer 4-9
image
  argument. 4-15
  device kernels
    converting to and from linear address
      mode 4-21
  paths. 4-21
  reads. 1-13
images
  cost of transferring 4-21
implicit copying of data. 1-12
index space
  n-dimensional 1-7
indexing
  registers vs LDS. 5-10, 6-17
indexing mode
  output domain. C-1
  vWinCoord register. C-1
indexing variables
  1D spawn C-1
  3D spawn C-1
  vTgroupid C-1
  vTid C-1
inheritance
  strict and multiple 7-1
inline literal constants 5-12
in-order queue property
  leveraging 4-34, 6-38
input parameters
  view and debug 4-2
input stream
  NDRange 1-17
Installable Client Driver (ICD). B-1
  AMD Accelerated Parallel Processing
    software stack B-1
  compliant version of code B-1, B-3
  overview B-1
  pre-ICD code snippet B-1, B-3
  using B-1
  using the platform API B-1
instruction
  ALU. 5-17, 6-24, 6-46
  ALU/LDS. 6-46
  AMD media. 5-24, 6-42
  bandwidth
    throughput for GPUs 5-23, 6-41
  barrier()
    kernel must include 6-18
  branch. 1-6
  FETCH 6-46
  global buffer 1-13
  kernel 1-14
  LDS. 6-46
  MAD 5-24, 6-42
  MEM_RAT_CACHELESS 6-5
  MEM_RAT_STORE 6-6
  sequence
    MEM_RAT 6-5
    stream cores 1-6
    TEX 6-5
    VFETCH. 6-5
    WAIT_ACK. 6-5
  STORE 6-46
  vfetch 6-5
  VLIW. 6-43
Instruction Set Architecture (ISA)
  defining 4-10
  dump
    examine LDS usage 5-20, 6-28
    showing the clause boundaries 6-46
    tools used to examine registers 5-18, 6-26
  exposing 4-9
instructions
  dynamic scheduling 1-2
  scalar and vector 1-3

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

integer
  performing operations 1-6
Intel C (C++) compiler
  compiling OpenCL on Windows 2-2
interleave
  ATI Radeon HD 5000 GPU 6-7
Intermediate Language (IL)
  See IL
internal memory
  ATI Radeon HD 5000 series devices. . . 6-11
interoperability context
  code for creating G-3
interrelationship of memory domains 1-12
ISA
  SI 5-33
ISA code
  viewing 4-8

J

jwrite combine
  CPU feature 4-22

K

kernel
  accessing
    local memory 5-11, 6-19
    making data available 4-18
  analyzing stream processor 4-9
  attribute syntax
    avoiding spill code and improving performance 5-18, 6-27
  avoid declaring global arrays 5-27, 6-49
  bandwidth and ratio 6-8
  barrier() instruction 6-18
  changing width, data type and work-group
    dimensions 5-6, 6-8
  clauses 6-46
  code
    parallel min() function 1-24
  code compilation
    adding a defined macro with the name of
      the extension A-3
  code example A-13
  code sample
    FastPath vs CompletePath 6-3
  command queue elements 1-14
  commands 2-6
  compiling 2-1
  compute C-1
    definition 1-17
    differences from pixel shader C-1
    linear index mode C-1
    matrix transpose C-2
    spawn/dispatch pattern C-1
    strengths 1-17
  converting to and from linear address
    mode images 4-21
  copying between device memory 4-21
  CPU 4-35, 6-39
  creating within programs 1-20
  debugging 2-2
  definition of 1-17
  device-specific binaries 2-2
  differences between CPU and
    GPU 4-35, 6-39
  distributing in OpenCL 2-1
  divergent branches
    packing order 5-22, 6-31
  enqueueing 4-33, 6-37
  estimating memory bandwidth 4-12
  example that collaboratively writes, then
    reads from local memory 5-11, 6-18
  executed as a function of multi-dimensional
    domains of indices 1-19
  executing 1-1
    runtime 4-21
    using corresponding command queue 1-19
  execution
    device-specific operations 1-19
    modifying the memory object 4-20
  execution calls
    command queue elements 1-14
  execution time
    hiding memory latency 5-17, 6-24
    latency hiding 5-22, 6-32
    sample code 4-10
  FastPath and CompletePath 6-4
  flushing 4-33, 6-37
  GFLOPS 1-13
  GPU 4-35, 6-39
    non-blocking execution 4-12
  hello world sample 3-2
  increasing the processing 6-30
  instructions over PCIe bus 1-14
  interactive tuning of OpenCL 4-8
  keyword 1-17
  launch time
    CPU devices 4-11
    GPU devices 4-11
    tracking 4-11
  level 5-20, 6-28
  loading 3-2
  moving work to 6-29, 6-30
  no breakpoint set 3-2

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

optimizing
for 6900 series GPUs 6-53
for Evergreen 6-53
overloading 7-2
passing data to
memory objects 4-13
performance
float4. 6-11
preserving sequentially-increasing
addressing of the original kernel 6-30
program
OpenCL 2-2
programming
enabling extensions A-3
programs using extensions A-2
required memory bandwidth. 4-12
running on compute unit 1-7
samples of coalescing patterns 6-13
settings C-1
staggered offsets 5-6, 6-9
stream. 1-13
structure
of naive compute kernel C-3
of naive pixel shader C-2
submitting for execution 1-20
synchronization points 1-18
unaligned access
float1. 6-14
unrolled
using float4 vectorization 6-45
use of available local memory 6-31
using constant buffers 5-13, 6-20
work-item 1-7
kernel and function overloading 7-1
kernel attribute
rreqd_work_group_size 4-7
kernel commands 2-6
kernel execution
accessing 3-1
confirm efficiency of 4-4
drill down 4-5
identifying where application is bound. . . 4-2
kernel optimization
analyzing ISA code 4-8
Kernel Time metric
AMD APP Profiler. 4-11
record execution time automatically. . . . 4-11
kernel_name
construction 3-2
KernelAnalyzer
AMD APP 4-8
kernels
debugging 3-2
timing the execution of. 4-10
kernels and shaders 1-7
Khronos
approved list of extensions A-3

L

L1
convolution 5-15, 6-22
matrix multiplication 5-15, 6-22
read path 5-15, 6-22
L1 cache 1-14, 5-14, 5-33, 6-1, 6-21
L1 vs LDS. 5-15, 6-22
native data type 5-15, 6-22
vs LDS 5-9
L2 cache 1-3, 1-14, 5-14, 6-1, 6-21
memory channels on the GPU . . 5-15, 6-22
latency
hiding 4-34
hiding in memory 1-15
latency hiding 1-9, 1-16, 5-16, 6-23
ALU pipeline 6-23
clause 6-46
execution time for each kernel. . . 5-22, 6-32
number of wavefronts/compute
unit 5-22, 6-32
scheduling wavefronts 6-23
launch dimension
performance 6-8
launch fails
preventing. 6-27
launch order
for get group ID 5-7, 6-10
get group ID
changing. 5-7, 6-10
launch overhead
reducing in Profiler 4-11
launch time
GPU vs CPU 4-31, 6-34
launching
threads 1-23
launching the kernel
determining local work size 5-21, 6-29
reduced work size
Cedar 6-32
LDS
allocation on a per-work-group
granularity. 5-19, 6-27
bank conflicts 5-33
pattern results 6-30
cache
accelerating local memory
accesses 5-9, 6-16

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

LDS vs L1 . . . . 5-15, 6-22
native format 5-15, 6-22
converting a scattered access pattern
to a coalesced pattern 5-15, 6-22
description 1-18
examining requests for bank
conflicts 5-10, 6-17
examining usage
generating ISA dump 5-20, 6-28
filling from global memory 5-15, 6-22
gather/scatter model 1-18
impact of usage on wavefronts/compute
unit 5-19
initiating with ALU operation 5-10, 6-17
instruction 6-46
limiting number of wavefronts 4-8
linking to GPR usage and wavefront-per-
SIMD count 5-16, 6-23
local memory size 5-19, 6-27
bank address 5-9, 6-16
mapping an algorithm 5-15
maximum allocation for work-group 5-33
optimizations and GPU
performance 5-30, 6-52
optimized matrix transpose C-2
outperforms
compute kernel C-4
pixel shader C-4
performance gain
compute kernel C-4
performance results of matrix transpose C-4
pixel shader C-1
read broadcast feature 5-15, 6-22
reading from global memory 5-15, 6-22
sharing
across work-groups 5-16, 6-23
between work-items 6-23
size 5-15, 6-22
size allocated to work-group 1-18
tools to examine the kernel 5-20, 6-28
usage effect
on compute-unit 5-19, 6-28
on wavefronts 5-19, 6-28
using barriers 5-16, 6-23
using local memory 1-13
vs L1 cache 5-9
vs registers
indexing flexibility 5-10, 6-17
LDS access instructions
placed in the same clause 6-46
LDS matrix transpose C-4
LDS model constraints 1-18
data sharing 1-18
memory accesses outside the
work-group 1-18
size is allocated per work-group 1-18
LDSBankConflict
optimizing local memory usage . . 5-10, 6-18
performance counter 5-10, 6-17
library
math 5-24, 6-42
SDKUtil 2-3
limiting resource
graph of 4-6
linear indexing mode
compute kernel C-1
linear layout format 5-26, 6-48
linear memory layout 1-14
memory stores 1-14
linear pattern
compute kernel C-1
linking
creating an executable 2-3
in the built-in OpenCL functions 2-2
object files 2-3
OpenCL on Linux 2-3
options
SDKUtil library 2-3
to a 32-bit library compiling on Linux 2-3
to a 64-bit library compiling on Linux 2-3
Linux
building 32-bit object files on a 64-bit
system 2-3
calling convention
_cdecl 2-7
compiling OpenCL 2-3
linking 2-3
linking
to a 32-bit library 2-3
to a 64-bit library 2-3
linking options
SDKUtil library 2-3
running code 2-7
SDKUtil library 2-3
list of supported extensions
approved by the Khronos Group A-3
literal constant 5-12
LLVM
compiler 2-2
framework
compiler 2-2
linker 2-2
LLVM AS
CPU processing 2-2
generating binaries 2-1

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

LLVM IR
BIF E-1
compatibility E-1
enabling recompilation to the target E-1
front-end translation 2-2
generating a new code E-1
LLVM IR-to-CAL IL module
GPU processing 2-2
load distribution
crossbar 5-3
local cache memory
key to effectively using. 5-10, 6-17
Local Data Store (LDS)
See LDS
local level for partitioning work. 5-20, 6-29
local memory
determining size 5-9, 6-16
domain 1-11
LDS
optimization 5-9, 6-16
size. 5-19, 6-27
no benefit for CPU 5-30
scratchpad memory 5-10, 6-18
writing data into 5-11, 6-18
local ranges
dividing from global NDRange . . . 5-16, 6-23
local work size 5-20, 6-29
location indexing
pixel shader C-1
loop
constructs
conditional expressions 5-32, 6-54
execute 1-10
types
experimenting 5-32, 6-54
unrolling 5-25, 6-43
4x 6-43
exposing more parallelism 6-43
increasing performance 5-32, 6-54
performance tips 5-28, 6-50
using pragma compiler directive
hint 5-25, 6-48
with clustered stores 6-44
vectorizing
using float4. 6-44
loop unrolling optimizations 5-28
loops
for vs do or while 5-32
Low-Level Virtual Machine (LLVM)
See LLVM

M

macros
GPU-specific. 2-1
predefined
CPU A-13
GPU A-13
OpenCL C kernels A-13
MAD
double-precision operations 6-41
instruction 5-24, 6-42
converting separate MUL/ADD
operations 5-24
single precision operation 6-41
MAD instruction
converting separate MUL/ADD
operations. 6-42
map calls. 4-20
tracking default memory objects 4-20
map_flags argument 4-19
mapping
executions onto compute units 1-7
memory into CPU address space
as uncached. 4-16
OpenCL 1-8
runtime transfers
copy memory objects. 4-19
the host application 4-18
user data into a single UAV. 6-4
work-items onto n-dimensional
grid (ND-Range). 1-8
work-items to stream cores 1-7
zero copy memory objects. 4-19
mapping/unmapping transfer
pin/unpin runtime 4-20
maps
non-blocking 4-35, 6-38
masking GPUs 1-15
math libraries 5-24, 6-42
function (non-native). 5-24, 6-42
native_function 5-24, 6-42
matrix multiplication
convolution
L1 . . . . 5-15, 6-22
matrix transpose
naive compute kernel. C-2
naive pixel shader C-2
performance comparison
compute kernel vs pixel shader. C-2
performance results
of compute kernel C-3
of LDS C-4
of pixel shader C-2

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

media instructions
AMD . . . 5-24, 6-42
mem_fence operation . . . 1-2
MEM_RAT
instruction sequence meaning . . . 6-5
means CompletePath . . . 6-6
MEM_RAT_CACHELESS
instruction . . . 6-5
means FastPath . . . 6-6
MEM_RAT_STORE instruction . . . 6-6
Memcpy
transferring between various kinds of host memory . . . 4-21
memories
interrelationship of . . . 1-11
memory
access . . . 1-11, 1-13
access patterns . . . 5-27, 6-49
bank conflicts on the LDS . . . 6-30
combining work-items in the NDRange index space . . . 6-30
compute kernels . . . 5-27, 6-49
holes . . . 6-30
pixel shaders . . . 5-27, 6-49
preserving . . . 6-30
accessing local memory . . . 5-11, 6-19
allocation
in pinned host memory . . . 4-21
select a device . . . 1-20
architecture . . . 1-11
bandwidth . . . 1-13
ATI Radeon HD 5870 GPU . . . 5-15, 6-22
calculating . . . 4-12
double copying . . . 1-13
estimation required by a kernel . . . 4-12
performance . . . 4-14
channels
ATI Radeon HD 5870 GPU . . . 5-15, 6-22
L2 cache . . . 5-15, 6-22
commands . . . 2-6
controllers
ATI Radeon HD 5870 GPU . . . 6-2
delivering bandwidth
ATI Radeon HD 5870 GPU . . . 5-9, 6-17
domains
constant . . . 1-11
global . . . 1-11
host . . . 1-11
interrelationship . . . 1-12
local . . . 1-11
PCIe . . . 1-11
private . . . 1-11
fence
barriers . . . 1-18
operations . . . 1-18
global
OpenCL . . . 5-8, 6-11
global (VRAM) . . . 1-13
hiding latency . . . 1-15, 1-16
highly efficient accessing . . . 5-11
host
cost of pinning/unpinning . . . 4-14
initializing with the passed data . . . 4-21
latency hiding reduction . . . 5-16, 6-23
limitation
partitioning into multiple clEnqueueNDRangeKernel commands . . . 5-20, 6-28, 6-29
linear layout . . . 1-14
loads . . . 1-13
local
increasing the processing . . . 6-30
moving processing tasks into the kernel . . . 6-31
scratchpad memory . . . 5-10, 6-18
mapping
CPU . . . 4-36, 6-39
uncached . . . 4-16
object allocation
OpenCL context . . . 1-19
object properties
OpenCL . . . 4-17
obtaining through querying
clDeviceInfo . . . 5-20, 6-29
OpenCL domains . . . 1-11
optimization of constant . . . 5-12
paths
ATI Radeon HD 5000 graphics processors . . . 6-3
pinned . . . 4-20
read operations . . . 1-13
request . . . 1-13
wavefront is made idle . . . 5-16, 6-23
source and destination
runtime transfers . . . 4-21
stores . . . 1-13
streaming . . . 1-13
system in GPU . . . 5-1
system pinned . . . 1-13
tiled layout . . . 5-26
tiling physical memory layouts . . . 5-26, 6-48
transfer management . . . 1-13
types used by the runtime . . . 4-13
uncached . . . 4-15
Unordered Access View (UAV) . . . 5-8, 6-11

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

write operations 1-13
memory access
stream cores. 1-13
memory alignment
page boundary 1-13
memory bandwidth
required by kernel 4-12
memory channel
contents of 5-3
memory channel mapping 5-4
memory commands 2-6
memory object
first use slower than subsequent. 4-16
memory object data
obtaining a pointer to access. 4-18
memory objects
accessing directly from the host 4-18
copy 4-18
map mode 4-19
transfer policy. 4-20
create 4-18
default. 4-20
enabling zero copy. 4-18
location 4-16
modifying 4-20
passing data to kernels 4-13
runtime
limits 4-14
policy 4-13
runtime policy
best performance practices 4-13
transferring data to and from the host. . 4-18
zero copy 4-18
mapping 4-19
optimizing data movement. 4-19
support 4-19
zero copy host resident
boosting performance 4-19
memory stride
description of 5-2, 6-6
menu
viewing
IL 4-10
ISA 4-10
source OpenCL 4-10
metadata structures
holding debug information 2-2
OpenCL-specific information 2-2
Microsoft Visual Studio
AMD APP Profiler
viewing IL and ISA code 4-10
microtile. 5-26
minGW
GDB running. 3-4
motion estimation algorithms
SAD 5-24, 6-42
MULs. 5-24, 6-42
multi-core
runtimes
Cilk 4-32, 6-36
schedulers 4-32, 6-36
multi-GPU environment
use of GLUT. G-7
multiple devices
creating a separate queue for each
device 4-32, 6-35
in OpenCL runtime. 4-29, 6-33
optimization. 4-1, 5-1, 6-1
partitioning work for 4-32, 6-35
when to use 4-31, 6-35

N

naive compute kernel
kernel structure. C-3
matrix transpose. C-2
naïve matrix transpose C-2
naive pixel shader
kernel structure. C-2
matrix transpose. C-2
namespaces
C++ support for 7-4
supported feature in C++. 7-1
naming conventions
API extension functions A-1
elements contained in extensions A-1
enumerated values. A-1
extensions. A-1
Khronos Group approved A-1
provided by a specific vendor A-1
provided by multiple vendors. A-1
native data type
L1 cache. 5-15, 6-22
native format
LDS cache 5-15, 6-22
native speedup factor
for certain functions 6-43
native_function math library 5-24, 6-42
n-dimensional grid (ND-Range) 1-8
n-dimensional index space
NDRange 1-7
NDRange
accumulation operations 1-17
balancing the workload 5-16, 6-23
defnition 1-17
dimensions 5-21, 6-31

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

exploiting performance of the GPU 5-16, 6-23
general guidelines
determining optimization 5-22, 6-32
global
divided into local ranges 5-16, 6-23
index space
combining work-items 6-30
input streams 1-17
n-dimensional index space 1-7
optimization 5-16, 6-23
summary 5-22, 6-32
partitioning work 5-20, 6-28
profiler reports the dimensions
GlobalWorkSize field 4-12
random-access functionality 1-17
reduction operations 1-17
variable output counts 1-17
nesting
if blocks 5-32, 6-54
non-active work-items 1-11
non-blocking maps 4-35, 6-38
non-coalesced writes 6-12
quarter-wavefront units accessing the
memory system 5-8, 6-12
normalized addresses 1-13
NULL and non-Null return values
extension functions A-3
NV12 Interleaved
specifying the output format F-3

O

-O0
compiler option 2-4
object files
linking 2-3
occupancy metric 5-17, 6-24
Occupancy Modeler
OpenCL kernel 4-6
offset argument 4-19
on-chip transpose
LDS outperforms pixel shader and
compute kernel C-4
Open Decode
context creation
allocating buffers F-2
initializing F-2
decompression
GPU F-1
five step process F-2
creating the context F-2
creating the decode session F-3
decode execution F-3
initializing F-2
releasing the resources and closing the
session F-4
Open Decode API
basic tutorial F-1
fixed-function decoding F-1
using AMD Unified Video Decoder block . F-1
open platform strategy
AMD Accelerated Parallel Processing . . . 1-7
Open Video Decode API
performing a decode F-1
OpenCL
Accelerated Parallel Processing
implementation 1-6
adding built-in functions to the language
cl_amd_media_ops extension A-7
allocating images 1-20
API 2-1
application scenarios and corresponding
paths for AMD platforms 4-25
applications
execution 2-6
avoiding over-allocating device memory 4-16
balancing the workload using multiple
devices 4-29, 6-33
beginning execution
synchronizing command 4-34, 6-37
Binary Image Format (BIF)
overview E-1
buffers 4-22
zero copy 4-23
building
create a context 1-19
programs 2-1
querying the runtime 1-19
the application 1-19
built-in functions
mad24 . . . . 5-24, 6-42
mul24 . . . . 5-24, 6-42
built-in timing capability 4-11
built-ins 5-30, 6-52
C printf A-12
C programming 1-18
checking for known symbols 3-2
C-like language with extensions
for parallel programming 1-18
coding 2-1
commands
copy buffers and images 4-21
read buffers and images 4-21
write buffers and images 4-21
commands are sent
Create Command Queue call F-3

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

compiler
  determining the used path 6-4
  toolchain 2-1
compiler and runtime components. 1-7
compiler options
  -D name 2-4
  -I dir 2-4
compiling
  on Linux 2-3
    linking. 2-3
  on Windows 2-2
  the program 2-2
context
  memory object allocation 1-19
conversion guidelines
  format string A-12
CPU processing 2-1
create kernels within programs 1-20
create one or more command queues . 1-20
create programs to run on one or more
  devices 1-20
create the context. F-2
creating a context
  selecting a device 1-20
creating at least one context 4-36, 6-40
CUDA programming 5-29, 6-51
data-parallel model
  hierarchical subdivision 1-1
debugging. 3-1
  clBuildProgram 3-2
desired platform 1-19
  selection 1-19
directives to enable or disable
  extensions A-2
distributing the kernel. 2-1
enqueued commands. 1-2
ensuring the kernels meet specification . 2-2
Event_Object is returned F-3
extensions
  enabling or disabling A-2
following same pattern 1-20
functions 2-2
general compilation path of applications . 2-1
generating
  .amdil E-3
  .llvmir E-3
  .text. E-3
  a binary E-1
  IL code 4-10
  ISA code. 4-10
global memory 5-8, 6-11
GPU processing 2-1
guidance for CPU programmers . 5-29, 6-52
hardware performance parameters 5-14, 6-20
host program 2-2
ICD code requirements B-1
implementations
  use clGetPlatformIds() function B-1
  use clGetPlatformInfo() function B-1
Installable Client Driver (ICD) B-1
introduction to start using UVD hardware F-1
kernel compiling 2-1
kernel symbols
  not visible in debugger. 3-2
kernels
  FastPath and CompletePath 6-4
limiting number of work-items in each
  group 5-20, 6-29
list
  of available implementations B-1
  of commands 2-5
managing each command queue 4-34, 6-38
mapping 1-8
math libraries
  function () 5-24, 6-42
  native_function () 5-24, 6-42
memory domains 1-11
memory object
  location 4-16
  properties 4-17
metadata structures 2-2
minimalist C program sample 1-20
optimizing
  data transfers 4-21
  register allocation 5-18, 6-27
optional
  extensions A-1
  kernel program. 2-2
partitioning the workload 4-29, 6-33
performance
  libraries components 1-7
  profiling components 1-7
Performance Marker library 4-2
printf capabilities. A-12
programming CPU
  key differences in optimization
    strategy 5-30, 6-52
programming model 1-18, 1-19
  allocating memory buffers 1-20
  executing kernels for specific devices 1-18
  queues of commands 1-18
  reading/writing data 1-18
providing an event 1-19

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

querying
  extensions A-1
  supported extensions using clGetPlatform-
    Info() A-1
read data back to the host from device . 1-20
read-only images
  FastPath 6-4
recompiling LLVM IR to generate a new
  code E-1
regular device buffers 4-22
re-written applications become source . . . 2-1
running
  data-parallel work 1-19
  on multiple devices 4-31, 6-35
  programs 2-1
  task-parallel work 1-19
runtime 2-1
  batching 4-11
  changing options A-6
  interface 2-2
  post-processes the incomplete AMD IL
    from OpenCL compiler 2-1
  recording timestamp information 4-10
  roundtrip chain 4-25
  timing the execution of kernels 4-10
  transfer methods 4-14
  using LLVM AS 2-1
  using multiple devices 4-29, 6-33
runtime policy for memory objects 4-13
  best performance practices 4-13
runtime transfer methods 4-14
sample code
  reading current value of timer clock . 4-11
scheduling asynchronously from a
  command-queue 4-33, 6-37
SDK partitions large number of
  work-groups into smaller pieces 5-20, 6-28
setting breakpoint 3-2
settings for compiling on Windows 2-2
spawning a new thread 4-34, 6-38
stack
  globally scoped constant arrays 5-13, 6-20
  improving performance 5-13, 6-19
  per-pointer attribute 5-13, 6-20
storing AMD IL
  .amdil E-1
storing immediate representation
  (LLVM IR)
  .llvmir E-1
storing OpenCL and driver versions
  .comment E-1
storing source program
  .source E-1
  submit the kernel for execution 1-20
  supported standard
    compiler options 2-4
  supports up to 256 work-items 6-25
  synchronizing a given event 1-19
  timer use with other system timers 4-11
  timestamps 4-10
  tracking time across changes in
    frequency and power states 4-11
  tuning the kernel for the target device . 4-35,
    6-39
  using a separate thread for each
    command-queue 4-33, 6-37
  viewing application statistics 4-4
  work-group sharing not possible . 5-16, 6-23
  write data to device 1-20
OpenCL device
  general overview 1-2
OpenCL kernel
  interactive tuning of 4-8
  Occupancy Modeler 4-6
OpenCL programs
  debugging 3-1
operation
  mem_fence 1-2
operations
  atomic 1-2
  device-specific
    kernel execution 1-19
    program compilation 1-19
  double-precision floating point 1-6
  integer 1-6
  memory-read 1-13
  memory-write 1-13
  single-precision floating point 1-6
optimization
  applying recursively (constant buffer
    pointers in single hardware buffer) . . 5-13
  constant memory
    levels of performance 5-12, 6-19
  key differences
    programming CPU using
      OpenCL 5-30, 6-52
  LDS 5-9, 6-16
    GPU performance 5-30, 6-52
  NDRange
    general guidelines 5-22, 6-32
  of execution range 5-16, 6-23
  of GPU global memory system 5-1, 6-1
  of local memory usage
    LDSBankConflict 5-10, 6-18
  of NDRange 5-16, 6-23

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

of register allocation
special attribute 5-18, 6-27
of the Cedar part 6-32
when programming
AMD Accelerated Parallel Processing. 4-1
compute devices 4-1, 5-1, 6-1
CPUs 4-1, 5-1, 6-1
multiple devices 4-1, 5-1, 6-1
work-group size 5-11, 6-19
optimized matrix transpose
LDS. C-2
optimizer
transfer of data. 2-2
optimizing
application performance with Profiler. . . . 4-1
optional extensions
for OpenCL. A-1
output domain
Euclidean space C-1
indexing mode C-1
output results
view and debug 4-2
overloading
in C++ language. 7-4
kernel 7-2
kernel and function. 7-1
overview
software and hardware. 1-1

P

Packed 16-bit and 8-bit operations
not natively supported 5-24
packing order
work-items following the same direction when
control-flow is encountered . . . 5-22, 6-31
page
pinning 4-14
unpinning 4-14
page boundary
memory alignment 1-13
parallel min() function
code sample. 1-25
example programs 1-23
kernel code. 1-24
programming techniques 1-23
runtime code. 1-23
steps. 1-23
parallel programming
memory fence
barriers 1-18
operations. 1-18
parallelism
enqueuing
multiple tasks 1-1
native kernels. 1-1
unrolling the loop to expose. 6-44
using vector data types 1-1
parallelization
DMA transfers 1-15
GPU 1-15
parameters
for 56xx, 57xx, 58xx, Eyfinity6, and 59xx
devices D-7
for 64xx devices D-5, D-6
for 65xx, 66xx, and 67xx devices D-4
for 68xx and 69xx devices. D-2, D-3
for Exxx, Cxx, 54xx, and 55xx devices . . D-8
partitioning simple static algorithms . 4-32, 6-35
partitioning the workload
guidelines
global level 5-20, 6-28
local level 5-20, 6-28
work 5-20, 6-28
multiple OpenCL devices 4-29, 6-33
NDRange 5-20, 6-28
on multiple devices. 4-32, 6-35
passing a class between host to the device 7-6
path to the kernel file
relative to executable 2-6, 2-7
paths
buffer. 4-21
image 4-21
PathUtilization counter
AMD APP Profiler. 6-5
pattern
characteristics of low-performance 5-3
patterns
transforming multiple into a single
instruction 5-25
PCIe
bus 1-14
communication between system and
GPU 1-14
CPU access of discrete GPU device
memory 4-16
data transfers between system and
GPU 1-14
GPU accessing pinned host memory . . 4-15
kernel instructions 1-14
memory domain 1-11
overview 1-14
skip copying between host memory
and PCIe memory 1-13
throughput. 1-14

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

PCIe bus
slower than host memory 4-16
peak interconnect bandwidth
definition 4-22
performance
affected by dynamic frequency scaling . 4-32
AMD OpenCL stack 5-13, 6-19
better with algorithm that changes
work-group size 5-29
characteristics
CPU 4-29, 6-33
GPU 4-29, 6-33
comparison
compute kernel vs pixel shader C-2
matrix transpose program C-2
CompletePath 6-3
constant memory 5-12, 6-19
counter
LDSBankConflict. 5-10, 6-17
CPU streaming writes 4-15
different device characteristics . . . 4-35, 6-39
experimenting with different loop
types 5-32, 6-54
FastPath 6-3
gain in compute kernel using LDS
feature. C-4
general tips
avoid declaring global arrays . . 5-27, 6-49
avoid writing code with dynamic pointer
assignment 5-28, 6-50
coding a simple and accurate
algorithm. 5-28, 6-50
data set reduction. 5-28, 6-50
loop unrolling 5-28, 6-50
removing or commenting-out
sections of code 5-28, 6-50
use predication rather than
control-flow 5-27, 6-49
GPU vs CPU. 4-31, 6-34
guidance
general tips 5-27, 6-49
improving
kernel attribute syntax 5-18, 6-27
re-ordering the code. 6-43
using float4 6-45
increasing
unrolling the loop 5-32, 6-54
launch dimension 6-8
of a copy 6-16
of the GPU
NDRange 5-16, 6-23
peak on all devices. 4-36, 6-40
recommendations
guidance for CUDA
programmers 5-29, 6-51
results
compute kernel. C-3
LDS. C-4
pixel shader C-2
tips for AMD-specific optimizations 5-25, 6-48
tips for GPU compute. 5-25, 6-48
when programming
AMD Accelerated Parallel
Processing 4-1, 5-1, 6-1
compute devices. 4-1, 5-1, 6-1
CPUs. 4-1, 5-1, 6-1
multiple devices 4-1, 5-1, 6-1
work-groups 1-18
performance characteristics
CPU vs GPU. 4-35
performance counter
ALUBusy 5-17, 6-24
for optimizing local memory
AMD APP Profiler. 5-10, 6-17
performance counters 4-5
Performance Marker library 4-2
per-pointer attribute
improving performance of OpenCL
stack 5-13, 6-20
per-thread registers. 5-30, 6-52
physical memory layouts
for images 5-26, 6-48
memory tiling. 5-26, 6-48
pin
transferring host memory to device
memory 4-14
pinned host memory 4-14
accessing through the PCIe bus 4-15
allocating memory. 4-21
CPU caching 4-15
improved transfer performance 4-15
initializing with passed data 4-21
runtime makes accessible 4-15
pinned memory 4-20
pinning
cost minimizing 4-20
definition 4-22
pinning cost 4-25
pipelining 1-13
pixel shader C-2
differences from compute kernel C-1
hierarchical-Z pattern C-1
indexing mode
vWinCoord register. C-1
kernel structure C-2

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

linear pattern C-1
matrix transpose. C-2
performance comparison C-2
performance results of matrix transpose . C-2
platform vendor string
remains constant for a particular vendor's
implementation. B-3
searching for desired OpenCL platform. . B-3
vs platform name string B-3
point (barrier) in the code. 1-18
population count
extension A-7
porting code
toGPU unchanged 5-2
post-processing incomplete AMD IL 2-1
power of two strides avoidance 5-6, 6-9
pragma unroll 5-26
predication
use rather than control-flow 5-27, 6-49
pre-ICD code snippet B-1, B-3
pre-pinned buffers. 4-24
private memory allocation
mapping to scratch region 5-14, 6-21
private memory domain 1-11
processing by command processors 1-3
processing elements
ALUs. 1-3
in compute unit. 5-2
SIMD arrays 1-3
Profiler
AMD APP 4-1
optimizing application performance with . 4-1
reducing launch overhead 4-11
timeline view. 4-2
two usage models 4-1
program
examples 1-20
simple buffer write 1-20
programming
AMD Accelerated Parallel Processing GPU
optimization 4-1
performance 4-1, 5-1, 6-1
basic steps with minimum code. 1-20
CPUs
performance 4-1, 5-1, 6-1
getting detailed information
after running the profiler 4-10
GPU 1-7
raw compute horsepower 5-30, 6-52
multiple devices
performance 4-1, 5-1, 6-1
task-parallel 1-1
techniques
simple tests
parallel min() function 1-23
programming model
AMD Accelerated Parallel Processing . . . 1-7
OpenCL 1-18, 1-19
executing kernels for specific devices 1-18
queues of commands 1-18
reading/writing data 1-18
project property settings
compiling on Windows 2-2
ptr arguments 4-15, 4-20

Q

quarter-wavefront units
non-coalesced writes 5-8, 6-12
querying
AMD-specific device attributes. A-5
clDeviceInfo
obtaining device memory. 5-20, 6-29
extensions
for a list of devices A-2
for a platform A-1
OpenCL A-1
for a specific device A-2
for available platforms A-2
the platform API B-1
the runtime
OpenCL building 1-19
querying device
when using constant buffers 5-13
queue
command 1-15, 1-19

R

Random Access Target (RAT). 6-5
random memory location
GPU storage of writes 1-13
random-access functionality
NDRange 1-17
read broadcast feature
LDS. 5-15, 6-22
read coalescing. 5-34
read imaging. 1-13
read path
L1 . . . . 5-15, 6-22
read-only buffers
constants 1-14
reads from a fixed address
collide and serialized 5-8
register allocation
preventing launch fails 6-27
register spilling 5-34

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

register spills
ScratchReg field
AMD APP Profiler. 5-18, 6-27
registers
GPU 6-25
per-thread 5-30, 6-52
shared among all active wavefronts on the
compute unit. 6-25
spilled 5-18
vaTid C-1
vs LDS
indexing flexibility 5-10, 6-17
vWinCoord C-1
rendering pipeline
displaying finished video. F-1
reordering data
coalesced writes 6-12
reqd_work_group_size
compiler removes barriers 5-11, 6-19
retiring work-groups. 5-5, 6-8
running code
on Linux 2-7
on Windows 2-6
runtime
change CL options 2-5
code
parallel min() function. 1-23
executing kernels on the device. 4-21
interface
OpenCL. 2-2
knowing data is in pinned host memory 4-15
limits of pinned host memory used for
memory objects 4-14
making pinned host memory accessible 4-15
multi-core
Cilk 4-32, 6-36
OpenCL. 2-1
changing options A-6
pin/unpin on every map/unmap transfer 4-20
pinned host memory staging buffers . . . 4-14
recognizing only data in pinned has
memory. 4-15
system functions 2-5
tracking the map location
copy map mode 4-20
transfers
depending on memory kind of
destination 4-21
source. 4-21
mapping for improved performance . . 4-19
types of memory used 4-13
zero copy buffers 4-23

S

same index
constant memory performance. . . 5-12, 6-19
same-indexed constants
caching 5-14, 6-21
sample code
computing the kernel execution time . . . 4-10
for reading the current value of OpenCL
timer clock 4-11
relationship between
buffer(s). 1-20
command queue(s). 1-20
context(s) 1-20
device(s) 1-20
kernel(s) 1-20
relationship between context(s) 1-20
-save-temps
compiler option 2-4
SC cache. 1-3
scalar instruction
SIMD execution. 1-2
scalar instructions 1-3
scalar unit
advantage of 5-32
scalar unit data cache
SC cache 1-3
scalra instructions 1-3
scattered writes 4-22
scheduler
heterogeneous devices
balancing grain size 4-33, 6-36
conservative work allocation . . 4-33, 6-36
sending different workload sizes
to different devices. 4-33, 6-36
using only the fast device 4-33, 6-36
multi-core 4-32, 6-36
scheduling
across both CPU and GPU
devices 4-32, 6-36
chores
CPU 4-33, 6-37
device fission extension 4-33, 6-37
dynamic
algorithm. 4-32, 6-36
GPU 1-15, 4-32, 6-36
asynchronous launch 4-33, 6-37
data location 4-34, 6-37
heterogeneous compute
devices 4-32, 6-36
wavefronts
compute unit. 5-17, 6-24
latency hiding 6-23

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

work-items
for execution 1-8
range 1-7
scope
global 1-2
scratch region
private memory allocation
mapping 5-14, 6-21
scratchpad memory 5-10, 6-18
ScratchReg field
AMD APP Profiler reports register
spills 5-18, 6-27
SDKUtil library 2-3
linking options 2-3
Linux 2-3
select () function
replacing clauses
with conditional assignments . . 5-31, 6-53
sequential access pattern
uses only half the banks on each cyle . 5-10
set
a breakpoint 3-2
SGPR
limiting number of wavefronts 4-7
SGPRs
use of 5-33
shader architecture
unified 1-7
Shader Resource View (SRV) 5-8, 6-11
shaders and kernels 1-7
Shared Registers
pixel shader C-1
SI ISA 5-33
SIMD 5-16, 6-23
SIMD arrays
processing elements 1-3
simple buffer write
code sample 1-21
example programs 1-20
simple direct-addressing patterns
constant memory performance . . 5-12, 6-19
simple static partitioning algorithms . 4-32, 6-35
simple stride one access patterns vs
large non-unit strides 5-2, 6-6
simple testing
programming techniques
parallel min function 1-23
single device associated with command
queue 1-19
single stream core execution 1-15
single-GPU environment
CL-GL interoperability G-7
single-precision floating-point
performing operations 1-6
single-precision FMA 5-33
small grain allocations
use at beginning of algorithm 4-33
software
overview 1-1
spawn order
of work-item 1-18
sequential 1-18
spawn rate
pixel shader C-1
spawn/dispatch pattern
compute kernel C-1
spawning a new thread
in OpenCL to manage each command
queue 4-34, 6-38
spill code
avoiding
kernel attribute syntax 5-18, 6-27
generated by the compiler 5-18, 6-27
spilled registers 5-18
SSE
packing 6-43
supporting instructions that write parts
of a register 4-30, 6-34
SSE instructions
generating efficient and packed . . 5-30, 6-52
staggered offsets
applying a coordinate transformation to the
kernel 5-6, 6-9
processing data in a different order . 5-6, 6-9
transformation 5-6, 5-7, 6-9
staging buffers
cost of copying to 4-14
stalls
memory fetch request 1-16
start-up time
CPU vs GPU 4-32
static analysis tool
KernelAnalyzer 4-8
static C++ kernel language 7-1
statistics
viewing of OpenCL application 4-4
stdout stream
writing output associated with the host
application A-12
STORE instructions 6-46
address calculation 6-46
stream core
compute units 1-5
executing kernels 1-5
idle 1-11

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

instruction sequence 1-6
processing elements. 1-5
scheduling wavefronts onto 6-1
stall 1-16
due to data dependency 1-17
stream kernel 1-13
stream processor
generating requests 5-10, 6-17
kernels
analyzing. 4-9
strides
power of two
avoiding. 5-6, 6-9
simple and large non-unit. 5-2, 6-6
structure
of kernel
naive compute kernel C-3
naive pixel shader C-2
summary
API, context,kernel, transfer,warnings . . . 4-4
Sum-of-Absolute Differences (SAD)
motion estimation 5-24, 6-42
supplemental compiler options A-6
synchronization
caveats 4-34, 6-37
command-queue barrier 1-2
confirm proper. 4-3
domains. 1-1
command-queue. 1-1
work-items. 1-1
events 1-2
global. 1-2
points
in a kernel. 1-18
synchronizing
a given event 1-19
event
enforce the correct order of execution 1-19
through barrier operations
work-items. 1-19
through fence operations
work-items. 1-19
syntax
_local. 5-10, 6-18
GCC option E-3
kernel attribute
avoiding spill code and improving
performance 5-18, 6-27
system
GFLOPS 1-13
pinned memory 1-13

T

Tahiti
see 79XX series devices or AMD Radeon HD 79XX. 1-2, 1-3
target device characteristics
determining work-size. 5-21, 6-29
task-parallel programming model 1-1
templates
C++ 7-5
kernel, member, default argument,
limited class, partial 7-1
terminology 1-17
TEX
instruction sequence meaning 6-5
texture system
caching 1-14
thread
launching. 1-23
threading
CPU vs GPU. 4-30, 6-34
device-optimal access pattern 1-23
GPU performance. 5-16, 6-23
threads
assigning a fixed set of architectural
registers
CPU 6-25
CPU device supports small
number 4-36, 6-40
GPU
single-cycle switching. 4-30, 6-34
throughput
PCIe 1-14
throughput of instructions
GPUs. 5-23, 6-41
tiled layout format 5-26, 6-48
tiled memory layouts. 5-26
timeline view
Profiler. 4-2
timer
resolution. 4-11
timer resolution 4-11
timestamps
CL_PROFILING_COMMAND_END 4-10
CL_PROFILING_COMMAND_QUEUED 4-10
CL_PROFILING_COMMAND_START . . 4-10
CL_PROFILING_COMMAND_SUBMIT . 4-10
in OpenCL. 4-10
information
OpenCL runtime 4-10
profiling 4-11
timing
API calls 4-3

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

built-in
OpenCL 4-11
of simplified execution of work-items
single stream core 1-15
the execution of kernels
OpenCL runtime. 4-10
toolchain
compiler 2-1
tools
examining amount of LDS used by the
kernel 5-20, 6-28
tools used to examine registers
AMD APP KernelAnalyzer 5-18, 6-26
AMD APP Profiler. 5-18, 6-26
ISA dump 5-18, 6-26
used by the kernel 5-18, 6-26
trace
data of API call. 4-5
Trace View
API 4-5
transcendental
core. 1-6
performing operations. 1-6
transfer
between device and host
command queue elements 1-14
cost of images 4-21
data
float4 format 6-11
select a device 1-20
to the optimizer 2-2
DMA 1-15
from system to GPU
command processor. 1-14
DMA engine 1-14
management
memory 1-13
work-item to fetch unit 1-13
transfer time
data. 4-2
transformation to staggered offsets . . . 5-7, 6-9
tuning of kernel
interactive 4-8
tutorial
Open Decode API F-1

U

unaligned access
bandwidths 6-15
using float1 6-15
uncached accesses 4-22
uncached speculative write combine 4-22
unified shader architecture 1-7
unit of work on AMD GPUs
wavefront 5-21, 6-29
unit stride
computations 5-3, 6-6
performing computations 5-2, 6-6
un-normalized addresses 1-13
Unordered Access View (UAV) 6-5
mapping user data 6-4
memory. 5-8, 6-11
unroll pragma 5-26
unrolling loop 5-25
unrolling the loop
4x 6-43
with clustered stores 6-44
user space program
requesting work-items C-1
USWC, uncached speculative write
combine 4-22
UVD clock
set to idle state. F-5
UVD decode function
starting decode execution F-3
UVD hardware
introduction to using. F-1
UVD hardware and firmware
initializing F-2

V

vAbsTid
3D indexing variables. C-1
vAbsTidFlat
1D indexing variables. C-1
variable output counts
NDRange 1-17
variadic arguments
use of in the built-in printf A-12
varying index
constant memory performance . . 5-13, 6-19
varying-indexed constants paths . . . 5-14, 6-21
vaTid
register indexing
absolute work-item id. C-1
settings C-1
vector data types
parallelism. 1-1
vector instructions. 1-3, 5-33
vector units 1-2
vectorization 5-34
CUDA 6-51
using float4. 6-45
vendor
platform vendor string B-3

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

vendor name
  matching platform vendor string 1-19
vendor-specific extensions
  AMD A-4
vertex fetch
  vfetch instruction 6-5
Very Long Instruction Word (VLIW)
  5-wide processing engine
    moving work into the kernel 6-29
  instruction 1-6
    work-item 1-9
  packing 6-43
    instructions into the slots 6-44
  processor
    five-way 6-1
  programming with 5-wide instruction . . . 6-43
VFETCH
  instruction sequence meaning 6-5
vfetch instruction 6-5
  vertex fetch 6-5
VGPRs 5-34
  limiting number of wavefronts 4-7
video
  displaying
    rendering pipeline F-1
Visual Studio 2008 Professional Edition 3-4
  compiling OpenCL on Windows 2-2
  developing application code 3-4
VRAM
  global memory 1-13
vTgroupid
  indexing variable C-1
vThreadGrpId
  3D indexing variables C-1
vThreadGrpIdFlat
  1D indexing variables C-1
vTid indexing variable C-1
vTidInGrp
  3D indexing variables C-1
vTidInGrpFlat
  1D indexing variables C-1
vWinCoord register
  indexing mode
    pixel shader C-1

W

wait commands 5-33
WAIT_ACK
  instruction sequence meaning 6-5
watermark
  additional scheduling
    reducing or eliminating device
      starvation 4-33, 6-37
wavefront
  accessing all the channels
    inefficient access pattern 5-6, 6-8
  block of work-items 1-9
  combining paths 1-10
  compute unit processes 5-16, 6-24
  compute unit supports up to 32
    OpenCL 6-25
  concept relating to compute kernels . . . 1-18
  creation
    pixel shader C-1
  definition 1-4, 1-10
  executing as an atomic unit 5-11, 6-19
  fully populated
    selecting work-group size 5-22, 6-32
  fundamental unit of work
    AMD GPU 5-21, 6-29
  generating bank conflicts and stalling . . 6-17
  global limits 6-25
    for the ATI Radeon HD 5870 6-25
  GPU reprocesses 5-9, 6-17
  granularity 1-10
  hiding latency 5-17, 5-22, 6-24, 6-32
  idle until memory request
    completes 5-16, 6-23
  latency hiding 5-16, 6-23
  LDS usage effects 5-19, 6-28
  mask 1-10
  masking 1-10
  number of work-items 1-10
  one access one channel 5-5
  pipelining work-items on a stream core . . 1-9
  providing at least two per compute unit . 6-32
  registers shared among all active wavefronts
    on the compute unit 6-25
  relationship to work-group 1-10
  relationship with work-groups 1-10
  required number spawned by GPU 1-11
  same quarter
    work-items 5-21, 6-31
  scheduling
    even and odd 6-46
    on ATI Radeon HD 5000 series
      GPU 5-2, 6-1
    onto stream cores 6-1
  size 1-9
    vs work-group size 5-11, 6-19
  size for optimum hardware usage 1-10
  size on AMD GPUs 1-9
  switching
    on clause boundaries 6-46
    to another clause 6-46
  total execution time 1-10

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

work-group 1-8, 1-18
work-item processing 1-9
work-items execute in lock-step. 5-2
wavefront/compute unit
global limits controlled by the
developer 5-18, 6-25
impact of register type 6-25
occupancy metric 5-17, 6-24
wavefront-per-SIMD count
use of LDS 5-16, 6-23
wavefronts
access consecutive groups 5-6
computing number per CU. 5-18
determining how many to hide latency . 5-17
limited by LDS 4-8
limited by SGPR. 4-7
limited by VGPRs 4-7
multiples should access different
channels 5-6
number of active limitied by work-group . 4-7
while loops
vs for loops. 5-32
Windows
calling convention. 2-7
compiling
Intel C (C++) compiler 2-2
OpenCL 2-2
Visual Studio 2008 Professional
Edition 2-2
debugging
OpenCL kernels 3-4
running code. 2-6
settings for compiling OpenCL. 2-2
work/kernel level for partitioning work 6-29
work-group
allocation of LDS size 1-18
and available compute units 5-20, 6-29
barriers 1-1
blocking strategy
when accessing a tiled image . 5-27, 6-49
composed of wavefronts 1-18
compute unit supports a maximum of
eight 6-24
concept relating to compute kernels . . . 1-18
defined 1-10
dimensions vs size. 5-21, 6-31
dispatching in a linear order
HD 5000 series GPU. 6-7
dividing global work-size into
sub-domains. 1-19
dividing work-items. 1-8
executing 2D
four number identification 5-7, 6-10
executing on a single compute unit. 5-2, 6-1
initiating order. 5-7, 6-10
limited number of active
LDS allocations 5-19, 6-27
maximum size can be obtained 6-32
moving work to kernel 6-29
number of wavefronts in 1-8
optimization
wavefront size 5-11, 6-19
partitioning into smaller pieces for
processing 5-20, 6-28
performance 1-18
processing a block in column-order 5-7, 6-10
processing increased on the fixed
pool of local memory 6-30
relationship to wavefront 1-10
relationship with wavefronts. 1-10
retiring in order. 5-5, 6-8
selecting size
wavefronts are fully populated. 5-22, 6-32
sharing not possible 5-16, 6-23
size
CUDA code 5-29, 6-51
second-order effects 5-21, 6-31
square 16x16 . . . . 5-22, 6-31
size limiting number of active wavefronts 4-7
specifying
default size at compile-time. . . 5-18, 6-27
wavefronts 1-8
staggering. 5-7, 6-10
avoiding channel conflicts 5-7, 6-10
tuning dimensions specified at
launch time. 5-18, 6-27
work-item
reaching point (barrier) in the code. . 1-18
sharing data through LDS
memory 5-21, 6-29
using high-speed local atomic
operations 5-21, 6-29
work-groups
assigned to CUs as needed 5-5
dispatching on HD 7000 series 5-5
no limit in OpenCL 5-20
work-item C-1
barriers 6-18
branch granularity. 1-10
communicating
through globally shared memory 1-19
through locally shared memory 1-19
creation. 1-11
deactivation. 1-13
dividing into work-groups 1-8

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.

does not write
coalesce detection ignores it 6-13
element 1-19
encountering barriers 1-1
executing
on a single processing element . . 5-2, 6-1
on same cycle in the processing
engine. 5-21, 6-31
the branch. 1-10
execution in lock-step 5-2
id C-1
kernel running on compute unit 1-7
limiting number in each group . . . 5-20, 6-29
mapping
onto n-dimensional grid (ND-Range) . . 1-8
to stream cores. 1-7
NDRange index space 6-30
non-active 1-11
number of registers used by 5-18
OpenCL supports up to 256 6-25
packing order 5-22, 6-31
processing wavefront 1-9
reaching point (barrier) in the code 1-18
read or write adjacent memory
addresses 5-3, 6-6
reading in a single value 5-8, 6-11
requested by user space program C-1
same wavefront
executing same instruction on each
cycle 5-22, 6-31
same program counter 5-22, 6-31
scheduling
for execution 1-8
on a GPU 6-1
the range of 1-7
sharing
data through LDS memory 5-21, 6-29
LDS 6-23
spawn order 1-18
synchronization
through barrier operations 1-19
through fence operations 1-19
typical access pattern 5-11
using high-speed local atomic
operations 5-21, 6-29
VLIW instruction 1-9
work-items
divergence in wavefront 1-10
execution of. 1-16
hardware limit 1-15
making up wavefront 1-10
number equal to product of all work-group
dimensions 5-20
pipelining on a stream core 1-9
reference consecutive memory
addresses 5-5
workload
execution
GPU 5-30, 6-52
workload balancing 4-32
write coalescing. 6-13
Write Combine (WC)
global memory system 5-1, 6-1
WriteInsts counters
AMD APP Profiler 4-12

X

X Window system
using for CL-GL interoperability G-9

Z

zero copy. 4-18
direct GPU access to 4-28
direct host access to. 4-27
performance boost 4-19
under Linux 4-19
when creating memory objects. 4-18
zero copy buffer
available buffer types 4-23
calling 4-23
size limit per buffer 4-23
zero copy buffers 4-23
runtime 4-23
zero copy memory objects 4-18
host resident
boosting performance. 4-19
mapping 4-19
optimizing data movement 4-19
support 4-19
zero copy on Fusion systems. 4-24

Copyright © 2012 Advanced Micro Devices, Inc. All rights reserved.