Programming Guide

AMD Accelerated Parallel Processing TECHNOLOGY

AMD Accelerated Parallel Processing

OpenCL™

June 2011

rev1.3c

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 performance and optimization when programming for AMD Accelerated Parallel Processing 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). The last section of this book is a glossary of acronyms and terms, as well as an index.

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/

Contact Information

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

Contents

Figures

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 executed on multiple elements of a memory object. These elements are in an index space,[1] which defines how the execution maps onto the work-items. In the OpenCL data-parallel model, it is not a strict requirement that there be one work-item for every element in a memory object over which a kernel is executed in parallel.

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 independent 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. See section 3.2, “Execution Model,” of the *OpenCL Specification*.

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 the work-group barrier. Either all of, or none of, the work-items in a work-group must encounter the barrier. As currently defined in the OpenCL Specification, global synchronization is not allowed.

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

Figure 1.1 shows a simplified block diagram of a generalized GPU compute device.

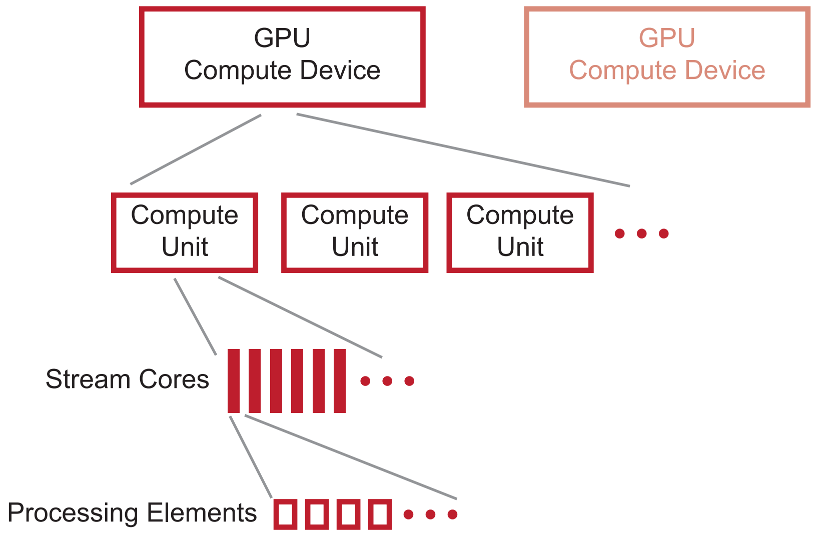

Figure 1.1 Generalized GPU Compute Device Structure

Figure 1.2 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.

GPU compute devices comprise groups of compute units (see Figure 1.1). Each compute unit contains numerous stream cores, which are responsible for executing kernels, each operating on an independent data stream. Stream cores, in turn, contain numerous processing elements, which are the fundamental, programmable computational units that perform integer, single-precision floating-point, double-precision floating-point, and transcendental operations. All stream cores within a compute unit execute the same instruction sequence in lock-step; different compute units can execute different instructions.

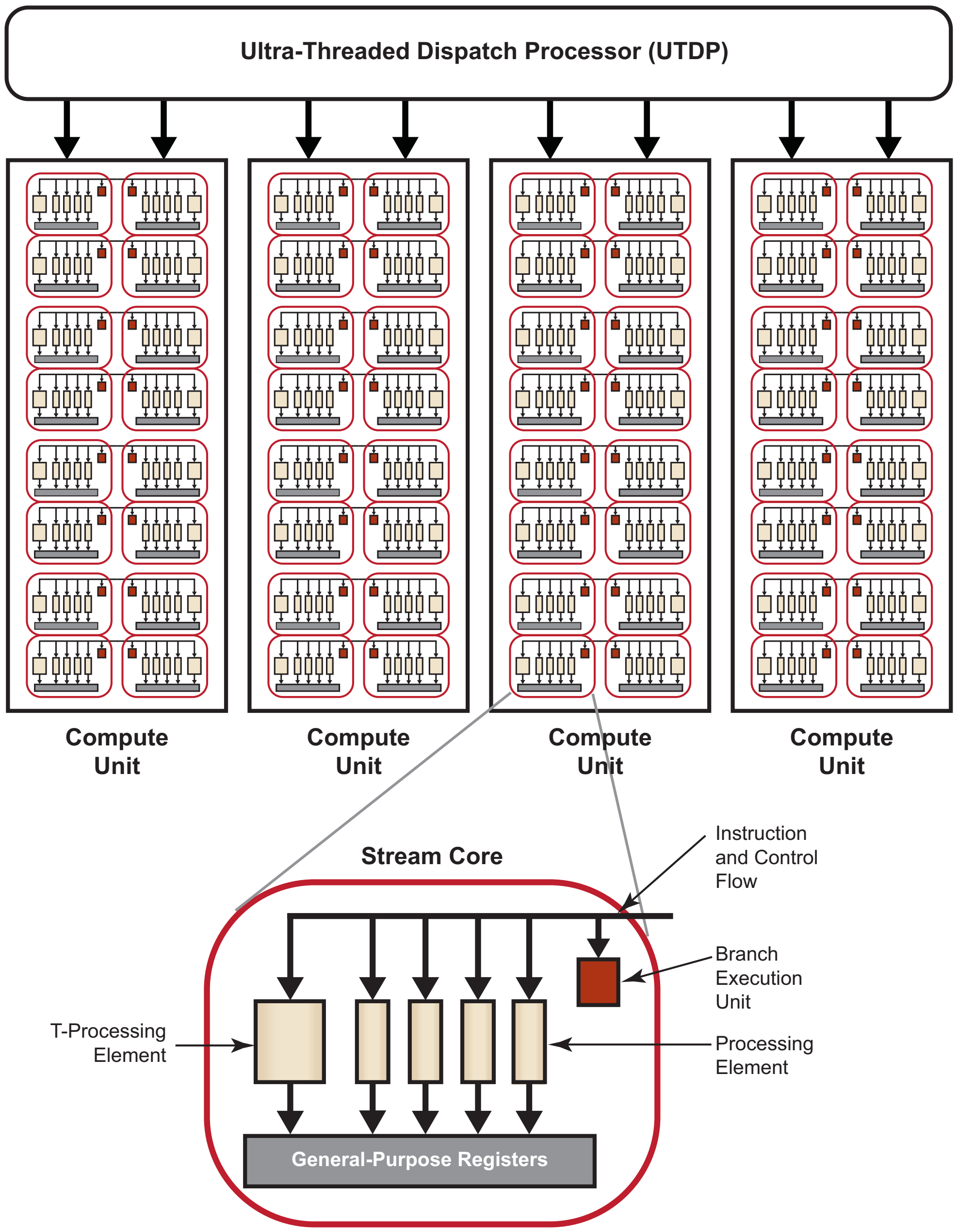

Figure 1.2 Simplified Block Diagram of the GPU Compute Device[1]

1. Much of this is transparent to the programmer.

A stream core is arranged as a five-way or four-way (depending on the GPU type) very long instruction word (VLIW) processor (see bottom of Figure 1.2). 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 processing elements. Processing elements can execute single-precision floating point or integer operations. One of the five processing elements also can perform transcendental operations (sine, cosine, logarithm, etc.)[1]. Double-precision floating point operations are processed by connecting two or four of the processing elements (excluding the transcendental core) to perform a single double-precision operation. The stream core also contains one branch execution unit to handle branch instructions.

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

1.3 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 1.3 illustrates the relationship of the AMD Accelerated Parallel Processing components.

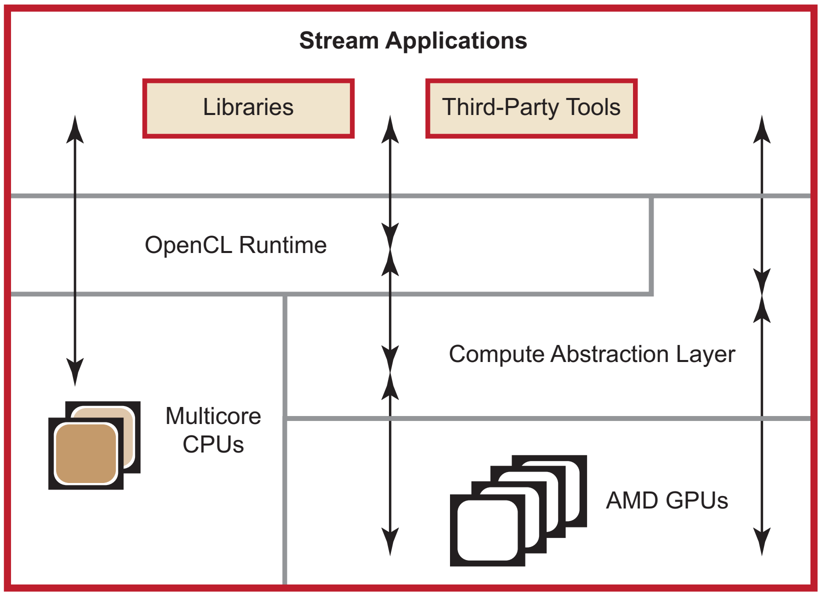

Figure 1.3 AMD Accelerated Parallel Processing Software Ecosystem

1. For a more detailed explanation of operations, see the AMD *Compute Abstraction Layer (CAL) Programming Guide*.

The AMD Accelerated Parallel Processing software stack provides end-users and developers with a complete, flexible suite of tools to leverage the processing 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
- Device Driver for GPU compute device – AMD Compute Abstraction Layer (CAL).[1]
- Performance Profiling Tools – AMD APP Profiler and AMD APP KernelAnalyzer.
- Performance Libraries – AMD Core Math Library (ACML) for optimized NDRange-specific algorithms.

The latest generation of AMD GPUs are programmed using the unified shader programming model. Programmable GPU compute devices execute various user-developed programs, called *stream kernels* (or simply: 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 stream cores, 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 stream cores is shown in Figure 1.4.

1. See the *AMD Compute Abstraction Layer (CAL) Programming Guide*.

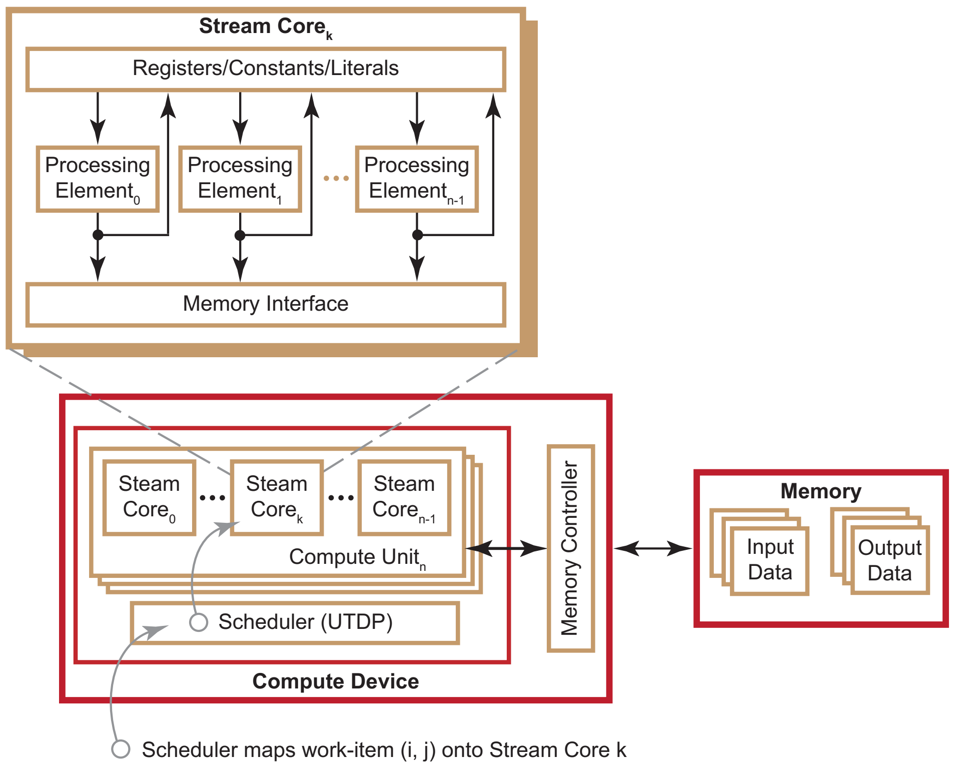

Figure 1.4 Simplified Mapping of OpenCL onto AMD Accelerated Parallel Processing

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 1.5, 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 1.7.2, “Wavefronts and Workgroups,” page 1-17.

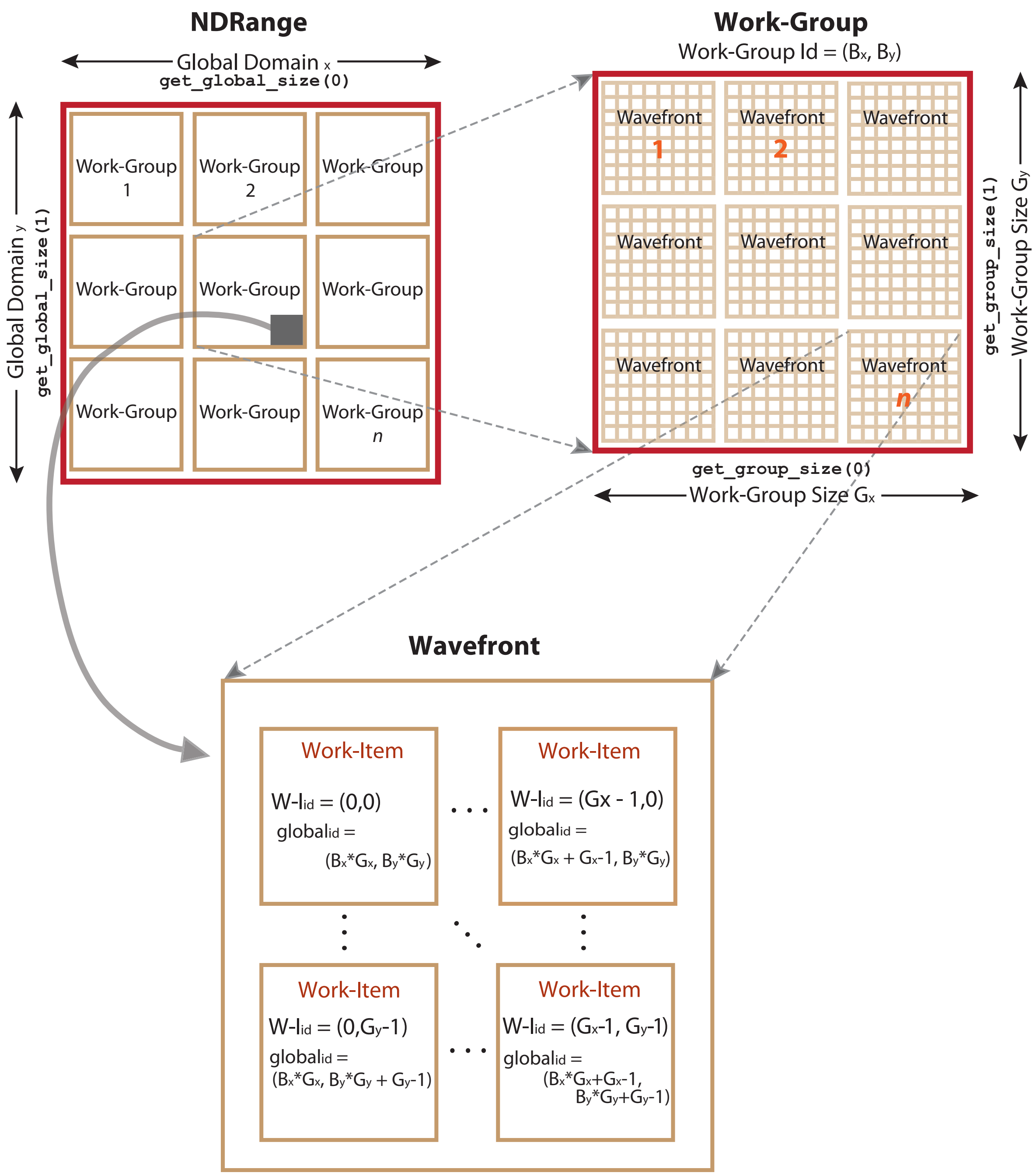

Figure 1.5 Work-Item Grouping Into Work-Groups and Wavefronts

1.3.1 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 ATI Radeon™ HD 5870 GPU compute device, the 16 stream cores 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 ATI Radeon™ HD 5400 series graphics cards has a wavefront size of 32 work-items. The ATI Radeon™ HD 5800 series has a wavefront size of 64 work-items.

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

1.3.2 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. Wavefronts are units of execution, where one wavefront consists of 64 or fewer work-items, two wavefronts would be 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.

<u>Example 2</u>: 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*.

1.3.3 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).

1.3.4 AMD Compute Abstraction Layer (CAL)

The AMD Compute Abstraction Layer (CAL) is a device driver library that provides a forward-compatible interface to AMD GPU compute devices (see Figure 1.6). CAL lets software developers interact with the GPU compute devices at the lowest-level for optimized performance, while maintaining forward compatibility. CAL provides:

- Device-specific code generation
- Device management
- Resource management
- Kernel loading and execution
- Multi-device support
- Interoperability with 3D graphics APIs

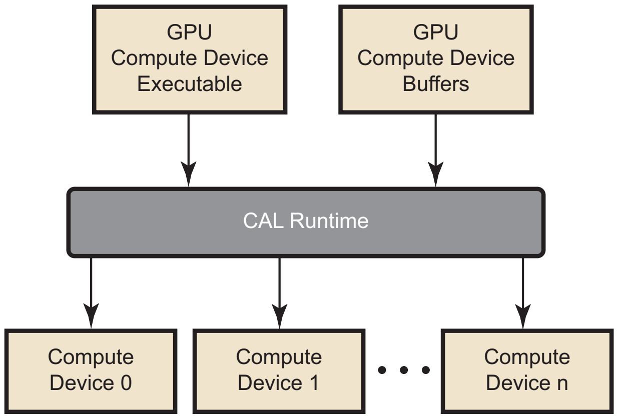

Figure 1.6 CAL Functionality

CAL includes a set of C routines and data types that allow higher-level software tools to control hardware memory buffers (device-level streams) and GPU compute device programs (device-level kernels). The CAL runtime accepts kernels written in AMD Intermediate Language (IL) and generates optimized code for the target architecture. It also provides access to device-specific features.

1.4 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 1.7 illustrates the interrelationship of the memories. (Note that the referenced color buffer is a write-only output buffer ina a pixel shader that ahas a predetermined location based on the pixel location.)

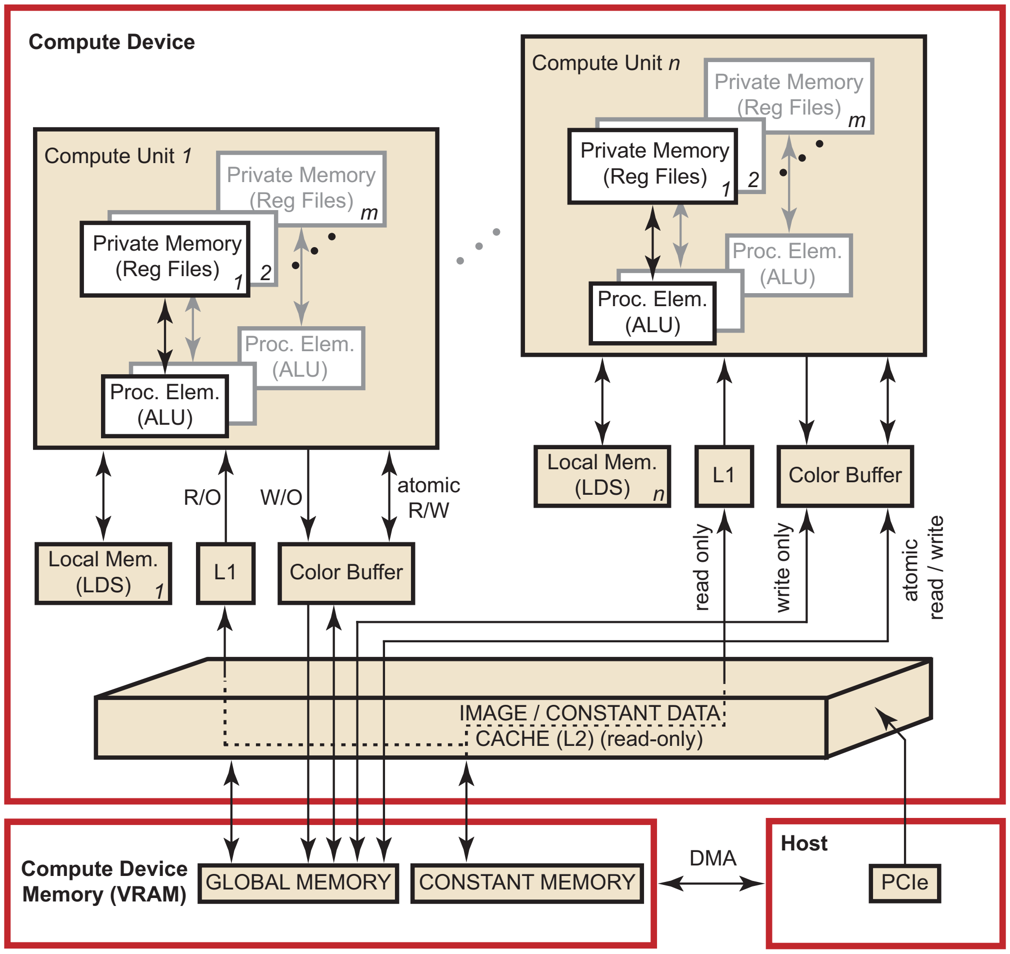

Figure 1.7 Interrelationship of Memory Domains

Figure 1.8 illustrates the standard dataflow between host (CPU) and GPU.

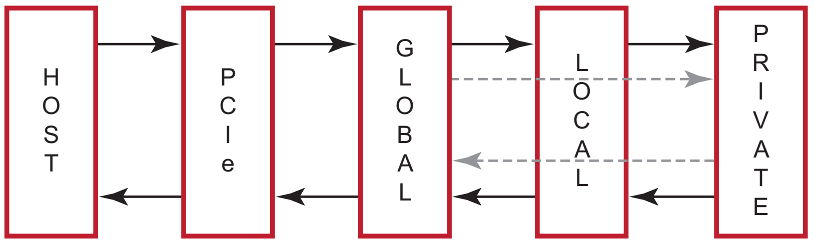

Figure 1.8 Dataflow between Host and GPU

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`.).

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, “OpenCL Performance and Optimization,” for more specifics about optimization techniques.

1.4.1 Memory Access

Using local memory (known as local data store, or LDS, as shown in Figure 1.7) 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.

1.4.2 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.

1.4.3 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.

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

1.4.4 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.

1.5 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.

1.5.1 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. 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.

1.5.2 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

1.5.3 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 device operations. 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 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 1.9 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 being executed. An optimally programmed GPU compute device typically has thousands of active work-items.

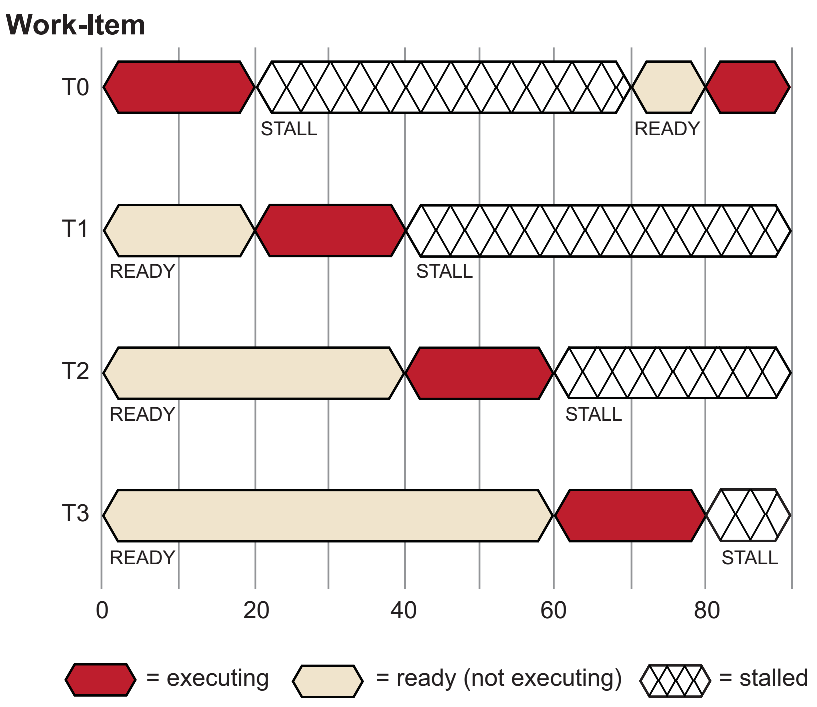

Figure 1.9 Simplified Execution Of Work-Items On A Single Stream Core

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 1.9, 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 1.10, T0 is the first to continue execution.

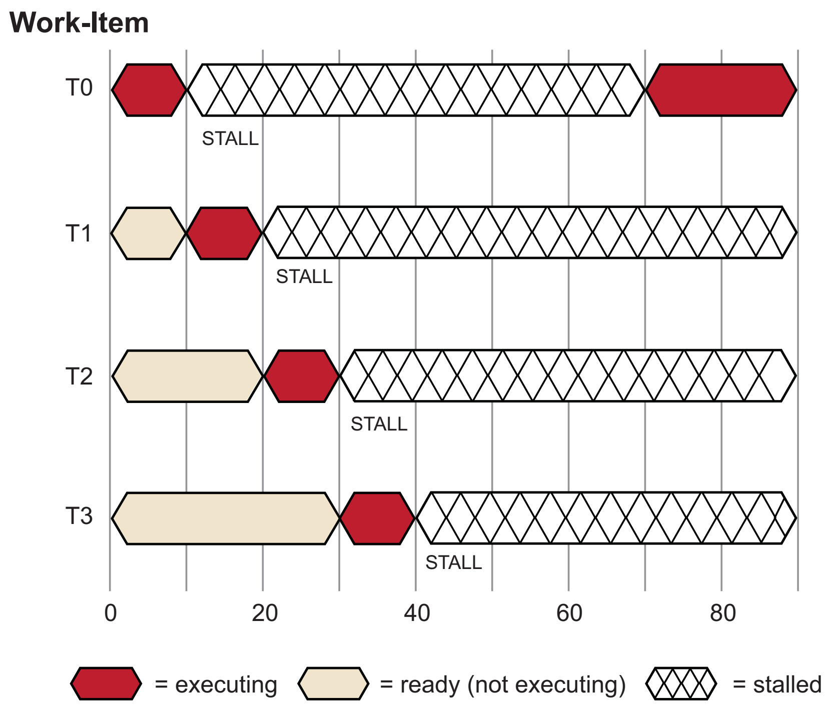

Figure 1.10 Stream Core Stall Due to Data Dependency

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

1.7 Terminology

1.7.1 Compute Kernel

To define a *compute kernel*, it is first necessary to define a kernel. A *kernel* is a small, user-developed program that is run repeatedly on a stream of data. It is a parallel function that operates on every element of input streams (called an NDRange). Unless otherwise specified, an AMD compute device program is a kernel composed of a main program and zero or more functions. This also is called 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.

There are multiple kernel types that are executed on an AMD Accelerated Parallel Processing device, including *vertex*, *pixel*, *geometry*, *domain*, *hull*, and now *compute*. Before the development of *compute kernels*, pixel shaders were sometimes used for non-graphic computing. Instead of relying on pixel shaders to perform computation, new hardware supports compute kernels, which are a better suited for general computation, and which also can be used to supplement graphical applications, enabling rendering techniques beyond those of the traditional graphics pipeline. 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.

1.7.1.1 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.

1.7.2 Wavefronts and Workgroups

Wavefronts and groups are two concepts relating to compute kernels that provide data-parallel granularity. Wavefronts execute N number of work-items in parallel, where N is specific to the hardware chip (for the ATI Radeon™ HD 5870 series, it is 64). A single instruction is executed over all work-items in a wavefront in parallel. 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.

1.7.3 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 for the ATI Radeon™ HD5XXX series. The constraints of the current LDS model are:

1. All read/writes are 32-bits and dword aligned.
2. 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.
3. Data can only be shared within work-items in a work-group.
4. Memory accesses outside of the work-group result in undefined behavior.

1.8 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 1.11 illustrates this model with queues of commands, reading/writing data, and executing kernels for specific devices.

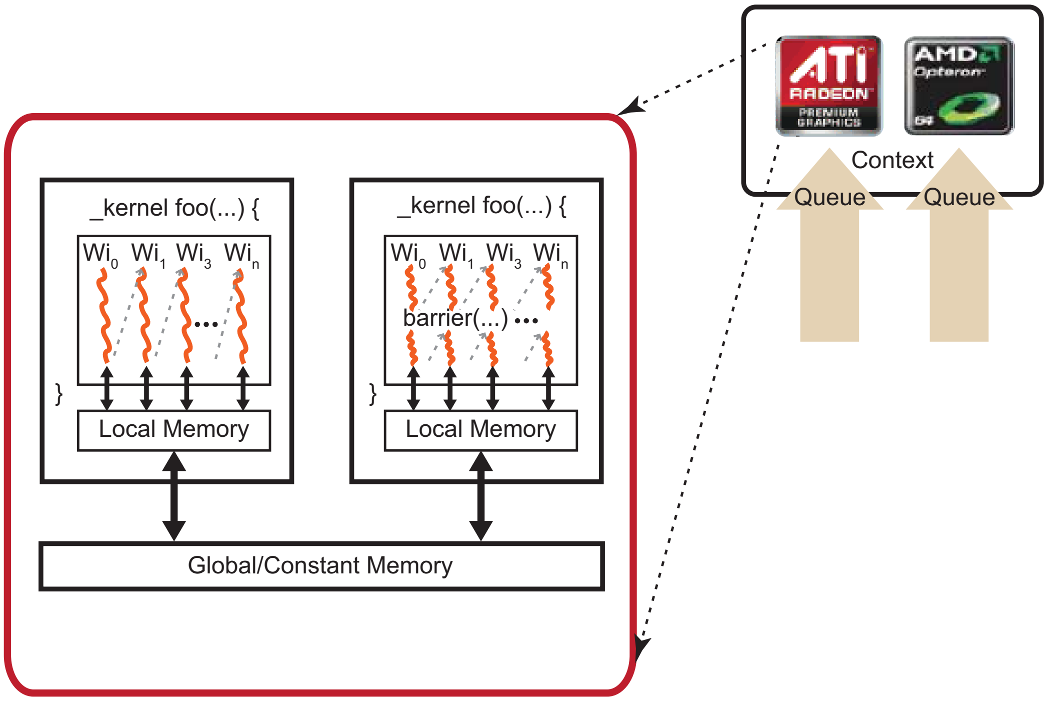

Figure 1.11 OpenCL Programming Model

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 1.11 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 1.11, 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 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.

1.9 Example Programs

The following subsections provide simple programming examples with explanatory comments.

1.9.1 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 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 on the device.
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);

// 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;
}
```

1.9.2 Second Example: SAXPY Function

This section provides an introductory sample for beginner-level OpenCL programmers using C++ bindings.

The sample implements the SAXPY function (**Y** = *a***X** + **Y**, where **X** and **Y** are vectors, and *a* is a scalar). The full code is reproduced at the end of this section. It uses C++ bindings for OpenCL. These bindings are available in the `CL/cl.hpp` file in the SDK available for AMD Accelerated Parallel Processing; they also are downloadable from the Khronos website: http://www.khronos.org/registry/cl .

The following steps guide you through this example.

1. Enable error checking through the exception handling mechanism in the C++ bindings by using the following define.

   ```
   #define __CL_ENABLE_EXCEPTIONS
   ```

   This removes the need to error check after each OpenCL call. If there is an error, the C++ bindings code throw an exception that is caught at the end of the try block, where we can clean up the host memory allocations. In this example, the C++ objects representing OpenCL resources (`cl::Context`, `cl::CommandQueue`, etc.) are declared as automatic variables, so they do not need to be released. If an OpenCL call returns an error, the error code is defined in the `CL/cl.h` file.

2. The kernel is very simple: each work-item, *i*, does the SAXPY calculation for its corresponding elements **Y**[i] = *a***X**[i] + **Y**[i]. Both **X** and **Y** vectors are stored in global memory; **X** is read-only, **Y** is read-write.

   ```
   __kernel void saxpy(const __global float * X,
                       __global float * Y,
                       const float a)
   {
       uint gid = get_global_id(0);
       Y[gid] = a* X[gid] + Y[gid];
   }
   ```

3. List all platforms on the machine, then select one.

   ```
   cl::Platform::get(&platforms);
   ```

4. Create an OpenCL context on that platform.

   ```
   cl_context_properties cps[3] = { CL_CONTEXT_PLATFORM,
                                    (cl_context_properties)(*iter)(), 0 };
   context = cl::Context(CL_DEVICE_TYPE_GPU, cps);
   ```

5. Get OpenCL devices from the context.

   ```
   devices = context.getInfo<CL_CONTEXT_DEVICES>();
   ```

6. Create an OpenCL command queue.

```
queue = cl::CommandQueue(context, devices[0]);
```

7. Create two buffers, corresponding to the **X** and **Y** vectors. Ensure the host-side buffers, pX and pY, are allocated and initialized. The `CL_MEM_COPY_HOST_PTR` flag instructs the runtime to copy over the contents of the host pointer pX in order to initialize the buffer bufX. The bufX buffer uses the `CL_MEM_READ_ONLY` flag, while bufY requires the `CL_MEM_READ_WRITE` flag.

```
bufX = cl::Buffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                  sizeof(cl_float) * length, pX);
```

8. Create a program object from the kernel source string, build the program for our devices, and create a kernel object corresponding to the SAXPY kernel. (At this point, it is possible to create multiple kernel objects if there are more than one.)

```
cl::Program::Sources sources(1, std::make_pair(kernelStr.c_str(),
                             kernelStr.length()));
program = cl::Program(context, sources);
program.build(devices);
kernel = cl::Kernel(program, "saxpy");
```

9. Enqueue the kernel for execution on the device (GPU in our example).

   Set each argument individually in separate `kernel.setArg()` calls. The arguments, do not need to be set again for subsequent kernel enqueue calls. Reset only those arguments that are to pass a new value to the kernel. Then, enqueue the kernel to the command queue with the appropriate global and local work sizes.

```
kernel.setArg(0, bufX);
kernel.setArg(1, bufY);
kernel.setArg(2, a);
queue.enqueueNDRangeKernel(kernel, cl::NDRange(),
                           cl::NDRange(length), cl::NDRange(64));
```

10. Read back the results from bufY to the host pointer pY. We will make this a blocking call (using the `CL_TRUE` argument) since we do not want to proceed before the kernel has finished execution and we have our results back.

```
    queue.enqueueReadBuffer(bufY, CL_TRUE, 0, length * sizeof(cl_float),
pY);
```

11. Clean up the host resources (pX and pY). OpenCL resources is cleaned up by the C++ bindings support code.

The `catch(cl::Error err)` block handles exceptions thrown by the C++ bindings code. If there is an OpenCL call error, it prints out the name of the call and the error code (codes are defined in `CL/cl.h`). If there is a kernel compilation error, the error code is `CL_BUILD_PROGRAM_FAILURE`, in which case it is necessary to print out the build log.

Example Code 2 –

```
#define __CL_ENABLE_EXCEPTIONS

#include <CL/cl.hpp>
#include <string>
#include <iostream>
#include <string>

using std::cout;
using std::cerr;
using std::endl;
using std::string;

/////////////////////////////////////////////////////////////////
// Helper function to print vector elements
/////////////////////////////////////////////////////////////////
void printVector(const std::string arrayName,
                 const cl_float * arrayData,
                 const unsigned int length)
{
    int numElementsToPrint = (256 < length) ? 256 : length;
    cout << endl << arrayName << ":" << endl;
    for(int i = 0; i < numElementsToPrint; ++i)
        cout << arrayData[i] << " ";
    cout << endl;
}

/////////////////////////////////////////////////////////////////
// Globals
/////////////////////////////////////////////////////////////////
int length        = 256;
cl_float * pX     = NULL;
cl_float * pY     = NULL;
cl_float a        = 2.f;



std::vector<cl::Platform> platforms;
cl::Context        context;
std::vector<cl::Device> devices;
cl::CommandQueue   queue;
cl::Program        program;
cl::Kernel         kernel;
cl::Buffer         bufX;
cl::Buffer         bufY;

/////////////////////////////////////////////////////////////////
// The saxpy kernel
/////////////////////////////////////////////////////////////////
string kernelStr    =
    "__kernel void saxpy(const __global float * x,\n"
    "                    __global float * y,\n"
    "                    const float a)\n"
    "{\n"
    "    uint gid = get_global_id(0);\n"
    "    y[gid] = a* x[gid] + y[gid];\n"
    "}\n";

/////////////////////////////////////////////////////////////////
// Allocate and initialize memory on the host
/////////////////////////////////////////////////////////////////
void initHost()
{
    size_t sizeInBytes = length * sizeof(cl_float);
    pX = (cl_float *) malloc(sizeInBytes);
    if(pX == NULL)
        throw(string("Error: Failed to allocate input memory on host\n"));
    pY = (cl_float *) malloc(sizeInBytes);
    if(pY == NULL)
        throw(string("Error: Failed to allocate input memory on host\n"));

    for(int i = 0; i < length; i++)
    {
        pX[i] = cl_float(i);
        pY[i] = cl_float(length-1-i);
    }

    printVector("X", pX, length);
    printVector("Y", pY, length);
}

/////////////////////////////////////////////////////////////////
// Release host memory
/////////////////////////////////////////////////////////////////
void cleanupHost()
{
    if(pX)
    {
        free(pX);
        pX = NULL;
    }
    if(pY != NULL)
    {
        free(pY);
        pY = NULL;
    }
}


void
main(int argc, char * argv[])






{
    try
    {
        /////////////////////////////////////////////////////////////////
        // Allocate and initialize memory on the host
        /////////////////////////////////////////////////////////////////
        initHost();

        /////////////////////////////////////////////////////////////////
        // Find the platform
        /////////////////////////////////////////////////////////////////
        cl::Platform::get(&platforms);
        std::vector<cl::Platform>::iterator iter;
        for(iter = platforms.begin(); iter != platforms.end(); ++iter)
        {
            if((*iter).getInfo<CL_PLATFORM_VENDOR>() == "Advanced Micro 
Devices, Inc.")
                break;
        }

        /////////////////////////////////////////////////////////////////
        // Create an OpenCL context
        /////////////////////////////////////////////////////////////////
        cl_context_properties cps[3] = { CL_CONTEXT_PLATFORM, 
(cl_context_properties)(*iter)(), 0 };
        context = cl::Context(CL_DEVICE_TYPE_GPU, cps);
        /////////////////////////////////////////////////////////////////
        // Detect OpenCL devices
        /////////////////////////////////////////////////////////////////
        devices = context.getInfo<CL_CONTEXT_DEVICES>();

        /////////////////////////////////////////////////////////////////
        // Create an OpenCL command queue
        /////////////////////////////////////////////////////////////////
        queue = cl::CommandQueue(context, devices[0]);

        /////////////////////////////////////////////////////////////////
        // Create OpenCL memory buffers
        /////////////////////////////////////////////////////////////////
        bufX = cl::Buffer(context,
                          CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                          sizeof(cl_float) * length,
                          pX);
        bufY = cl::Buffer(context,
                          CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
                          sizeof(cl_float) * length,
                          pY);

        /////////////////////////////////////////////////////////////////
        // Load CL file, build CL program object, create CL kernel object
        /////////////////////////////////////////////////////////////////
        cl::Program::Sources sources(1, std::make_pair(kernelStr.c_str(),
                                        kernelStr.length()));
        program = cl::Program(context, sources);
        program.build(devices);
        kernel = cl::Kernel(program, "saxpy");

        /////////////////////////////////////////////////////////////////
        // Set the arguments that will be used for kernel execution
        /////////////////////////////////////////////////////////////////
        kernel.setArg(0, bufX);
        kernel.setArg(1, bufY);
        kernel.setArg(2, a);

        /////////////////////////////////////////////////////////////////
        // Enqueue the kernel to the queue
        // with appropriate global and local work sizes
        /////////////////////////////////////////////////////////////////
        queue.enqueueNDRangeKernel(kernel, cl::NDRange(),
                                   cl::NDRange(length), cl::NDRange(64));

        /////////////////////////////////////////////////////////////////
        // Enqueue blocking call to read back buffer Y
        /////////////////////////////////////////////////////////////////
        queue.enqueueReadBuffer(bufY, CL_TRUE, 0, length *
                                sizeof(cl_float), pY);

        printVector("Y", pY, length);

        /////////////////////////////////////////////////////////////////
        // Release host resources
        /////////////////////////////////////////////////////////////////
        cleanupHost();
    }
    catch (cl::Error err)
    {
        /////////////////////////////////////////////////////////////////
        // Catch OpenCL errors and print log if it is a build error
        /////////////////////////////////////////////////////////////////
        cerr << "ERROR: " << err.what() << "(" << err.err() << ")" <<
           endl;
        if (err.err() == CL_BUILD_PROGRAM_FAILURE)
        {
            string str =
                program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(devices[0]);
            cout << "Program Info: " << str << endl;
        }
        cleanupHost();
    }
    catch(string msg)
    {
        cerr << "Exception caught in main(): " << msg << endl;
        cleanupHost();
    }
}
```

1.9.3 Third 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 should be an order of magnitude higher than that of the CPU, due to the parallelized memory subsystem of the graphics card.

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.

Kernel Code –

9. 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.
10. 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.
11. 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.
12. 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.
13. 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.

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"
"                    size_t         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"
"                                                                       \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;
   }

   // 2. Tell compiler to dump intermediate .il and .isa GPU files.

   putenv("GPU_DUMP_DEVICE_KERNEL=3");

   // 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;
   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);

// 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 );

cl_int ret = clBuildProgram( program, 1, &device, NULL, NULL, NULL );

// 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);

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],
                                                               dbg_ptr[3] );

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

     }

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

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.

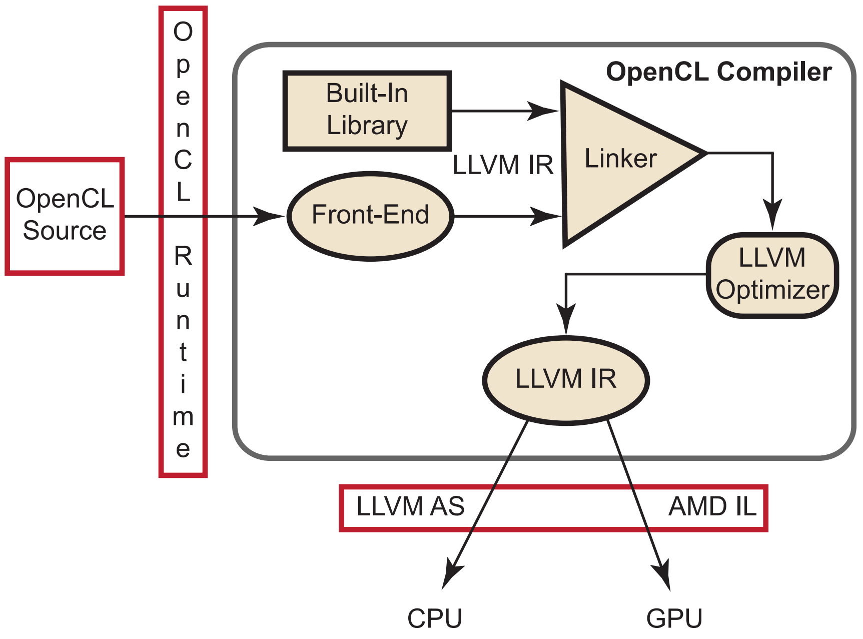

Figure 2.1 OpenCL Compiler Toolchain

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

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`.
- 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
```

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."
- `-f[no]-alias` — Specifies that all global kernel pointers do [not] alias with any other global kernel pointers. This is equivalent to adding `const restrict` to all global pointers if `-fno-alias` is specified.

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()`.

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.

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.

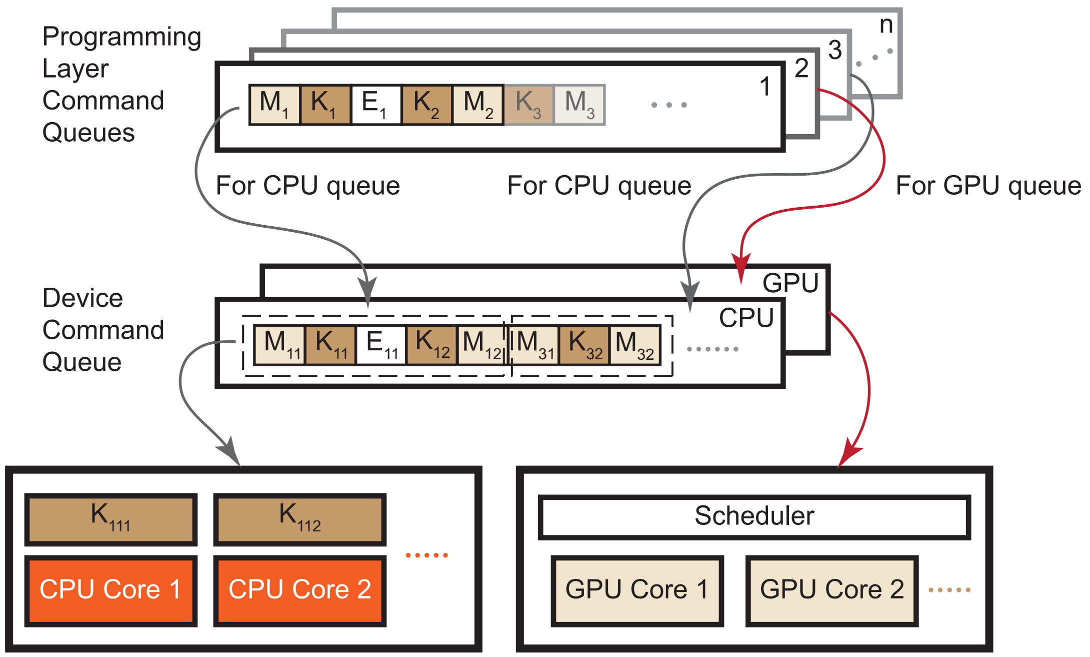

Figure 2.2 Runtime Processing Structure

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`.

2.4 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:

```
__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__)
        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());
}
```

Chapter 3
Debugging OpenCL

AMD Accelerated Parallel Processing provides 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.

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

3.3 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
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)
```

3.4 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.

Chapter 4 OpenCL Performance and Optimization

This chapter discusses performance and optimization when programming for AMD Accelerated Parallel Processing GPU compute devices, as well as CPUs and multiple devices.

4.1 AMD APP Profiler

The AMD APP Profiler provides a Microsoft® Visual Studio® integrated view of key static kernel characteristics such as workgroup dimensions and memory transfer sizes, as well as kernel execution time, dynamic hardware performance counter information (ALU operations, local bank conflicts), and kernel disassembly. For information on installing the profiler, see the Installation Notes for AMD Accelerated Parallel Processing. The performance counters available through the Profiler are listed in Table 4.1.

After following the installation instructions, you can run the AMD APP Profiler from within Visual Studio to generate profile information for your application. After verifying that the application compiles and runs successfully, click Start Profiling to generate the profile information. The profiling process may run the application multiple times to generate the complete set of performance information.

Some sections in the remainder of this document reference information and analysis that can be provided by the AMD APP Profiler. See this tool for the available performance counter descriptions.

Table 4.1 lists and briefly describes the performance counters available through the AMD APP Profiler.

Table 4.1 Performance Counter Descriptions

Name	Description
Method	The kernel name (appended with the kernel handle) or the data transfer operation name.
ExecutionOrder	The order of execution for the kernel and data transfer operations from the program.
GlobalWorkSize	The global work-item size of the kernel.
GroupWorkSize	The work-group size of the kernel.
Time	For a kernel dispatch operation: time spent executing the kernel in milliseconds (does not include the kernel setup time). For a buffer or image object operation: time spent transferring data in milliseconds.

Name	Description
LocalMemSize	The amount of local memory (LDS for GPU) in bytes being used by the kernel.
DataTransferSize	The data transfer size in kilobytes.
GPRs	The number of General Purpose Registers (GPRs) allocated by the kernel.
ScratchRegs	The number of scratch registers used by the kernel. To improve performance (get this number down to zero), reduce the number of GPRs used by the kernel.
FCStacks	The number of flow control stack size used by the kernel (only for GPU device). This number can affect the number of in-flight wavefronts. To reduce the stack size, reduce the amount of flow control nesting in the kernel.
Wavefronts	Total wavefronts.
ALUInsts	The average ALU instructions executed per work-item (affected by flow control).
FetchInsts	The average Fetch instructions from the video memory executed per work-item (affected by flow control).
WriteInsts	The average Write instructions to the video memory executed per work-item (affected by flow control).
ALUBusy	The percentage of GPUTime ALU instructions are processed. Value range: 0% (bad) to 100% (optimal).
ALUFetchRatio	The ratio of ALU to Fetch instructions. If the number of Fetch instruction is zero, then one will be used instead.
ALUPacking	The ALU vector packing efficiency (in percentage). This value indicates how well the Shader Compiler packs the scalar or vector ALU in a kernel to the five-way (or four-way, depending on the GPU type) VLIW instructions. Value range: 0% (bad) to 100% (optimal). Values below 70 percent indicate that ALU dependency chains may be preventing full use of the processor.
FetchSize	The total kilobytes fetched from the video memory. This is measured with all extra fetches and any cache or memory effects taken into account.
CacheHit	The percentage of fetches that hit the data cache. Value range: 0% (no hit) to 100% (optimal).
FetchUnitBusy	The percentage of GPUTime the Fetch unit is active. The result includes the stall time (FetchUnitStalled). This is measured with all extra fetches and any cache or memory effects taken into account. Value range: 0% to 100% (fetch-bound).
FetchUnitStalled	The percentage of GPUTime the Fetch unit is stalled. Try reducing the number of fetches or reducing the amount per fetch if possible. Value range: 0% (optimal) to 100% (bad).
WriteUnitStalled	The percentage of GPUTime Write unit is stalled. Value range: 0% to 100% (bad).
Performance Counters for ATI Radeon™ HD 5000 Series Graphics Cards	
LDSFetchInsts	The average Fetch instructions from the LDS executed per work-item (affected by flow control). This counter is a subset of the ALUInsts counter.
LDSWriteInsts	The average Write instructions to the LDS executed per work-item (affected by flow control). This counter is a subset of the ALUInsts counter.
FastPath	The total kilobytes written to the video memory through the FastPath that support only basic operations: no atomics or sub-32 bit types. This is an optimized path in the hardware.

Name	Description
CompletePath	The total kilobytes written to the video memory through the CompletePath which supports atomics and sub-32 bit types (byte, short). This number includes bytes for load, store, and atomics operations on the buffer. This number may indicate a big performance impact (higher number equals lower performance). If possible, remove the usage of this Path by moving atomics to the local memory or partition the kernel.
PathUtilization	The percentage of bytes written through the FastPath or CompletePath compared to the total number of bytes transferred over the bus. To increase the path utilization, use the FastPath. Value range: 0% (bad) to 100% (optimal).
LDSBankConflict	The percentage of GPUTime LDS is stalled by bank conflicts. Value range: 0% (optimal) to 100% (bad).

4.2 Analyzing Stream Processor Kernels

4.2.1 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 an environment variable, `GPU_DUMP_DEVICE_KERNEL`.

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*.

The instruction set architecture (ISA) defines the instructions and formats accessible to programmers and compilers for the AMD GPUs. The Evergreen-family ISA instructions and microcode are documented in the *AMD Evergreen-Family ISA Instructions and Microcode*. (For a complete description of the R700 ISA, see the *R700-Family Instruction Set Architecture*.)

4.2.2 Generating IL and ISA Code

In Microsoft Visual Studio, the AMD APP Profiler provides an integrated tool to view 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. Currently, the AMD APP Profiler works only with 32-bit applications.

Developers also can generate IL and ISA code from their OpenCL™ kernel by setting the environment variable `GPU_DUMP_DEVICE_KERNEL` to one of the following possible values:

Value	Description
1	Save intermediate IL files in local directory.
2	Disassemble ISA file and save in local directory.
3	Save both the IL and ISA files in local directory.

After setting the flag, run the OpenCL host application. When the host application builds the kernel, the OpenCL compiler saves the `.il` and `.isa` files for each kernel in the local directory.

4.3 Estimating Performance

4.3.1 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

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 al 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.

4.3.2 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
}
```

Any of the normal CPU time-of-day routines can measure 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. 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*.

4.3.3 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 20-200 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:

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

4.4 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.

4.4.1 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 4.2 lists the performance of each memory type given a quad-core AMD Opteron™ 1381 2.5 GHz CPU and an ATI Radeon™ 5870 discrete GPU. The performance was measured using the BufferBandwidth SDK sample.

Table 4.2 Memory Bandwidth in GB/s (R = read, W = write) I

	Direct Access		Runtime Transfer Source				
Direct Access Target/Runtime Transfer Destination	**CPU**	**from GPU**	**Host Memory**	**Pinned Host Memory**	**Device-Visible Host Memory**	**Device Memory**	**Host-Visible Device Memory**
Host Memory	R 7.3 W 7.1	No access	3.8		Not used for OpenCL memory objects on discrete parts.	2.2	
Pinned Host Memory		R 5.5 W 3.7				6.4	
Device-Visible Host Memory	Not used for OpenCL memory objects on discrete parts.						
Device Memory	No access	R 130 W 108	5.0	5.1		45	
Host-Visible Device Memory	R 0.01 W 6.0						

4.4.1.1 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 4.4.1.2). 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.

- <=32KiB: 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.
- >32KiB and <=16MiB: The host memory physical pages containing the data are pinned, the GPU DMA engine is used, and the pages then are unpinned.
- >16MiB: 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.

4.4.1.2 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.

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 4.4.1.3), which does not use the cache coherency protocol.

4.4.1.3 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.

4.4.1.4 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 (except for the host-visible device memory portion described in section 4.4.1.5).

4.4.1.5 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 lower 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.

4.4.2 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 4.3 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.

Currently, multiple GPU devices are not supported.

Table 4.3 OpenCL Memory Object Properties

			clEnqueueMapBuffer/ clEnqueueMapImage/ clEnqueueUnmapMemObject	
clCreateBuffer/ clCreateImage Flags Argument	**Device Type**	**Location**	**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)	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			

clCreateBuffer/ clCreateImage Flags Argument	Device Type	Location	clEnqueueMapBuffer/ clEnqueueMapImage/ clEnqueueUnmapMemObject	
			Map Mode	**Map Location**
`CL_MEM_ALLOC_HOST_PTR` (`clCreateImage` on Windows 7, Vista & 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 (host memory is passed to `host_ptr` argument of `cCreateBuffer` / `clCreateImage` is pinned; it is unpinned when memory object is deleted (unless the only device in context is CPU; then, no pinning is done, and host memory is used)
	Fusion APU	Device-visible host memory		
	CPU	Use *Map Location* directly	Zero copy	
`CL_MEM_USE_PERSISTENT_MEM_AMD` (Windows 7, Vista)	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)	Same as Default.			

4.4.3 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 4.3 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.

4.4.3.1 Zero Copy Memory Objects

`CL_MEM_USE_PERSISTENT_MEM_AMD` is a new AMD extension as of SDK 2.4, and `CL_MEM_ALLOC_HOST_PTR` behaves differently than it did with SDK 2.3. Together, they support zero copy memory objects. The former provides device-resident zero copy memory objects, and the latter provides 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.

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.

Zero copy device resident images are supported. However, zero copy host resident images are not supported.

Currently, Linux does not support any zero copy memory objects.

4.4.3.2 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 4.4 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.

Table 4.4 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	If CL_MEM_USE_HOST_PTR and map location not current, then device to host; otherwise, none	Host to device
CL_MAP_READ CL_MAP_WRITE	Device to host if map location no current	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, the memory should be 4KiB aligned. 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 4.4.1.1, host memory transfers incur considerable cost in pinning/unpinning on every transfer. If used, minimize the pinning cost by ensuring the memory is 4KiB 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.

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 object, 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. When the memory object is allocated on the device on first use, the data is transferred to the device. 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.

4.4.4 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 4.4.1.1, “Host Memory,” page 4-8, 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 4.4.1.3, “Device-Visible Host Memory,” page 4-9. 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.

4.4.5 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.

Currently, the runtime does not exploit the full potential of asynchronous DMA transfers, but coding applications in this manner will automatically gain the benefits once further support is added.

4.5 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.

4.5.1 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 2.0 16x bus and a Cypress architecture (AMD Radeon™ HD 58XX series) graphics card has a nominal interconnect bandwidth of eight GB/s. The peak bandwidth achievable at application level is in the 5 to 6 GB/s range.
- *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 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. Accesses to the WC address space are uncached. While chipset-dependent, executing this path across an interconnect to graphics memory typically provides fast streamed writes, but slow reads. On some platforms, scattered writes of less than cache line size are slow.

4.5.2 Buffers

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

4.5.2.1 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 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.

4.5.2.2 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` buffer is
  - a zero copy buffer that resides on the host.
  - directly accessible by the host at host memory bandwidth.
  - directly accessible by the device across the interconnect.
  - pinned and can be copied to, and from, the device at peak interconnect bandwidth using `clEnqueueCopyBuffer`.
- 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 WC path).
  - copyable to, and from, the device at peak interconnect bandwidth using `clEnqueueCopyBuffer`.

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 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).

4.5.2.3 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, beginning with the one that is easiest and most generic, then moving on to higher-performing and more specialized cases.

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()`, but currently does not achieve peak interconnect bandwidth in the default usage.
- If an application is able to let OpenCL allocate the buffer, options 3) and 4) below can be used to avoid the extra `memcpy()`. In the case of option 5), `memcpy()` and transfer are one and the same thing.

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. It is an ideal choice if the application wants to transfer a buffer that has already been allocated elsewhere (for example through `malloc()` or `mmap()`). The performance is currently less than peak interconnect bandwidth. For example, on a high-performance workstation running Windows 7, it might reach 60% of peak interconnect bandwidth.

Option 2 - `clEnqueueMapBuffer()` and `clEnqueueUnmapMemObject()`

This is the best 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 thorugh `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, .. )`

      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 3 - `clEnqueueCopyBuffer()` on a pre-pinned host buffer (requires zero copy support)

If an application has the choice of letting OpenCL perform the original memory allocation, this is the best choice. This assumes that original application data is held in the CL buffer throughout its lifetime.

If an application already has buffers allocated through `malloc()` or `mmap()`, option 3) performs identically to option 2).

On a platform supporting zero copy, the following command creates a pre-pinned zero copy host buffer:

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

For the purpose of this transfer path, only the pre-pinned attribute of the buffer is used.

The application also creates a regular device buffer:

```
devbuf = clCreateBuffer( .., e.g. CL_MEM_READ_WRITE,   ..)
```

The transfer sequence is as follows:

a. Application modifies or reads the 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.

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

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

b. Transfer to, or from, the device.

After the application has modified pinned buffer, it can transfer the buffer into, or from, another device buffer:

```
clEnqueueCopyBuffer( .., buf, devbuf , ..) or clEnqueueCopyBuffer(
.., devbuf, buf, ..)
```

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 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.

a. 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.

b. 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 WC 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.

   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.

4.6 Global Memory Optimization

Figure 4.1 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.

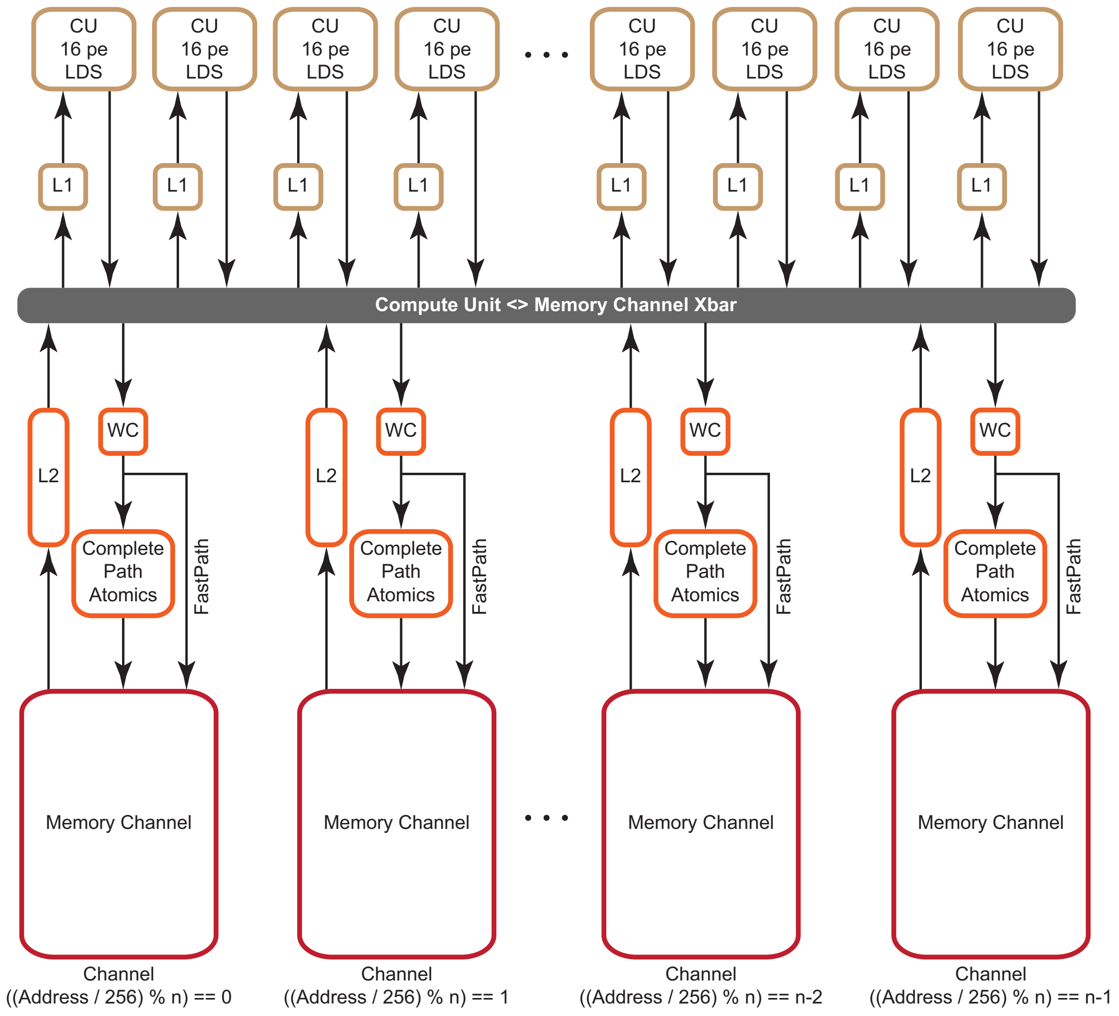
CU
16 pe
LDS
L1
Compute Unit <> Memory Channel Xbar
L2
WC
Complete Path Atomics
FastPath
Memory Channel
Channel
((Address / 256) % n) == 0
((Address / 256) % n) == 1
((Address / 256) % n) == n-2
((Address / 256) % n) == n-1

Figure 4.1 Memory System

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

The ATI Radeon™ HD 5870 GPU has eight memory controllers ("Memory Channel" in Figure 4.1). 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.

4.6.1 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.

4.6.1.1 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 4.2 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 atomic add forces CompletePath.

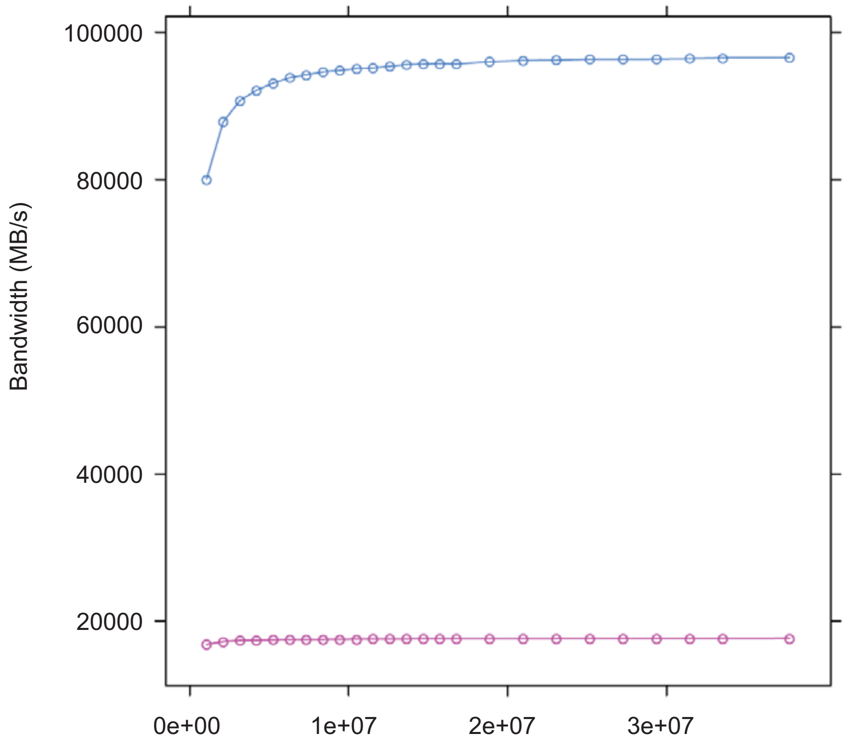

Figure 4.2 FastPath (blue) vs CompletePath (red) Using float1

The kernel code follows. Note that the atomic extension must be enabled under OpenCL 1.0.

```
__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 4.5 lists the effective bandwidth and ratio to maximum bandwidth.

Table 4.5 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.

4.6.1.2 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. 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
```

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).

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.

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.

4.6.2 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 bank.
- 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, 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)`

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 4.6 shows how much the launch dimension can affect performance. It lists each kernel's effective bandwidth and ratio to maximum bandwidth.

Table 4.6 Bandwidths for Different Launch Dimensions

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 - 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.

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.

4.6.2.1 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 4.3 illustrates the transformation to staggered offsets.

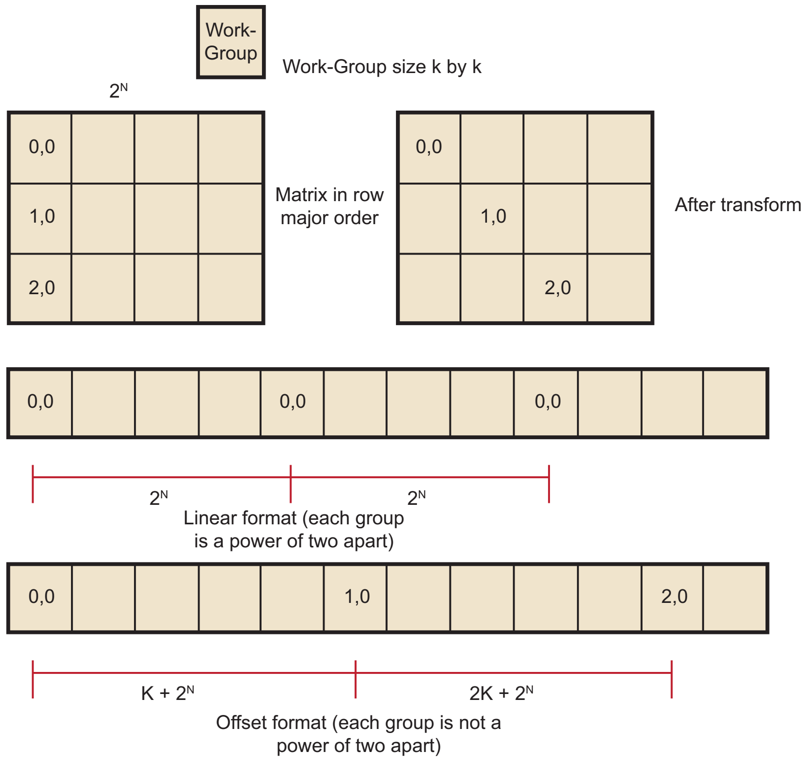
Work-Group
Work-Group size k by k
2^N
0,0
1,0
2,0
Matrix in row major order
0,0
1,0
2,0
After transform
0,0
0,0
0,0
2^N
2^N
Linear format (each group is a power of two apart)
0,0
1,0
2,0
K + 2^N
2K + 2^N
Offset format (each group is not a power of two apart)

Figure 4.3 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.

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);
}
```

4.6.2.2 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
```

4.6.3 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 4.4. Change to float4 after eliminating the conflicts.

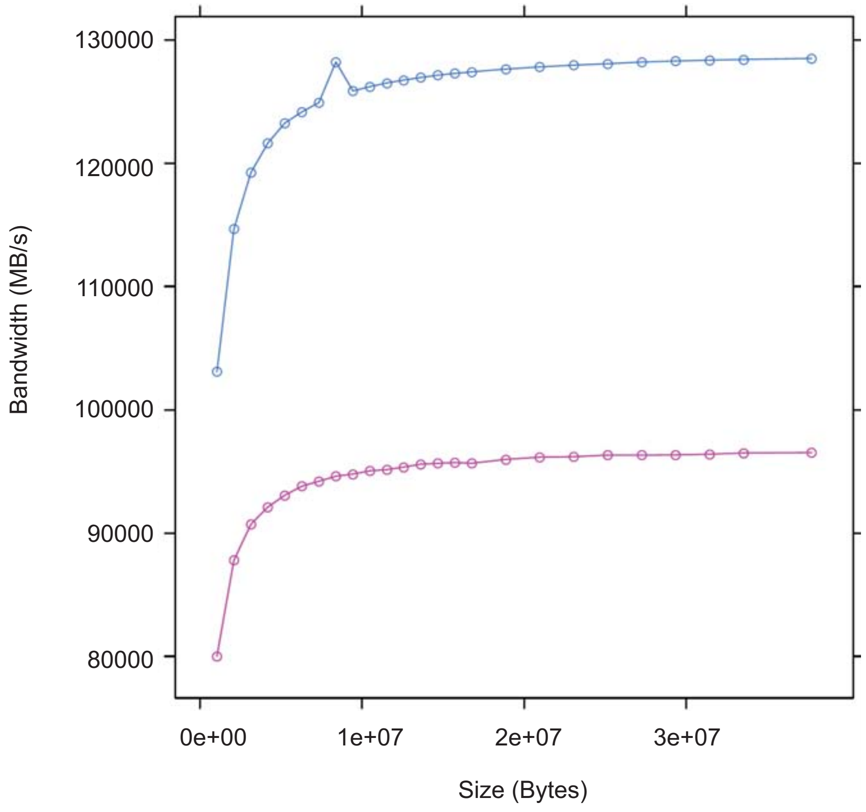

Figure 4.4 Two Kernels: One Using float4 (blue), the Other float1 (red)

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 4.7).

Table 4.7 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%

4.6.4 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 4.5.

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.

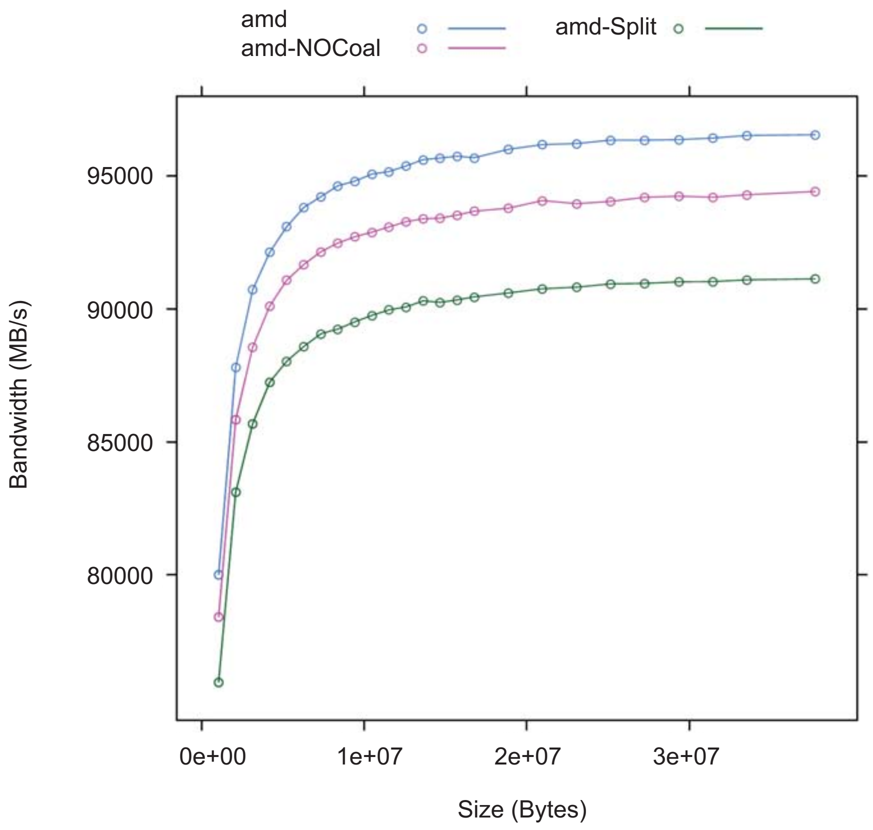

Figure 4.5 Effect of Varying Degrees of Coalescing - Coal (blue), NoCoal (red), Split (green)

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.

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)
{
  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 4.8 lists the effective bandwidth and ratio to maximum bandwidth for each kernel type.

Table 4.8 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.

4.6.5 Alignment

The program in Figure 4.6 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.

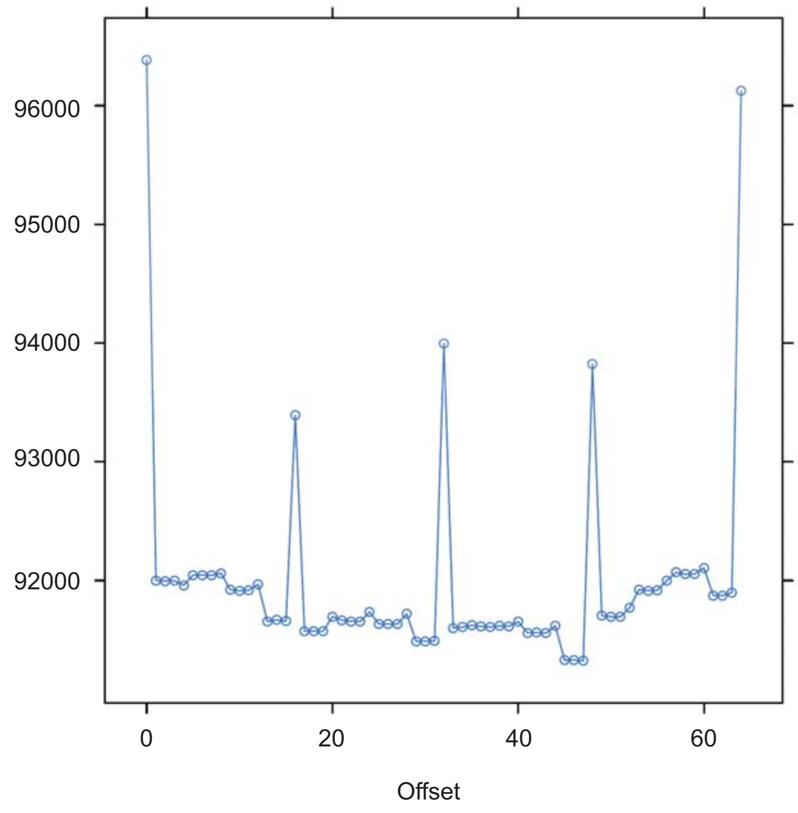

Figure 4.6 Unaligned Access Using float1

```
__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 4.9 lists the effective bandwidth and ratio to maximum bandwidth for each kernel type.

Table 4.9 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%

4.6.6 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.

4.6.7 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.

4.7 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 float2 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:

`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.

4.8 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.

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.

4.9 OpenCL Memory Resources: Capacity and Performance

Table 4.10 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 4.10 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 4.11.2, "Resource Limits on Active Wavefronts," page 4-45, 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 4.6, "Global Memory Optimization," page 4-20.

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`.

Specifying `-fno-alias` as an argument to clBuildProgram implicitly adds the restrict argument to all global pointers in the program.

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.

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 4.10 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.”

4.10 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.

As of Stream SDK 2.2, it is necessary to use OpenCL images in order to get fully generalized L1 cache support. A program that requires the use of OpenCL buffers must transfer buffer content to images, or use LDS instead.

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 SIMD. 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-SIMD 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 SIMD. This, in turn, reduces memory latency hiding. Requesting larger amounts of LDS per work-group results in fewer wavefronts per SIMD, 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).

4.11 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.

4.11.1 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 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.

4.11.2 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.

4.11.2.1 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 4.11 shows how the registers used in the kernel impacts the register-limited wavefronts/compute unit.

Table 4.11 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

GP Registers used by Kernel	Register-Limited Wavefronts / Compute-Unit
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

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.

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 4.2, “Analyzing Stream Processor Kernels,” page 4-3), 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.

4.11.2.2 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 4.11 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.

4.11.2.3 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 4.12 provides more details about how LDS usage can impact the wavefronts/compute unit.

Table 4.12 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 4.2, "Analyzing Stream Processor Kernels," page 4-3), then search for the string `SQ_LDS_ALLOC:SIZE` in the ISA dump. Note that the value is shown in hexadecimal format.

4.11.3 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.

4.11.3.1 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. Current-generation AMD GPUs typically have 2-20 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).

4.11.3.2 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.

4.11.3.3 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 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.

4.11.3.4 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):

WaveFront0	0,0	1,0	2,0	3,0	4,0	5,0	6,0	7,0	8,0	9,0	10,0	11,0	12,0	13,0	14,0	15,0
	16,0	17,0	18,0	19,0	20,0	21,0	22,0	23,0	24,0	25,0	26,0	27,0	28,0	29,0	30,0	31,0
	0,1	1,1	2,1	3,1	4,1	5,1	6,1	7,1	8,1	9,1	10,1	11,1	12,1	13,1	14,1	15,1
	16,1	17,1	18,1	19,1	20,1	21,1	22,1	23,1	24,1	25,1	26,1	27,1	28,1	29,1	30,1	31,1
WaveFront1	0,2	1,2	2,2	3,2	4,2	5,2	6,2	7,2	8,2	9,2	10,2	11,2	12,2	13,2	14,2	15,2
	16,2	17,2	18,2	19,2	20,2	21,2	22,2	23,2	24,2	25,2	26,2	27,2	28,2	29,2	30,2	31,2
	0,3	1,3	2,3	3,3	4,3	5,3	6,3	7,3	8,3	9,3	10,3	11,3	12,3	13,3	14,3	15,3
	16,3	17,3	18,3	19,3	20,3	21,3	22,3	23,3	24,3	25,3	26,3	27,3	28,3	29,3	30,3	31,3

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.

4.11.4 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.

4.11.5 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.

4.12 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.

4.12.1 CPU and GPU Devices

Table 4.13 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 4.13 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 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 SIMD-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.

4.12.2 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).

4.12.3 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 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.

4.12.4 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 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)
        }
    }
```

4.12.5 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 4.11.4, "Optimizing for Cedar," page 4-52, for optimization information specific to this device).

As described in Section 4.14, "Clause Boundaries," page 4-65, CPUs and GPUs have very different performance characteristics, and some of these impact how one writes an optimal kernel. Notable differences include:

- The SIMD 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 SIMD 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 4.13.4, "VLIW and SSE Packing," page 4-63, 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

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 4.12.3, "Partitioning Work for Multiple Devices," page 4-55.

4.12.6 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.

4.13 Instruction Selection Optimizations

4.13.1 Instruction Bandwidths

Table 4.14 lists the throughput of instructions for GPUs.

Table 4.14 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 Cypress 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.

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 4.14 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 4.14, 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.

4.13.2 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.

4.13.3 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 4.15 lists the native speedup factor for certain functions.

Table 4.15 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

4.13.4 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 4.7 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 4.7 Unmodified Loop

Figure 4.8 is the same loop unrolled 4x.

```
__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 4.8 Kernel Unrolled 4X

Figure 4.9 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 4.9 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 4.10 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 4.10 Unrolled Kernel Using float4 for Vectorization

4.14 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
```

4.15 Additional Performance Guidance

This section is a collection of performance tips for GPU compute and AMD-specific optimizations.

4.15.1 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. When `unroll-factor` is not specified, it is equivalent to specifying the static loop count as `unroll-factor`.

4.15.2 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 4.11 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.

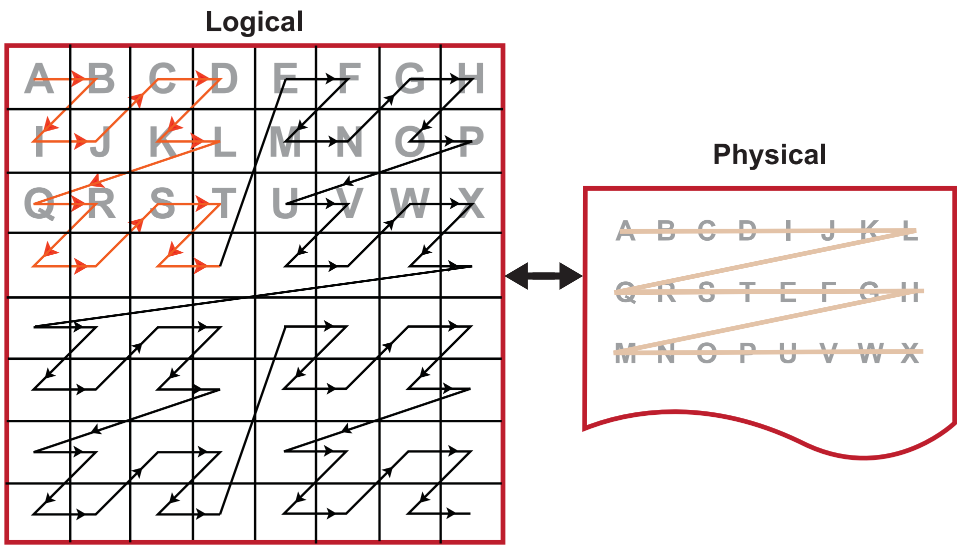

Figure 4.11 One Example of a Tiled Layout Format

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.

4.15.3 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;
  ```

  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.

4.15.4 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.

4.15.5 Guidance for CPU Programmers Using OpenCL

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.

4.15.6 Optimizing Kernels for Evergreen and 6900-Series GPUs

4.15.6.1 Clauses

The architecture for the 6900 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.

4.15.6.2 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;
```

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.

4.15.6.3 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`).

4.15.6.4 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.

4.15.6.5 Avoid Nested `if`s

Because the GPU is a SIMD 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.

4.15.6.6 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.

4.15.6.7 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.

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.2, on page A-13, 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.

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
```

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 make sure the extension is supported.

The `clGetExtensionFunctionAddress()` function cannot be used to get core API function addresses.

A.6 List of Supported Extensions

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.

- `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 following extension:

```
cl_amd_fp64
cl_amd_device_attribute_query
cl_amd_event_callback
cl_amd_media_ops
cl_amd_printf
```

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_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.8.3 `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 does not need to be explicitly enabled through `#pragma` when using the SDK v2 of AMD Accelerated Parallel Processing.

A.8.4 `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)
```

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));
```

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.8.5 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`.

A.9 Supported Functions for `cl_amd_fp64`

Table A.1 lists the functions supported by `cl_amd_fp64` on three platforms.

Table A.1 Functions Supported by `cl_amd_fp64`

X = Supported	* = Beta	**Evergreen**[1]	**RV770**[2]	**x86 CPU**
	Query `clGetDeviceInfo()` with `CL_DEVICE_DOUBLE_FP_CONFIG`	X	*	X
	double conversions	X	*	X
Types	type: double	X	*	X
	type: double2	X	*	X
	type: double3	X	*	X
	type: double4	X	*	X
	type: double8	X	*	X
	type: double16	X	*	X

Operators and Relational Functions	+	X	*	X
	-	X	*	X
	*	X	*	X
	/	X	*[3]	X
	Relational Functions (<, <=, >, >=, !=, ==)	X	*	X
	isequal()	X	*	X
	isnotequal()	X	*	X
	isgreater()	X	*	X
	isgreaterequal()	X	*	X
	isless()	X	*	X
	islessequal()	X	*	X
	islessgreater()	X	*	X
	isfinite()	X	*	X
	isinf()	X	*	X
	isnan()	X	*	X
	isnormal()	X	*	X
	isordered()	X	*	X
	isunordered()	X	*	X
	signbit()	X	*	X
	bitselect()	X	*	X
	select()	X	*	X

X = Supported * = Beta		Evergreen[1]	RV770[2]	x86 CPU
Math Functions	acos()	X	*	X
	acosh()			X
	acospi()			X
	asin()	X	*	X
	asinh()			X
	asinpi()			X
	atan()	X	*	X
	atan2()			X
	atanh()			X
	atanpi()			X
	atan2pi()			X
	cbrt()	X	*	X
	ceil()	X	*	X
	copysign()	X	*	X
	cos()	X	*	X
	cosh()			X
	cospi()	X	*	X
	erf()	X	*	X
	exp()	X	*	X
	exp2()	X	*	X
	exp10()	X	*	X
	expm1()	X	*	X
	fabs()	X	*	X
	fdim()	X	*	X
	floor()	X	*	X
	fma()	X		X
	fmax()	X	*	X
	fmin()	X	*	X
	fmod()			X
	fract()	X	*	X
	frexp()	X	*	X
	hypot()			X
	ilogb()	X	*	X
	ldexp()	X	*	X
	lgamma()			X
	lgamma_r()			X
	log()	X	*	X
	log2()	X	*	X
	log10()	X	*	X
	log1p()			X
	logb()	X	*	X

X = Supported * = Beta		Evergreen[1]	RV770[2]	x86 CPU
Math Functions (cont'd)	mad()	X	*	X
	maxmag()	X	*	X
	minmag()	X	*	X
	mad()	X	*	X
	maxmag()	X	*	X
	minmag()	X	*	X
	modf()	X	*	X
	nan()	X	*	X
	nextafter()	X	*	X
	pow()	X	*	X
	pown()	X	*	X
	powr()	X	*	X
	remainder()			X
	rint()	X	*	X
	rootn()	X	*	X
	round()	X	*	X
	rsqrt()	X		X
	sin()	X	*	X
	sincos()	X	*	X
	sinh()			X
	sinpi()	X	*	X
	sqrt()	X	*	X
	tan()	X	*	X
	tanh()			X
	tanpi()	X	*	X
	trunc()	X	*	X

X = Supported * = Beta		Evergreen[1]	RV770[2]	x86 CPU
Macros	HUGE_VAL	X	*	X
	FP_FAST_FMA	X	*	X
	DBL_DIG	X	*	X
	DBL_MANT_DIG	X	*	X
	DBL_MAX_10_EXP	X	*	X
	DBL_MAX_EXP	X	*	X
	DBL_MIN_10_EXP	X	*	X
	DBL_MIN_EXP	X	*	X
	DBL_MAX	X	*	X
	DBL_MIN	X	*	X
	DBL_EPSILON	X	*	X
	M_E	X	*	X
	M_LOG2E	X	*	X
	M_LOG10E	X	*	X
	M_LN2	X	*	X
	M_LN10	X	*	X
	M_PI	X	*	X
	M_PI_2	X	*	X
	M_PI_4	X	*	X
	M_1_PI	X	*	X
	M_2_PI	X	*	X
	M_2_SQRTPI	X	*	X
	M_SQRT2	X	*	X
	M_SQRT1_2	X	*	X
Common Functions	clamp()	X	*	X
	degrees()	X	*	X
	max()	X	*	X
	min()	X	*	X
	mix()	X	*	X
	radians()	X	*	X
	step()	X	*	X
	smoothstep()	X	*	X
	sign()	X	*	X
Geometric Functions	cross()	X	*	X
	dot()	X	*	X
	distance()	X		X
	length()	X		X
	normalize()	X		X
Vector Data Load and	vload*n*()	X	*	X
	vstore*n*()	X	*	X

X = Supported * = Beta		Evergreen[1]	RV770[2]	x86 CPU
Async Copies and Prefetch	async_work_group_copy	X	*	X
	wait_group_events()	X	*	X
	prefetch()	X	*	X

1. ATI Radeon™ HD 5900 series GPUs, ATI Radeon™ HD 5800 series GPUs, AMD FirePro™ V8800 series GPUs, AMD FirePro™ V7800 series GPUs and AMD FireStream™ 9300 series GPU Compute Accelerators.
2. ATI Radeon™ HD 4800 series GPUs, AMD Mobility Radeon™ HD 4800 series GPUs, AMD FirePro™ V8700 series GPUs, and AMD FireStream™ 9200 series GPUs.
3. Denormals flushed to zero, too large/small produce NaN instead of inf.

A.10 Extension Support by Device

Table A.2 and Table A.3 list the extension support for selected devices.

Table A.2 Extension Support for Devices 1

	AMD GPU					
Extension	**Cayman[1]**	**Barts[2]**	**Cypress[3]**	**Juniper[4]**	**Redwood[5]**	**Cedar[6]**
cl_khr_*_atomics	Yes	Yes	Yes	Yes	Yes	Yes
cl_khr_gl_sharing	Yes	Yes	Yes	Yes	Yes	Yes
cl_khr_byte_addressable_store	Yes	Yes	Yes	Yes	Yes	Yes
cl_ext_device_fission	No	No	No	No	No	No
cl_amd_device_attribute_query	Yes	Yes	Yes	Yes	Yes	Yes
cl_amd_fp64	Yes	No	Yes	No	No	No
Images	Yes	Yes	Yes	Yes	Yes	Yes
cl_khr_d3d10_sharing	Yes	Yes	Yes	Yes	Yes	Yes
cl_amd_media_ops	Yes	Yes	Yes	Yes	Yes	Yes
cl_amd_printf	Yes	Yes	Yes	Yes	Yes	Yes
cl_amd_popcnt	Yes	Yes	Yes	Yes	Yes	Yes
cl_khr_3d_image_writes	Yes	Yes	Yes	Yes	Yes	Yes
Platform Extensions						
cl_khr_icd	Yes	Yes	Yes	Yes	Yes	Yes
cl_amd_event_callback	Yes	Yes	Yes	Yes	Yes	Yes
cl_amd_offline_devices	Yes	Yes	Yes	Yes	Yes	Yes

1. AMD Radeon™ HD 6900 series.
2. AMD Radeon™ HD 6800 series.
3. ATI Radeon™ HD 5900 series and 5800 series, AMD FirePro™ V8800 series and V8700 series.
4. ATI Radeon™ HD 5700 series, AMD Mobility Radeon™ HD 5800 series, AMD FirePro™ V5800 series, AMD Mobility FirePro™ M7820.
5. 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
6. ATI Radeon™ HD 5400 Series, AMD Mobility Radeon™ HD 5400 Series

Table A.3 Extension Support for Devices 2 and CPUs

	AMD GPU		x86 CPU
Extension	**RV770** [1]	**RV730/RV710**[2]	**with SSE2 or later**
cl_khr_*_atomics	No	No	Yes
cl_khr_gl_sharing	Yes	Yes	Yes
cl_khr_byte_addressable_store	No	No	Yes
cl_ext_device_fission	No	No	Yes
cl_amd_device_attribute_query	Yes	Yes	Yes
cl_amd_fp64	Yes	No	Yes
Images	No	No	Yes[3]
cl_khr_d3d10_sharing	Yes	Yes	Yes
cl_amd_media_ops	No	No	Yes
cl_amd_printf	Yes	Yes	Yes
cl_amd_popcnt	No	No	Yes
cl_khr_3d_image_writes	No	No	No
Platform Extensions			
cl_khr_icd	Yes	Yes	Yes
cl_amd_event_callback	Yes	Yes	Yes
cl_amd_offline_devices	Yes	Yes	Yes

1. ATI Radeon™ HD 4800 Series, ATI Radeon™ HD 4700 Series, AMD Mobility Radeon™ HD 4800 Series, AMD FirePro™ V8700 Series, AMD Mobility FirePro™ M7740, AMD FireStream™ 9200 Series Compute Accelerator
2. ATI Radeon™ HD 4600 Series, ATI Radeon™ HD 4500 Series, ATI Radeon™ HD 4300 Series, AMD Mobility Radeon™ HD 4600 Series, AMD Mobility Radeon™ HD 4500 Series, AMD Mobility Radeon™ HD 4300 Series, AMD FirePro™ V7700 Series, AMD FirePro™ V5700 Series, AMD FirePro™ V3700 Series, ATI Radeon™ Embedded E4690 Discrete
3. Environment variable `CPU_IMAGE_SUPPORT` must be set.

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.
     */
```

```
    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);
```

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.

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.

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.

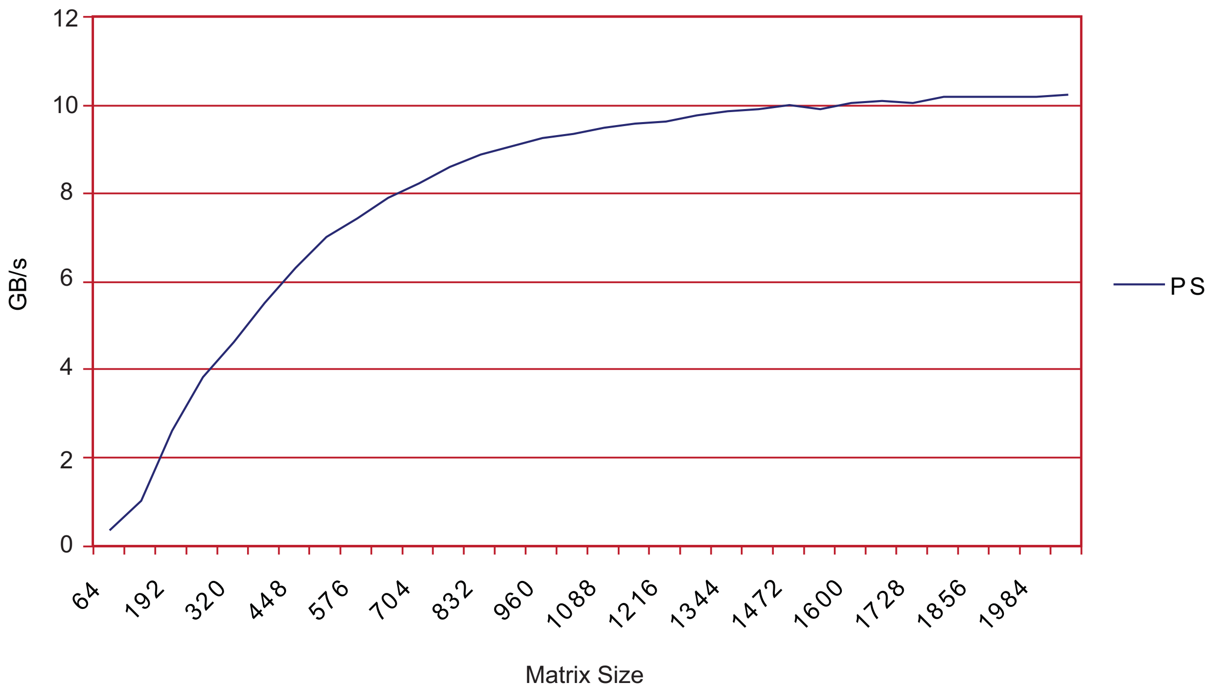

Figure C.1 Pixel Shader Matrix Transpose

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.

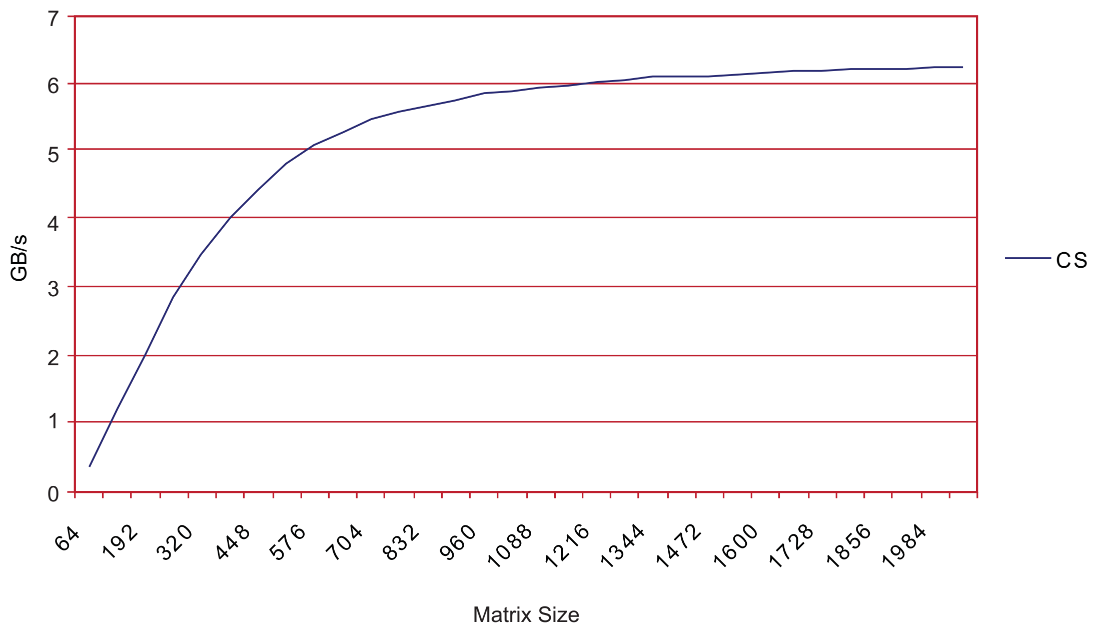

Figure C.2 Compute Kernel Matrix Transpose

C.6 LDS Matrix Transpose

Figure C.3 shows the performance results using the LDS for this matrix transpose.

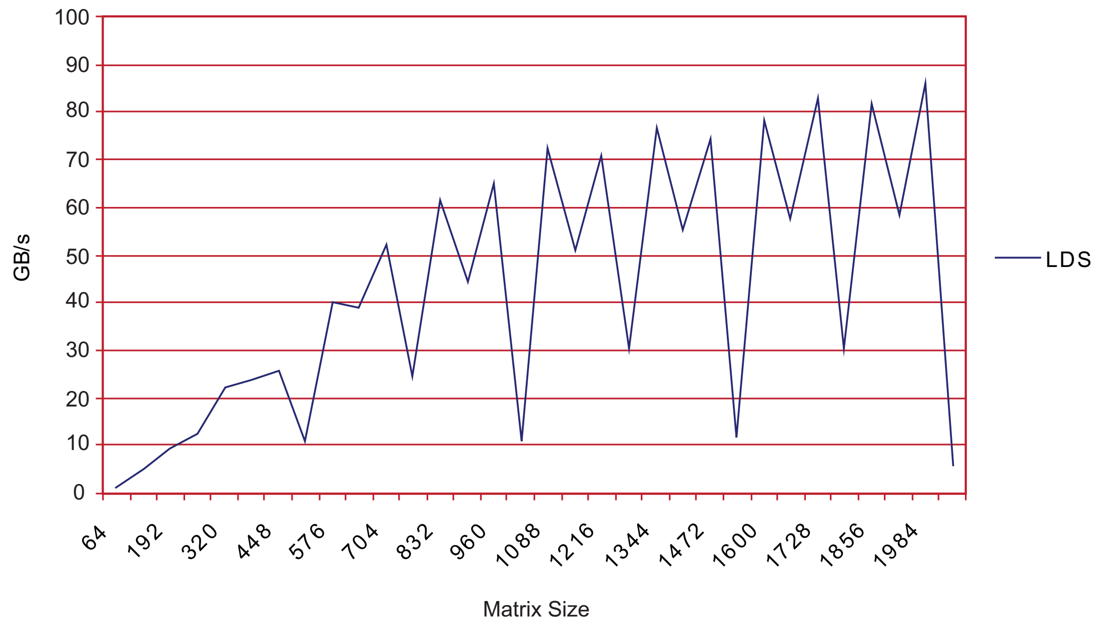

Figure C.3 LDS Matrix Transpose

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.

Appendix D
Device Parameters

On the following pages, Table D.1 through Table D.5 provide device-specific information for AMD GPUs.

Table D.1 Parameters for 68xx and 69xx Devices

	Barts PRO	Barts XT	Blackcomb PRO	Cayman PRO	Blackcomb XT	Cayman XT
Product Name (AMD Radeon™ HD)	6850	6870	6950M	6950	6970M	6970
Engine Speed (MHz)	775	900	580	800	680	880
Compute Resources						
Compute Units	12	14	12	22	12	24
Stream Cores	192	224	192	352	192	384
Processing Elements	960	1120	960	1408	960	1536
Peak Gflops	1488	2016	1113.6	2252.8	1305.6	2703.36
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	32 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	6267	16220
LDS Read (GB/s)	1190	1613	891	2253	1044	2703
Constant Cache Read (GB/s)	2381	3226	1782	4506	2089	5407
L1 Read (GB/s)	595	806	445	1126	522	1352
L2 Read (GB/s)	397	461	297	410	348	451
Global Memory (GB/s)	128	134	115	160	115	176
Global Limits						
Max Wavefronts / GPU	496	496	496	512	496	512
Max Wavefronts / CU (avg)	41.3	35.4	41.3	23.3	41.3	21.3
Max Work-Items / GPU	31744	31744	31744	32768	31744	32768
Memory						
Memory Channels	8	8	8	8	8	8
Memory Bus Width (bits)	256	256	256	256	256	256
Memory Type and Speed (MHz)	GDDR5 1000	GDDR5 1050	GDDR5 900	GDDR5 1250	GDDR5 900	GDDR5 1375
Frame Buffer	1 GB	1 GB	1 GB	1 GB or 2 GB	1 GB or 2 GB	2 GB

Table D.2 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

Table D.3 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

Table D.4 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	32k	32k	32k	32k	32k	32k	32k
LDS Banks / CU	16	32	32	32	32	32	32
Constant Cache / GPU	16k	24k	24k	32k	40k	48k	96k
Max Constants / CU	8k	8k	8k	8k	8k	8k	8k
L1 Cache Size / CU	8k	8k	8k	8k	8k	8k	8k
L2 Cache Size / GPU	128k	256k	256k	512k	512k	512k	2 x 512k
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 GB

Table D.5 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

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-2.

The LLVM IR enables recompilation from LLVM IR to the target. When a binary is used to run on a device for which it was not generated, OpenCL recompiles the LLVM IR to generate a new code for the device. Note that the LLVM IR is only universal within 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 the GPU works only for a GPU device; the LLVM IR for the CPU works only for a CPU device.

BIF2.0 is supported since Stream SDK 2.2.

E.1.1 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.1.2 Recompilation

OpenCL allows building a program with either a source or a binary. If using a binary, OpenCL checks if the BIF binary is good for the target device. If so, it uses the executable in the binary; if not, it tries to recompile from the binary if the binary has the LLVM IR in the `.llvmir` section that was generated before for the same device type (either CPU or GPU). For example, if the binary that was generated for an RV770 device before (device type = GPU) has the LLVM IR, and an application wants to use this binary to build for another GPU device, such as an AMD Radeon™ HD 58XX devices, OpenCL recompiles the LLVM IR in the binary to generate a new binary for devices in this series. But if the LLVM IR was generated for a CPU device, then building for another GPU device fails because CPU and GPU are different device types.

The OpenCL source in a binary is for reference only, and OpenCL does not attempt to recompile the source if the source is present in a binary. However, OpenCL does try to recompile from AMD IL (GPU only) for the device for which the binary was generated. This recompilation from AMD IL is a tentative feature, which is subject to change.

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.

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.

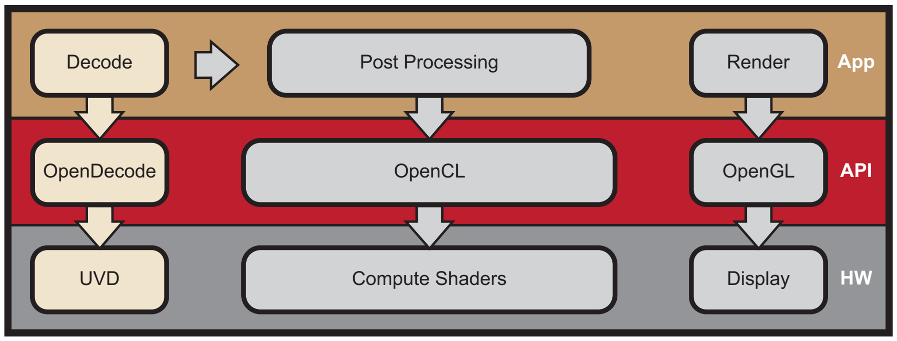
Decode
Post Processing
Render
App
OpenDecode
OpenCL
OpenGL
API
UVD
Compute Shaders
Display
HW
Raw YUV Video after decode
YUV data in OpenCL buffers are used by Compute Shaders without need for copying
Post-processed YUV surface is used as a texture for display

Figure F.1 Open Decode with Optional Post-Processing

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 1-20 for `clCreateContext` usage.)

```
intptr_t properties[] =
    {
     CL_CONTEXT_PLATFORM, (cl_context_properties)platform,
            0
};

ovdContext = clCreateContext(properties,
                             1,
                             &clDeviceID,
                             0,
                             0,
                             &err);
```

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);

    // 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 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);
```

Glossary of Terms

Term	**Description**
*	Any number of alphanumeric characters in the name of a microcode format, microcode parameter, or instruction.
< >	Angle brackets denote streams.
[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).
{BUF, SWIZ}	One of the multiple options listed. In this case, the string *BUF* or the string *SWIZ*.
{x | y}	One of the multiple options listed. In this case, x or y.
0.0	A single-precision (32-bit) floating-point value.
0x	Indicates that the following is a hexadecimal number.
1011b	A binary value, in this example a 4-bit value.
29'b0	29 bits with the value 0.
7:4	A bit range, from bit 7 to 4, inclusive. The high-order bit is shown first.
ABI	Application Binary Interface.
absolute	A displacement that references the base of a code segment, rather than an instruction pointer. See relative.
active mask	A 1-bit-per-pixel mask that controls which pixels in a "quad" are really running. Some pixels might not be running if the current "primitive" does not cover the whole quad. A mask can be updated with a `PRED_SET*` ALU instruction, but updates do not take effect until the end of the ALU clause.
address stack	A stack that contains only addresses (no other state). Used for flow control. Popping the address stack overrides the instruction address field of a flow control instruction. The address stack is only modified if the flow control instruction decides to jump.
ACML	AMD Core Math Library. Includes implementations of the full BLAS and LAPACK routines, FFT, Math transcendental and Random Number Generator routines, stream processing backend for load balancing of computations between the CPU and GPU compute device.
aL (also AL)	Loop register. A three-component vector (x, y and z) used to count iterations of a loop.
allocate	To reserve storage space for data in an output buffer ("scratch buffer," "ring buffer," "stream buffer," or "reduction buffer") or for data in an input buffer ("scratch buffer" or "ring buffer") before exporting (writing) or importing (reading) data or addresses to, or from that buffer. Space is allocated only for data, not for addresses. After allocating space in a buffer, an "export" operation can be done.

Term	**Description**
ALU	Arithmetic Logic Unit. Responsible for arithmetic operations like addition, subtraction, multiplication, division, and bit manipulation on integer and floating point values. In stream computing, these are known as *stream cores*. ALU.[X,Y,Z,W] - an ALU that can perform four vector operations in which the four operands (integers or single-precision floating point values) do not have to be related. It performs “SIMD” operations. Thus, although the four operands need not be related, all four operations execute the same instruction. ALU.Trans - (not relevant on HD 6900 and later devices) An ALU unit that can perform one ALU.Trans (transcendental, scalar) operation, or advanced integer operation, on one integer or single-precision floating-point value, and replicate the result. A single instruction can co-issue four ALU.Trans operations to an ALU.[X,Y,Z,W] unit and one (possibly complex) operation to an ALU.Trans unit, which can then replicate its result across all four component being operated on in the associated ALU.[X,Y,Z,W] unit.
AMD APP KernelAnalyzer	A performance profiling tool for developing, debugging, and profiling stream kernels using high-level stream computing languages.
AR	Address register.
ATI Stream™ SDK	A complete software development suite for developing applications for AMD Accelerated Parallel Processing compute devices. Currently, the ATI Stream SDK includes OpenCL and CAL.
aTid	Absolute work-item ID (formerly *thread ID*). It is the ordinal count of all work-items being executed (in a draw call).
b	A bit, as in *1Mb* for one megabit, or *lsb* for least-significant bit.
B	A byte, as in *1MB* for one megabyte, or *LSB* for least-significant byte.
BLAS	Basic Linear Algebra Subroutines.
border color	Four 32-bit floating-point numbers (XYZW) specifying the border color.
branch granularity	The number of work-items executed during a branch. For AMD GPUs, branch granularity is equal to wavefront granularity.
burst mode	The limited write combining ability. See write combining.
byte	Eight bits.
cache	A read-only or write-only on-chip or off-chip storage space.
CAL	Compute Abstraction Layer. A device-driver library that provides a forward-compatible interface to AMD Accelerated Parallel Processing compute devices. This lower-level API gives users direct control over the hardware: they can directly open devices, allocate memory resources, transfer data and initiate kernel execution. CAL also provides a JIT compiler for AMD IL.
CF	Control Flow.
cfile	Constant file or constant register.
channel	A component in a vector.
clamp	To hold within a stated range.
clause	A group of instructions that are of the same type (all stream core, all fetch, etc.) executed as a group. A clause is part of a CAL program written using the compute device ISA. Executed without pre-emption.

Term	**Description**
clause size	The total number of slots required for an stream core clause.
clause temporaries	Temporary values stored at GPR that do not need to be preserved past the end of a clause.
clear	To write a bit-value of 0. Compare “set”.
command	A value written by the host processor directly to the GPU compute device. The commands contain information that is not typically part of an application program, such as setting configuration registers, specifying the data domain on which to operate, and initiating the start of data processing.
command processor	A logic block in the R700 (HD4000-family of devices) that receives host commands, interprets them, and performs the operations they indicate.
component	(1) A 32-bit piece of data in a “vector”. (2) A 32-bit piece of data in an array. (3) One of four data items in a 4-component register.
compute device	A parallel processor capable of executing multiple work-items of a kernel in order to process streams of data.
compute kernel	Similar to a pixel shader, but exposes data sharing and synchronization.
compute shader	Similar to a pixel shader, but exposes data sharing and synchronization.
compute unit pipeline	A hardware block consisting of five stream cores, one stream core instruction decoder and issuer, one stream core constant fetcher, and support logic. All parts of a compute unit pipeline receive the same instruction and operate on different data elements. Also known as “slice.”
constant buffer	Off-chip memory that contains constants. A constant buffer can hold up to 1024 four-component vectors. There are fifteen constant buffers, referenced as cb0 to cb14. An immediate constant buffer is similar to a constant buffer. However, an immediate constant buffer is defined within a kernel using special instructions. There are fifteen immediate constant buffers, referenced as icb0 to icb14.
constant cache	A constant cache is a hardware object (off-chip memory) used to hold data that remains unchanged for the duration of a kernel (constants). “Constant cache” is a general term used to describe constant registers, constant buffers or immediate constant buffers.
constant file	Same as constant register.
constant index register	Same as “AR” register.
constant registers	On-chip registers that contain constants. The registers are organized as four 32-bit component of a vector. There are 256 such registers, each one 128-bits wide.
constant waterfalling	Relative addressing of a constant file. See waterfalling.
context	A representation of the state of a CAL device.
core clock	See engine clock. The clock at which the GPU compute device stream core runs.
CPU	Central Processing Unit. Also called host. Responsible for executing the operating system and the main part of the application. The CPU provides data and instructions to the GPU compute device.
CRs	Constant registers. There are 512 CRs, each one 128 bits wide, organized as four 32-bit values.

Term	**Description**
CS	Compute shader; commonly referred to as a compute kernel. A shader type, analogous to VS/PS/GS/ES.
CTM	Close-to-Metal. A thin, HW/SW interface layer. This was the predecessor of the AMD CAL.
DC	Data Copy Shader.
device	A *device* is an entire AMD Accelerated Parallel Processing compute device.
DMA	Direct-memory access. Also called DMA engine. Responsible for independently transferring data to, and from, the GPU compute device's local memory. This allows other computations to occur in parallel, increasing overall system performance.
double word	Dword. Two words, or four bytes, or 32 bits.
double quad word	Eight words, or 16 bytes, or 128 bits. Also called "octword."
domain of execution	A specified rectangular region of the output buffer to which work-items are mapped.
DPP	Data-Parallel Processor.
dst.X	The X "slot" of an destination operand.
dword	Double word. Two words, or four bytes, or 32 bits.
element	A component in a vector.
engine clock	The clock driving the stream core and memory fetch units on the GPU compute device.
enum(7)	A seven-bit field that specifies an enumerated set of decimal values (in this case, a set of up to 27 values). The valid values can begin at a value greater than, or equal to, zero; and the number of valid values can be less than, or equal to, the maximum supported by the field.
event	A token sent through a pipeline that can be used to enforce synchronization, flush caches, and report status back to the host application.
export	To write data from GPRs to an output buffer (scratch, ring, stream, frame or global buffer, or to a register), or to read data from an input buffer (a "scratch buffer" or "ring buffer") to GPRs. The term "export" is a partial misnomer because it performs both input and output functions. Prior to exporting, an allocation operation must be performed to reserve space in the associated buffer.
FC	Flow control.
FFT	Fast Fourier Transform.
flag	A bit that is modified by a CF or stream core operation and that can affect subsequent operations.
FLOP	Floating Point Operation.
flush	To writeback and invalidate cache data.
FMA	Fused multiply add.
frame	A single two-dimensional screenful of data, or the storage space required for it.
frame buffer	Off-chip memory that stores a frame. Sometimes refers to the all of the GPU memory (excluding local memory and caches).

Term	**Description**
FS	Fetch subroutine. A global program for fetching vertex data. It can be called by a “vertex shader” (VS), and it runs in the same work-item context as the vertex program, and thus is treated for execution purposes as part of the vertex program. The FS provides driver independence between the process of fetching data required by a VS, and the VS itself. This includes having a semantic connection between the outputs of the fetch process and the inputs of the VS.
function	A subprogram called by the main program or another function within an AMD IL stream. Functions are delineated by `FUNC` and `ENDFUNC`.
gather	Reading from arbitrary memory locations by a work-item.
gather stream	Input streams are treated as a memory array, and data elements are addressed directly.
global buffer	GPU memory space containing the arbitrary address locations to which uncached kernel outputs are written. Can be read either cached or uncached. When read in uncached mode, it is known as mem-import. Allows applications the flexibility to read from and write to arbitrary locations in input buffers and output buffers, respectively.
global memory	Memory for reads/writes between work-items. On HD Radeon 5XXX series devices and later, atomic operations can be used to synchronize memory operations.
GPGPU	General-purpose compute device. A GPU compute device that performs general-purpose calculations.
GPR	General-purpose register. GPRs hold vectors of either four 32-bit IEEE floating-point, or four 8-, 16-, or 32-bit signed or unsigned integer or two 64-bit IEEE double precision data components (values). These registers can be indexed, and consist of an on-chip part and an off-chip part, called the “scratch buffer,” in memory.
GPU	Graphics Processing Unit. An integrated circuit that renders and displays graphical images on a monitor. Also called Graphics Hardware, Compute Device, and Data Parallel Processor.
GPU engine clock frequency	Also called 3D engine speed.
GPU compute device	A parallel processor capable of executing multiple work-items of a kernel in order to process streams of data.
GS	Geometry Shader.
HAL	Hardware Abstraction Layer.
host	Also called CPU.
iff	If and only if.
IL	Intermediate Language. In this manual, the AMD version: AMD IL. A pseudo-assembly language that can be used to describe kernels for GPU compute devices. AMD IL is designed for efficient generalization of GPU compute device instructions so that programs can run on a variety of platforms without having to be rewritten for each platform.
in flight	A work-item currently being processed.
instruction	A computing function specified by the *code* field of an IL_OpCode token. Compare “opcode”, “operation”, and “instruction packet”.
instruction packet	A group of tokens starting with an IL_OpCode token that represent a single AMD IL instruction.

Term	**Description**
int(2)	A 2-bit field that specifies an integer value.
ISA	Instruction Set Architecture. The complete specification of the interface between computer programs and the underlying computer hardware.
kcache	A memory area containing “waterfall” (off-chip) constants. The cache lines of these constants can be locked. The “constant registers” are the 256 on-chip constants.
kernel	A user-developed program that is run repeatedly on a stream of data. A parallel function that operates on every element of input streams. A device program is one type of kernel. Unless otherwise specified, an AMD Accelerated Parallel Processing compute device program is a kernel composed of a main program and zero or more functions. Also called Shader Program. This is not to be confused with an OS kernel, which controls hardware.
LAPACK	Linear Algebra Package.
LDS	Local Data Share. Part of local memory. These are read/write registers that support sharing between all work-items in a work-group. Synchronization is required.
LERP	Linear Interpolation.
local memory fetch units	Dedicated hardware that a) processes fetch instructions, b) requests data from the memory controller, and c) loads registers with data returned from the cache. They are run at stream core or engine clock speeds. Formerly called texture units.
LOD	Level Of Detail.
loop index	A register initialized by software and incremented by hardware on each iteration of a loop.
lsb	Least-significant bit.
LSB	Least-significant byte.
MAD	Multiply-Add. A fused instruction that both multiplies and adds.
mask	(1) To prevent from being seen or acted upon. (2) A field of bits used for a control purpose.
MBZ	Must be zero.
mem-export	An AMD IL term random writes to the global buffer.
mem-import	Uncached reads from the global buffer.
memory clock	The clock driving the memory chips on the GPU compute device.
microcode format	An encoding format whose fields specify instructions and associated parameters. Microcode formats are used in sets of two or four. For example, the two mnemonics, `CF_WORD[0,1]` indicate a microcode-format pair, `CF_WORD0` and `CF_WORD1`.
MIMD	Multiple Instruction Multiple Data. – Multiple SIMD units operating in parallel (Multi-Processor System) – Distributed or shared memory
MRT	Multiple Render Target. One of multiple areas of local GPU compute device memory, such as a “frame buffer”, to which a graphics pipeline writes data.
MSAA	Multi-Sample Anti-Aliasing.
msb	Most-significant bit.

Term	**Description**
MSB	Most-significant byte.
neighborhood	A group of four work-items in the same wavefront that have consecutive work-item IDs (Tid). The first Tid must be a multiple of four. For example, work-items with Tid = 0, 1, 2, and 3 form a neighborhood, as do work-items with Tid = 12, 13, 14, and 15.
normalized	A numeric value in the range [a, b] that has been converted to a range of 0.0 to 1.0 using the formula: normalized value = value/ (b–a+ 1)
oct word	Eight words, or 16 bytes, or 128 bits. Same as “double quad word”. Also referred to as octa word.
opcode	The numeric value of the *code* field of an “instruction”. For example, the opcode for the CMOV instruction is decimal 16 (0x10).
opcode token	A 32-bit value that describes the operation of an instruction.
operation	The function performed by an “instruction”.
PaC	Parameter Cache.
PCI Express	A high-speed computer expansion card interface used by modern graphics cards, GPU compute devices and other peripherals needing high data transfer rates. Unlike previous expansion interfaces, PCI Express is structured around point-to-point links. Also called PCIe.
PoC	Position Cache.
pop	Write “stack” entries to their associated hardware-maintained control-flow state. The `POP_COUNT` field of the `CF_WORD1` microcode format specifies the number of stack entries to pop for instructions that pop the stack. Compare “push.”
pre-emption	The act of temporarily interrupting a task being carried out on a computer system, without requiring its cooperation, with the intention of resuming the task at a later time.
processor	Unless otherwise stated, the AMD Accelerated Parallel Processing compute device.
program	Unless otherwise specified, a program is a set of instructions that can run on the AMD Accelerated Parallel Processing compute device. A device program is a type of kernel.
PS	Pixel Shader, aka pixel kernel.
push	Read hardware-maintained control-flow state and write their contents onto the stack. Compare pop.
PV	Previous vector register. It contains the previous four-component vector result from a ALU.[X,Y,Z,W] unit within a given clause.
quad	For a compute kernel, this consists of four consecutive work-items. For pixel and other shaders, this is a group of 2x2 work-items in the NDRange. Always processed together.
rasterization	The process of mapping work-items from the domain of execution to the SIMD engine. This term is a carryover from graphics, where it refers to the process of turning geometry, such as triangles, into pixels.
rasterization order	The order of the work-item mapping generated by rasterization.
RAT	Random Access Target. Same as UAV. Allows, on DX11 hardware, writes to, and reads from, any arbitrary location in a buffer.
RB	Ring Buffer.

Term	**Description**
register	For a GPU, this is a 128-bit address mapped memory space consisting of four 32-bit components.
relative	Referencing with a displacement (also called offset) from an index register or the loop index, rather than from the base address of a program (the first control flow [CF] instruction).
render backend unit	The hardware units in a processing element responsible for writing the results of a kernel to output streams by writing the results to an output cache and transferring the cache data to memory.
resource	A block of memory used for input to, or output from, a kernel.
ring buffer	An on-chip buffer that indexes itself automatically in a circle.
Rsvd	Reserved.
sampler	A structure that contains information necessary to access data in a resource. Also called Fetch Unit.
SC	Shader Compiler.
scalar	A single data component, unlike a vector which contains a set of two or more data elements.
scatter	Writes (by uncached memory) to arbitrary locations.
scatter write	Kernel outputs to arbitrary address locations. Must be uncached. Must be made to a memory space known as the global buffer.
scratch buffer	A variable-sized space in off-chip-memory that stores some of the “GPRs”.
set	To write a bit-value of 1. Compare “clear”.
shader processor	Pre-OpenCL term that is now deprecated. Also called thread processor.
shader program	User developed program. Also called kernel.
SIMD	Pre-OpenCL term that is now deprecated. Single instruction multiple data unit. – Each SIMD receives independent stream core instructions. – Each SIMD applies the instructions to multiple data elements. Now called a compute unit.
SIMD Engine	Pre-OpenCL term that is now deprecated. A collection of thread processors, each of which executes the same instruction each cycle.
SIMD pipeline	In OpenCL terminology: compute unit pipeline. Pre-OpenCL term that is now deprecated. A hardware block consisting of five stream cores, one stream core instruction decoder and issuer, one stream core constant fetcher, and support logic. All parts of a SIMD pipeline receive the same instruction and operate on different data elements. Also known as “slice.”
Simultaneous Instruction Issue	Input, output, fetch, stream core, and control flow per SIMD engine.
slot	A position, in an “instruction group,” for an “instruction” or an associated literal constant. An ALU instruction group consists of one to seven slots, each 64 bits wide. All ALU instructions occupy one slot, except double-precision floating-point instructions, which occupy either two or four slots. The size of an ALU clause is the total number of slots required for the clause.
SPU	Shader processing unit.

Term	**Description**
SR	Globally shared registers. These are read/write registers that support sharing between all wavefronts in a SIMD (not a work-group). The sharing is column sharing, so work-items with the same work-item ID within the wavefront can share data. All operations on SR are atomic.
src0, src1, etc.	In floating-point operation syntax, a 32-bit source operand. Src0_64 is a 64-bit source operand.
stage	A sampler and resource pair.
stream	A collection of data elements of the same type that can be operated on in parallel.
stream buffer	A variable-sized space in off-chip memory that stores an instruction stream. It is an output-only buffer, configured by the host processor. It does not store inputs from off-chip memory to the processor.
stream core	The fundamental, programmable computational units, responsible for performing integer, single, precision floating point, double precision floating point, and transcendental operations. They execute VLIW instructions for a particular work-item. Each processing element handles a single instruction within the VLIW instruction.
stream operator	A node that can restructure data.
swizzling	To copy or move any component in a source vector to any element-position in a destination vector. Accessing elements in any combination.
thread	Pre-OpenCL term that is now deprecated. One invocation of a kernel corresponding to a single element in the domain of execution. An instance of execution of a shader program on an ALU. Each thread has its own data; multiple threads can share a single program counter. Generally, in OpenCL terms, there is a one-to-one mapping of work-items to threads.
thread group	Pre-OpenCL term that is now deprecated. It contains one or more thread blocks. Threads in the same thread-group but different thread-blocks might communicate to each through global per-SIMD shared memory. This is a concept mainly for global data share (GDS). A thread group can contain one or more wavefronts, the last of which can be a partial wavefront. All wavefronts in a thread group can run on only one SIMD engine; however, multiple thread groups can share a SIMD engine, if there are enough resources. Generally, in OpenCL terms, there is a one-to-one mapping of work-groups to thread groups.
thread processor	Pre-OpenCL term that is now deprecated. The hardware units in a SIMD engine responsible for executing the threads of a kernel. It executes the same instruction per cycle. Each thread processor contains multiple stream cores. Also called shader processor.
thread-block	Pre-OpenCL term that is now deprecated. A group of threads which might communicate to each other through local per SIMD shared memory. It can contain one or more wavefronts (the last wavefront can be a partial wavefront). A thread-block (all its wavefronts) can only run on one SIMD engine. However, multiple thread blocks can share a SIMD engine, if there are enough resources to fit them in.
Tid	Work-item id (previously called a *thread id*) within a thread block. An integer number from 0 to Num_threads_per_block-1
token	A 32-bit value that represents an independent part of a stream or instruction.
UAV	Unordered Access View. Same as random access target (RAT). They 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).

Term	**Description**
uncached read/write unit	The hardware units in a GPU compute device responsible for handling uncached read or write requests from local memory on the GPU compute device.
vector	(1) A set of up to four related values of the same data type, each of which is an element. For example, a vector with four elements is known as a “4-vector” and a vector with three elements is known as a “3-vector”. (2) See “AR”. (3) See ALU.[X,Y,Z,W].
VLIW design	Very Long Instruction Word. – Co-issued up to 6 operations (5 stream cores + 1 FC); where FC = flow control. – 1.25 Machine Scalar operation per clock for each of 64 data elements – Independent scalar source and destination addressing
vTid	Work-item ID (formerly *thread ID*) within a work-group.
waterfall	To use the address register (AR) for indexing the GPRs. Waterfall behavior is determined by a “configuration registers.”
wavefront	Group of work-items executed together on a single SIMD engine. Composed of quads. A full wavefront contains 64 work-items; a wavefront with fewer than 64 work-items is called a partial wavefront. Wavefronts that have fewer than a full set of work-items are called partial wavefronts. For the HD4000-family of devices, there are 64. 32, 16 work-items in a full wavefront. Work-items within a wavefront execute in lockstep.
write combining	Combining several smaller writes to memory into a single larger write to minimize any overhead associated with write commands.

Index