



Introduction to OpenCL™ Programming

Training Guide

© 2010 Advanced Micro Devices Inc. All rights reserved.

The contents of this document are provided in connection with Advanced Micro Devices, Inc. ("AMD") products. AMD makes no representations or warranties with respect to the accuracy or completeness of the contents of this publication and reserves the right to discontinue or make changes to products, specifications, product descriptions, and documentation at any time without notice. No license, whether express, implied, arising by estoppel or otherwise, to any intellectual property rights is granted by this publication. Except as set forth in AMD's Standard Terms and Conditions of Sale, AMD assumes no liability whatsoever, and disclaims any express or implied warranty, relating to its products including, but not limited to, the implied warranty of merchantability, fitness for a particular purpose, or infringement of any intellectual property right. AMD's products are not designed, intended, authorized or warranted for use as components in systems intended for surgical implant into the body, or in other applications intended to support or sustain life, or in any other application in which the failure of AMD's product could create a situation where personal injury, death, or severe property or environmental damage may occur. AMD reserves the right to discontinue or make changes to its products at any time without notice.

Trademarks

AMD, the AMD Arrow logo, ATI, the ATI logo, AMD Athlon, AMD LIVE!, AMD Phenom, AMD Sempron, AMD Turion, AMD64, All-in-Wonder, Avivo, Catalyst, CrossFireX, FirePro, FireStream, HyperMemory, OverDrive, PowerPlay, PowerXpress, Radeon, Remote Wonder, SurroundView, Theater, TV Wonder, The Ultimate Visual Experience, and combinations thereof are trademarks of Advanced Micro Devices, Inc.

OpenCL and the OpenCL logo are trademarks of Apple Inc., used by permission by Khronos.

Microsoft, Windows, and Vista are registered trademarks of the Microsoft Corporation in the United States and/or other jurisdictions.

Other names are for informational purposes only and may be trademarks of their respective owners.

Contents

Chapter 1 Course Overview	1
1.1 Objectives	1
1.2 Audience	1
1.3 References	1
Chapter 2 Introduction	3
2.1 What is GPU Compute?	3
2.2 A Brief History of GPU Compute	3
2.3 Heterogeneous Computing	4
Chapter 3 Introduction to OpenCL	7
3.1 What is OpenCL?	7
3.1.1 Benefits of OpenCL	7
3.2 Anatomy of OpenCL	8
3.2.1 Language Specification	8
3.2.2 Platform API	8
3.2.3 Runtime API	8
3.3 OpenCL Architecture	8
3.3.1 The Platform Model	9
3.3.2 The Execution Model	11
3.3.3 The Memory Model	14
Chapter 4 Getting Started with OpenCL	17
4.1 The Software Development Environment and Tools	17
4.1.1 Requirements	17
4.1.2 Installing on Windows	19
4.1.3 Installing on Linux	21
4.2 "Hello World" in OpenCL	22
4.3 Compiling OpenCL Source	24
4.3.1 Compiling on Linux	25
4.3.2 Compiling on Windows	25
Chapter 5 OpenCL Programming in Detail	29

5.1 Executing an OpenCL Program.....	29
5.2 Resource Setup.....	30
5.2.1 Query for Platform Information and OpenCL Devices.....	30
5.2.2 Query for OpenCL Device.....	31
5.2.3 Creating a Context.....	33
5.2.4 Creating the Command Queue.....	35
5.3 Kernel Programming and Compiling.....	35
5.3.1 Creating the Program Object.....	36
5.3.2 Building Program Executables.....	37
5.3.3 Creating Kernel Objects.....	38
5.3.4 Setting Kernel Arguments.....	39
5.4 Program Execution.....	41
5.4.1 Determining the Problem Space.....	41
5.4.2 Cleaning Up.....	43
5.5 Memory Objects.....	44
5.5.1 Creating Buffer Objects.....	44
5.5.2 Reading, Writing, and Copying Buffer Objects.....	45
5.5.3 Retaining and Releasing Buffer Objects.....	47
5.5.4 Creating Image Objects.....	47
5.5.5 Retaining and Releasing Image Objects.....	50
5.6 Synchronization.....	50
Chapter 6 The OpenCL C Language	57
6.1 Restrictions.....	57
6.2 Data Types.....	57
6.2.1 Scalar Data Types.....	57
6.2.2 Vector Data Types.....	58
6.2.3 Vector Operations.....	60
6.3 Type Casting and Conversions.....	60
6.4 Qualifiers.....	63
6.4.1 Address Space Qualifiers.....	63
6.4.2 Image Qualifiers.....	64
6.5 Built-in Functions.....	64
6.5.1 Work-item Functions.....	64
6.5.2 Image Access Functions.....	66
6.5.3 Synchronization Functions.....	66

Chapter 7 Application Optimization and Porting	69
7.1 Debugging OpenCL	69
7.1.1 Setting Up for Debugging	69
7.1.2 Setting Breakpoints	69
7.2 Performance Measurement	71
7.2.1 Using the ATI Stream Profiler	73
7.3 General Optimization Tips	75
7.3.1 Use Local Memory	75
7.3.2 Work-group Size	76
7.3.3 Loop Unrolling	77
7.3.4 Reduce Data and Instructions	77
7.3.5 Use Built-in Vector Types	77
7.4 Porting CUDA to OpenCL	77
7.4.1 General Terminology	78
7.4.2 Kernel Qualifiers	78
7.4.3 Kernel Indexing	78
7.4.4 Kernel Synchronization	79
7.4.5 General API Terminology	79
7.4.6 Important API Calls	80
7.4.7 Additional Tips	81
Chapter 8 Exercises	83
8.1 Matrix Transposition Exercise	83
8.1.1 Completing the Code	83
8.2 Matrix Multiplication Exercise	85
Appendix A Exercise Solutions	89
A.1 Matrix Transposition Solution	89
A.2 Matrix Multiplication Solution	93

Figures

Chapter 2 Introduction

Figure 2–1 ATI Stream SDK v1 Stack	4
------------------------------------------	---

Chapter 3 Introduction to OpenCL

Figure 3–1 OpenCL Platform Model	9
Figure 3–2 Stream Core Housing Five Processing Elements	11
Figure 3–3 Grouping Work-items Into Work-groups	12
Figure 3–4 Work-group Example	12

Chapter 5 OpenCL Programming in Detail

Figure 5–1 Typical OpenCL Platform Info Retrieved from clGetPlatformInfo()	31
Figure 5–2 Device Capabilities of the ATI Radeon HD 4770 GPU	33
Figure 5–3 Sample Build Log Output	38
Figure 5–4 Representation of Data Set in N-dimensional Space	41
Figure 5–5 Kernel Execution with One Device and One Queue	51
Figure 5–6 Two Devices with Two Queues, Unsynchronized	52
Figure 5–7 Two Devices with Two Queues Synchronized Using Events	53

Chapter 7 Application Optimization and Porting

Figure 7–1 Listing OpenCL Debugging Symbols	71
Figure 7–2 Setting a Conditional Breakpoint	71

Tables

Chapter 3 Introduction to OpenCL

Table 3–1 Simple Example of Scalar Versus Parallel Implementation	13
-----------------------------------------------------------------------------	----

Chapter 4 Getting Started with OpenCL

Table 4–1 GPUs	19
Table 4–2 Drivers (Minimum Version Requirements)	19

Chapter 5 OpenCL Programming in Detail

Table 5–1 List of device types supported by clGetDeviceIDs()	32
Table 5–2 Supported CL_MEM Flags	45
Table 5–3 Supported Channel Data Order	48
Table 5–4 Supported Channel Data Types	48
Table 5–5 Supported List param_name and Return Values for clGetEventInfo()	54

Chapter 6 The OpenCL C Language

Table 6–1 Supported Scalar Data Types	58
Table 6–2 Supported Vector Data Types	58

Chapter 7 Application Optimization and Porting

Table 7–1 Supported Profiling Data for clGetEventProfilingInfo()	72
Table 7–2 General Terminology Equivalents	78
Table 7–3 Kernel Qualifier Equivalents	78
Table 7–4 Indexing Terminology Equivalents Used in Kernel Functions	79
Table 7–5 Synchronization Terminology Equivalents Used in Kernel Functions	79
Table 7–6 General API Terminology Equivalents	80
Table 7–7 API Call Equivalents	80

Chapter 1

Course Overview

1.1 Objectives

This training session introduces participants to the fundamentals of the OpenCL™ (Open Computing Language) programming language. Areas covered include the following:

- Introduction to parallel computing with heterogeneous systems.
- Introduction to the OpenCL programming framework.
- Setting up the OpenCL development environment.
- Understand and using the OpenCL API.
- Optimizing an OpenCL application.

1.2 Audience

This course is intended for programmers. It assumes prior experience in the C programming language.

1.3 References

- [The OpenCL Specification v1.0](http://www.khronos.org/registry/cl/specs/opengl-1.0.48.pdf) (<http://www.khronos.org/registry/cl/specs/opengl-1.0.48.pdf>)
- [OpenCL 1.0 Reference Page](http://www.khronos.org/opencl/sdk/1.0/docs/man/xhtml/) (<http://www.khronos.org/opencl/sdk/1.0/docs/man/xhtml/>)
- [ATI Stream Software Development Kit \(SDK\)](http://developer.amd.com/stream) (<http://developer.amd.com/stream>)
- [ATI Stream SDK OpenCL Programming Guide](http://developer.amd.com/gpu_assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf) (http://developer.amd.com/gpu_assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf)
- [AMD Developer Forums - OpenCL](http://developer.amd.com/opencl) (<http://developer.amd.com/opencl>)
- [ATI Stream Profiler Knowledge Base](http://developer.amd.com/support/KnowledgeBase?Category=2&SubCategory=54) (<http://developer.amd.com/support/KnowledgeBase?Category=2&SubCategory=54>)

- [GNU GDB Documentation](http://www.gnu.org/software/gdb/documentation) (<http://www.gnu.org/software/gdb/documentation>)
- [OpenCL Zone](http://developer.amd.com/openclzone) (<http://developer.amd.com/openclzone>)
- [Image Convolution Tutorial](http://developer.amd.com/gpu/ATIStreamSDK/ImageConvolutionOpenCL/Pages/ImageConvolutionUsingOpenCL.aspx) (<http://developer.amd.com/gpu/ATIStreamSDK/ImageConvolutionOpenCL/Pages/ImageConvolutionUsingOpenCL.aspx>)

Note: The information contained in this guide is current at the time of publication, and may not apply to future technologies, products, or designs.

Chapter 2

Introduction

This chapter provides a brief overview and history of GPU compute.

2.1 What is GPU Compute?

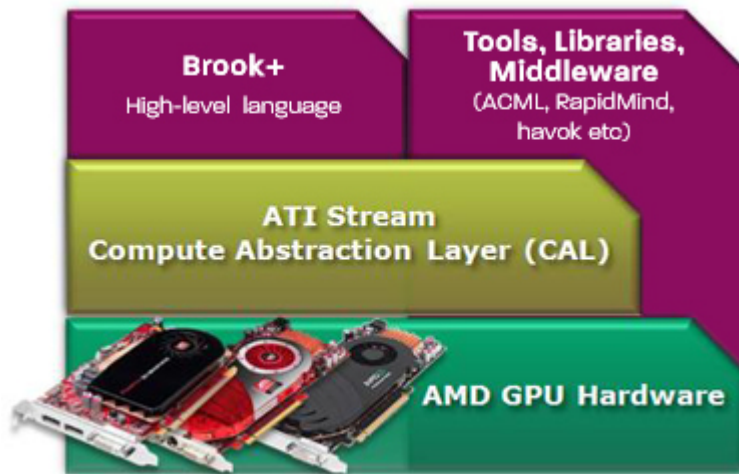
GPU compute is the use of Graphics Processing Units (GPUs) for general purpose computation instead of traditional graphics rendering. GPUs are high performance multi-core processors that can be used to accelerate a wide variety of applications using parallel computing. The highly programmable nature of GPUs makes them useful as high-speed coprocessors that perform beyond their intended graphics capabilities.

2.2 A Brief History of GPU Compute

The birth of the GPU compute revolution occurred in November 2006, when AMD introduced its Close to Metal (CTM) low-level hardware programming interface that allowed developers to take advantage of the native instruction set and memory of modern GPUs for general-purpose computation. CTM provided developers with the low-level, deterministic, and repeatable access to hardware necessary to develop essential tools such as compilers, debuggers, math libraries, and application platforms. With the introduction of CTM, a new class of applications was realized. For example, a GPU-accelerated client for Folding@Home was created that was capable of achieving a 20- to 30-fold speed increase over its predecessor (for more on this story, see <http://folding.stanford.edu/English/FAQ-ATI#ntoc5>.)

AMD's continuing commitment to provide a fully-featured software development environment that exposes the power of AMD GPUs resulted in the introduction of the ATI Stream SDK v1 in December 2007. The ATI Stream SDK v1 added a new high-level language, called ATI Brook+. CTM evolved into ATI CAL (Compute Abstraction Layer), the supporting API layer for Brook+. The introduction of the ATI Stream SDK v1 meant that AMD was able to provide both high-level and low-level tools for general-purpose access to AMD GPU hardware.

Figure 2–1 ATI Stream SDK v1 Stack



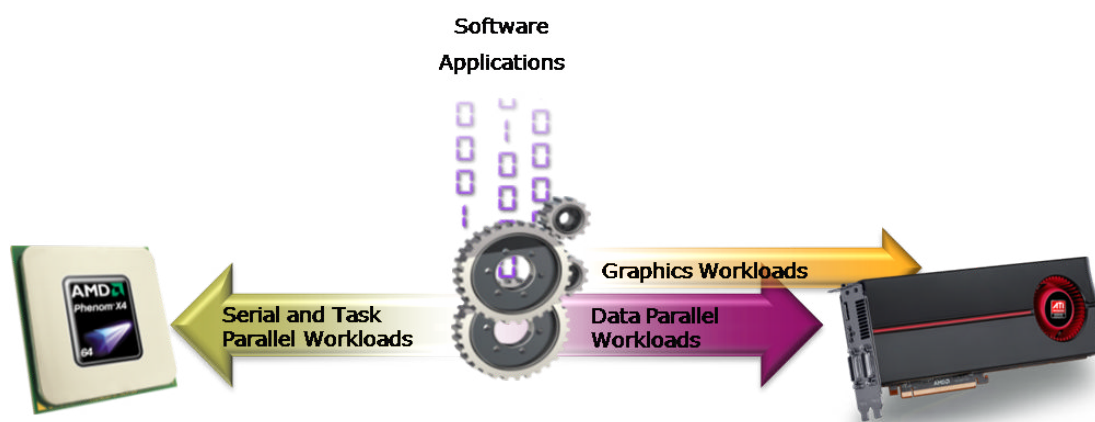
A drawback to using the ATI Stream SDK v1 was that applications created with the SDK ran only on AMD GPU hardware. To achieve greater adoption for general-purpose computing, an open standard was needed.

In June 2008, AMD, along with various industry players in GPU compute and other accelerator technologies, formed the OpenCL working group under The Khronos Group. Khronos, already known for leadership in other open specifications such as OpenGL®, was a logical choice to drive the OpenCL specification. Five months later, the group completed the technical specification for OpenCL 1.0 and released it to the public. Immediately after that release, AMD announced its intent to adopt the OpenCL programming standard and integrate a compliant compiler and runtime into its free ATI Stream SDK v2. In December 2009, AMD released the ATI Stream SDK v2.0 with OpenCL 1.0 support.

2.3 Heterogeneous Computing

Heterogeneous computing involves the use of a various types of computational units. A computation unit can be a general-purpose processing unit (such as a CPU), a graphics processing unit (such as a GPU), or a special-purpose processing unit (such as digital signal processor, or DSP).

In the past, most computer applications were able to scale with advances in CPU technologies. With modern computer applications requiring interactions with various systems (such as audio/video systems, networked applications, etc.) even the advances in CPU technology proved insufficient to cope with this need. To achieve greater performance gains, specialized hardware was required, making the system heterogeneous. The addition of various types of computation units in these heterogeneous systems allows application designers to select the most suitable one on which to perform tasks.



Chapter 3

Introduction to OpenCL

This chapter introduces OpenCL and describes the anatomy and architecture of the OpenCL API.

3.1 What is OpenCL?

The Open Computing Language (OpenCL) is an open and royalty-free parallel computing API designed to enable GPUs and other coprocessors to work in tandem with the CPU, providing additional raw computing power. As a standard, OpenCL 1.0 was released on December 8, 2008, by The Khronos Group, an independent standards consortium.

Developers have long sought to divide computing problems into a mix of concurrent subsets, making it feasible for a GPU to be used as a math coprocessor working with the CPU to better handle general problems. The potential of this heterogeneous computing model was encumbered by the fact that programmers could only choose proprietary programming languages, limiting their ability to write vendor-neutral, cross-platform applications. Proprietary implementations such as NVIDIA's CUDA limited the hardware choices of developers wishing to run their application on another system without having to retool it.

3.1.1 Benefits of OpenCL

A primary benefit of OpenCL is substantial acceleration in parallel processing. OpenCL takes all computational resources, such as multi-core CPUs and GPUs, as peer computational units and correspondingly allocates different levels of memory, taking advantage of the resources available in the system. OpenCL also complements the existing OpenGL® visualization API by sharing data structures and memory locations without any copy or conversion overhead.

A second benefit of OpenCL is cross-vendor software portability. This low-level layer draws an explicit line between hardware and the upper software layer. All the hardware implementation specifics, such as drivers and runtime, are invisible to the upper-level software programmers through the use of high-level abstractions, allowing the developer to take advantage of the best hardware without having to reshuffle the upper software infrastructure. The change from proprietary programming to open standard also contributes to the acceleration of general computation in a cross-vendor fashion.

3.2 Anatomy of OpenCL

The OpenCL development framework is made up of three main parts:

- Language specification
- Platform layer API
- Runtime API

3.2.1 Language Specification

The language specification describes the syntax and programming interface for writing kernel programs that run on the supported accelerator (GPU, multi-core CPU, or DSP). Kernels can be precompiled or the developer can allow the OpenCL runtime to compile the kernel program at runtime.

The OpenCL programming language is based on the ISO C99 specification with added extensions and restrictions. Additions include vector types and vector operations, optimized image access, and address space qualifiers. Restrictions include the absence of support for function pointers, bit-fields, and recursion. The C language was selected as the first base for the OpenCL language due to its prevalence in the developer community. To ensure consistent results across different platforms, OpenCL C also provides a well-defined IEEE 754 numerical accuracy for all floating point operations and a rich set of built-in functions. For complete language details on the language, see the OpenCL specification.

3.2.2 Platform API

The platform-layer API gives the developer access to software application routines that can query the system for the existence of OpenCL-supported devices. This layer also lets the developer use the concepts of device context and work-queues to select and initialize OpenCL devices, submit work to the devices, and enable data transfer to and from the devices.

3.2.3 Runtime API

The OpenCL framework uses contexts to manage one or more OpenCL devices. The runtime API uses contexts for managing objects such as command queues, memory objects, and kernel objects, as well as for executing kernels on one or more devices specified in the context.

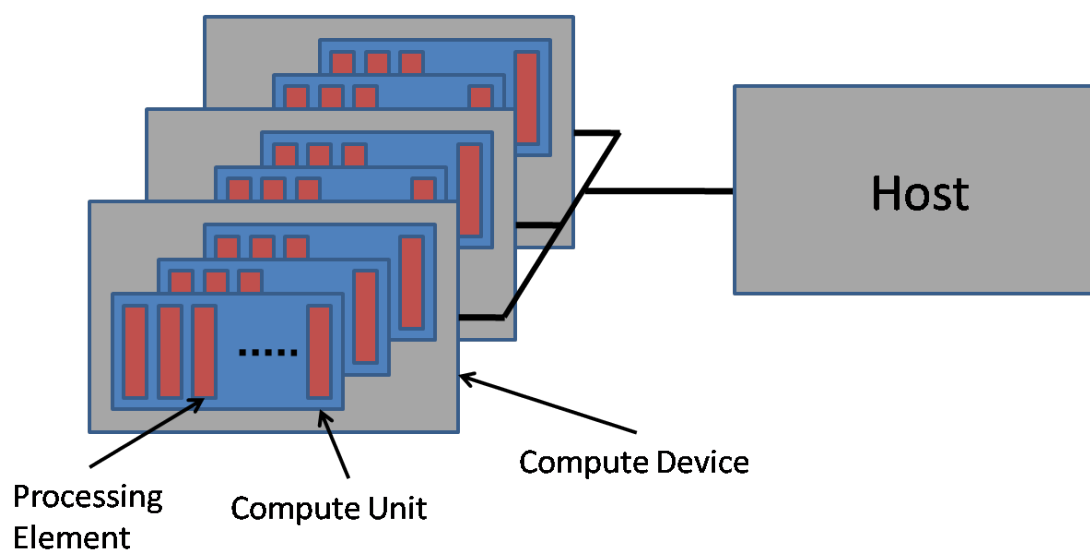
3.3 OpenCL Architecture

3.3.1 The Platform Model

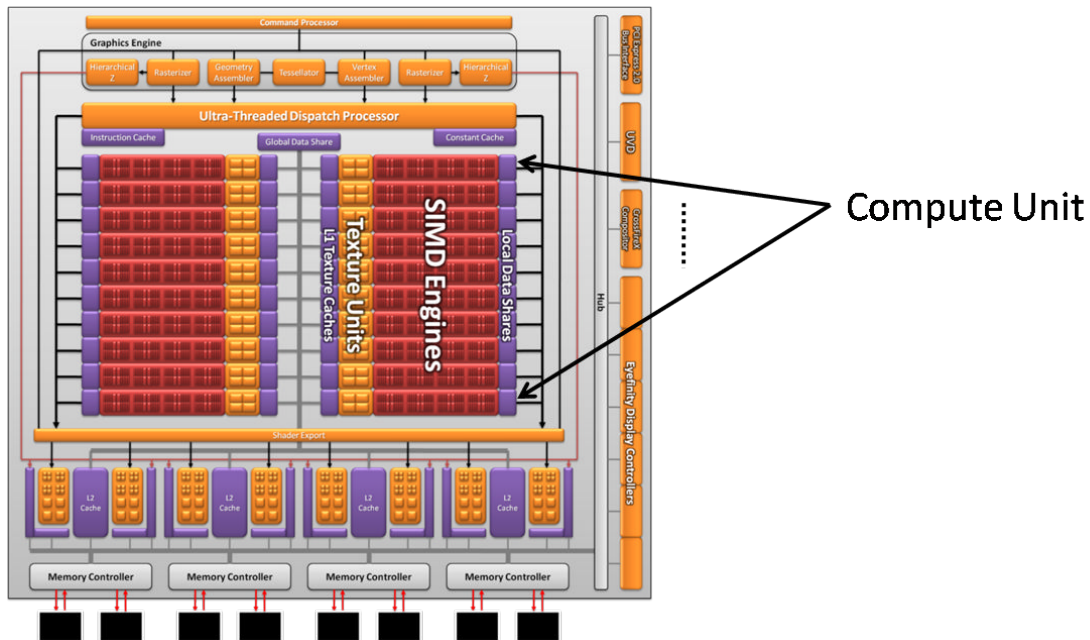
The OpenCL platform model is defined as a host connected to one or more OpenCL devices. [Figure 3-1 OpenCL Platform Model](#) shows the platform model comprising one host plus multiple compute devices, each having multiple compute units, each of which have multiple processing elements.

A host is any computer with a CPU running a standard operating system. OpenCL devices can be a GPU, DSP, or a multi-core CPU. An OpenCL device consists of a collection of one or more compute units (cores). A compute unit is further composed of one or more processing elements. Processing elements execute instructions as SIMD (Single Instruction, Multiple Data) or SPMD (Single Program, Multiple Data). SPMD instructions are typically executed on general purpose devices such as CPUs, while SIMD instructions require a vector processor such as a GPU or vector units in a CPU.

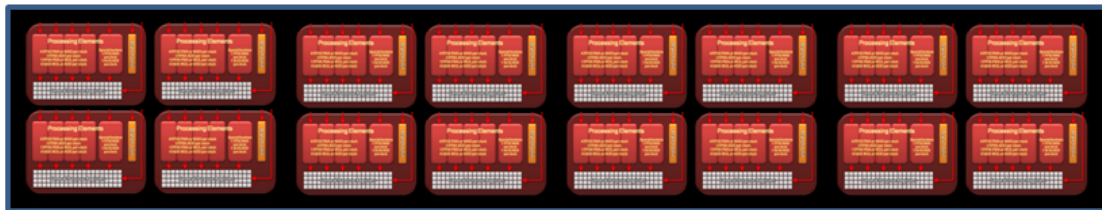
Figure 3-1 OpenCL Platform Model



The following depiction of the ATI Radeon™ HD 5870 GPU architecture illustrates a compute device construct. The ATI Radeon HD 5870 GPU is made up of 20 SIMD units, which translates to 20 compute units in OpenCL:

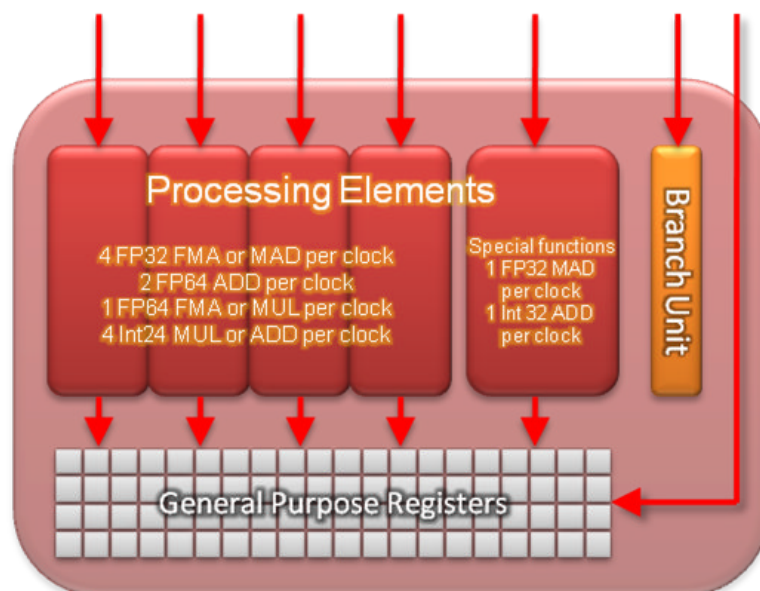


Compute Unit



Each SIMD unit contains 16 stream cores, and each stream core houses five processing elements. Thus, each compute unit in the ATI Radeon HD 5870 has 80 (16×5) processing elements.

Figure 3–2 Stream Core Housing Five Processing Elements



3.3.2 The Execution Model

The OpenCL execution model comprises two components: kernels and host programs. Kernels are the basic unit of executable code that runs on one or more OpenCL devices. Kernels are similar to a C function that can be data- or task-parallel. The host program executes on the host system, defines devices context, and queues kernel execution instances using command queues. Kernels are queued in-order, but can be executed in-order or out-of-order.

3.3.2.1 Kernels

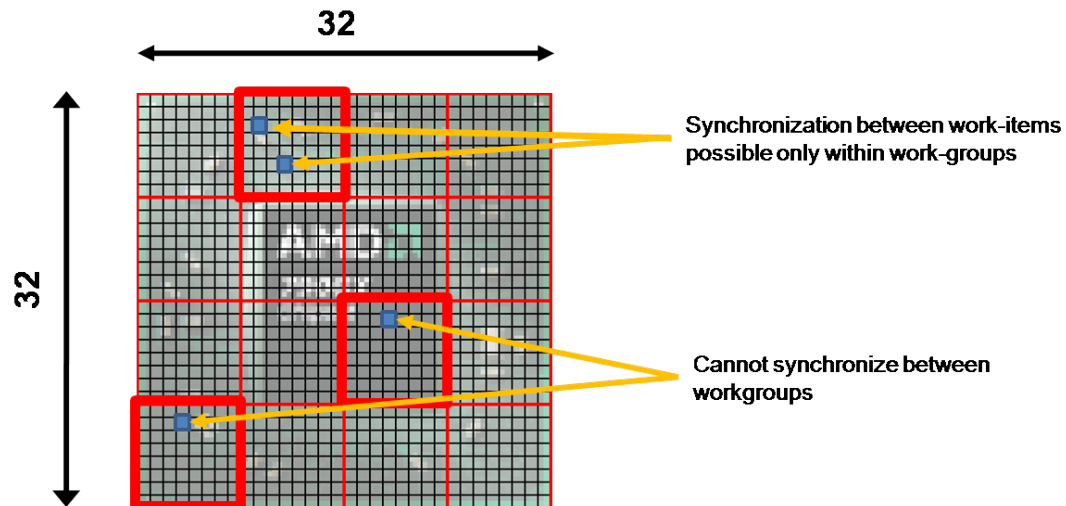
OpenCL exploits parallel computation on compute devices by defining the problem into an N-dimensional index space. When a kernel is queued for execution by the host program, an index space is defined. Each independent element of execution in this index space is called a work-item. Each work-item executes the same kernel function but on different data. When a kernel command is placed into the command queue, an index space must be defined to let the device keep track of the total number of work-items that require execution. The N-dimensional index space can be N=1, 2, or 3. Processing a linear array of data would be considered N=1; processing an image would be N=2, and processing a 3D volume would be N=3.

Processing a 1024x1024 image would be handled this way: The global index space comprises a 2-dimensional space of 1024 by 1024 consisting of 1 kernel execution (or work-item) per pixel with a total of 1,048,576 total executions. Within this index space, each work-item is assigned a unique global ID. The work-item for pixel x=30, y=22 would have global ID of (30,22).

OpenCL also allows grouping of work-items together into work-groups, as shown in the following figure. The size of each work-group is defined by its own local index space. All work-items in the same work-group are executed together on the same

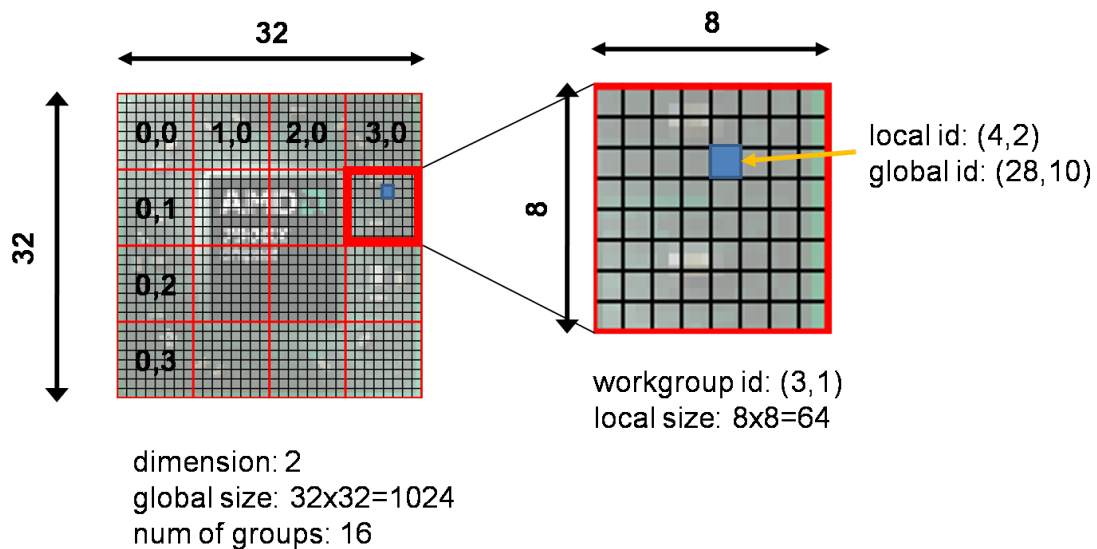
device. The reason for executing on one device is to allow work-items to share local memory and synchronization. Global work-items are independent and cannot be synchronized. Synchronization is only allowed between the work-items in a work-group.

Figure 3-3 Grouping Work-items Into Work-groups



The following example shows a two-dimensional image with a global size of 1024 (32x32). The index space is divided into 16 work-groups. The highlighted work-group has an ID of (3,1) and a local size of 64 (8x8). The highlighted work-item in the work-group has a local ID of (4,2), but can also be addressed by its global ID of (28,10).

Figure 3-4 Work-group Example



The following example illustrates how a kernel in OpenCL is implemented. In this example, each element is squared in a linear array. Normally, a scalar function would be required with a simple *for* loop iterating through the elements in the array and then squaring it. The data-parallel approach is to read an element from the array *in parallel*, perform the operation in parallel, and write it to the output. Note that the code segment on the right does not have a *for* loop: It simply reads the index value for the particular kernel instance, performs the operation, and writes the output.

Table 3–1 Simple Example of Scalar Versus Parallel Implementation

Scalar C Function	Data-Parallel Function
<pre>void square(int n, const float *a, float *result) { int i; for (i=0; i<n; i++) result[i] = a[i]*a[i]; }</pre>	<pre>kernel void dp_square (global const float *a, global float *result) { int id= get_global_id(0); result[id] = a[id]*a[id]; } // dp_square execute over "n" work-items</pre>

The OpenCL execution model supports two categories of kernels: OpenCL kernels and native kernels.

OpenCL kernels are written in the OpenCL C language and compiled with the OpenCL compiler. All devices that are OpenCL-compliant support execution of OpenCL kernels.

Native kernels are extension kernels that could be special functions defined in application code or exported from a library designed for a particular accelerator. The OpenCL API includes functions to query capabilities of devices to determine if native kernels are supported.

If native kernels are used, developers should be aware that the code may not work on other OpenCL devices.

3.3.2.2 Host Program

The host program is responsible for setting up and managing the execution of kernels on the OpenCL device through the use of context. Using the OpenCL API, the host can create and manipulate the context by including the following resources:

- **Devices** — A set of OpenCL devices use by the host to execute kernels.
- **Program Objects** — The program source or program object that implements a kernel or collection of kernels.
- **Kernels** — The specific OpenCL functions that execute on the OpenCL device.
- **Memory Objects** — A set of memory buffers or memory maps common to the host and OpenCL devices.

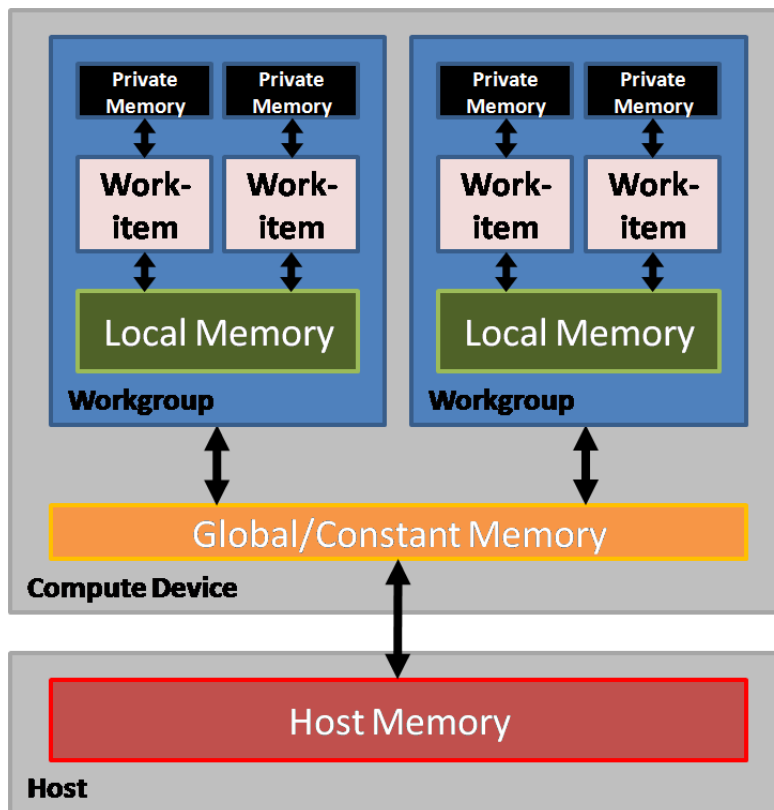
After the context is created, command queues are created to manage execution of the kernels on the OpenCL devices that were associated with the context. Command queues accept three types of commands:

- **Kernel execution commands** run the kernel command on the OpenCL devices.
- **Memory commands** transfer memory objects between the memory space of the host and the memory space of the OpenCL devices.
- **Synchronization commands** define the order in which commands are executed.

Commands are placed into the command queue in-order and execute either in-order or out-of-order. In in-order mode, the commands are executed serially as they are placed onto the queue. In out-of-order mode, the order the commands execute is based on the synchronization constraints placed on the command.

3.3.3 The Memory Model

Since common memory address space is unavailable on the host and the OpenCL devices, the OpenCL memory model defines four regions of memory accessible to work-items when executing a kernel. The following figure shows the regions of memory accessible by the host and the compute device:



Global memory is a memory region in which all work-items and work-groups have read and write access on both the compute device and the host. This region of memory can be allocated only by the host during runtime.

Constant memory is a region of global memory that stays constant throughout the execution of the kernel. Work-items have only read access to this region. The host is permitted both read and write access.

Local memory is a region of memory used for data-sharing by work-items in a work-group. All work-items in the same work-group have both read and write access.

Private memory is a region that is accessible to only one work-item.

In most cases, host memory and compute device memory are independent of one another. Thus, memory management must be explicit to allow the sharing of data between the host and the compute device. This means that data must be explicitly moved from host memory to global memory to local memory and back. This process works by enqueueing read/write commands in the command queue. The commands placed into the queue can either be blocking or non-blocking. Blocking means that the host memory command waits until the memory transaction is complete before continuing. Non-blocking means the host simply puts the command in the queue and continues, not waiting until the memory transaction is complete.

Chapter 4

Getting Started with OpenCL

This chapter discusses how to set up the OpenCL development environment in the Microsoft® Windows® and Linux® platforms.

4.1 The Software Development Environment and Tools

4.1.1 Requirements

The ATI Stream Software Development Kit (SDK) v2 is required to create OpenCL applications on the AMD platform. The SDK for Windows or Linux may be downloaded free from the [ATI Stream SDK v2 Product Page](http://developer.amd.com/stream) (<http://developer.amd.com/stream>).

File Name	Launch Date	Bitness	Description
Linux® (openSUSE™ 11.0, Ubuntu® 9.10)			
ati-stream-sdk-v2.01-lnx32.tgz (34.2MB)	02/11/2010	32-bit	ATI Stream SDK built for 32-bit Linux®
ati-stream-sdk-v2.01-lnx64.tgz (59.2MB)	02/11/2010	64-bit	ATI Stream SDK built for 64-bit Linux®
Linux® (Red Hat® Enterprise Linux® 5.3)			
ati-stream-sdk-v2.01-rhel32.tgz (35.3MB)	02/11/2010	32-bit	ATI Stream SDK built for 32-bit Red Hat® Enterprise Linux®
ati-stream-sdk-v2.01-rhel64.tgz (61.0MB)	02/11/2010	64-bit	ATI Stream SDK built for 64-bit Red Hat® Enterprise Linux®
Windows Vista® SP1 / Windows® 7			
ati-stream-sdk-v2.01-vista-win7-32.exe (49.2MB)	02/11/2010	32-bit	ATI Stream SDK built for 32-bit Microsoft® Windows Vista® and Microsoft® Windows® 7
ati-stream-sdk-v2.01-vista-win7-64.exe (91.9MB)	02/11/2010	64-bit	ATI Stream SDK built for 64-bit Microsoft® Windows Vista® and Microsoft® Windows® 7
Windows® XP SP3 (32-bit) / SP2 (64-bit)			
ati-stream-sdk-v2.01-xp32.exe (49.1MB)	02/11/2010	32-bit	ATI Stream SDK built for 32-bit Microsoft® Windows® XP
ati-stream-sdk-v2.01-xp64.exe (91.7MB)	02/11/2010	64-bit	ATI Stream SDK built for 64-bit Microsoft® Windows® XP

At the time of publication of this guide, the following operating system, compilers, CPUs, GPUs, and drivers were supported by the ATI Stream SDK v2. For the latest list of supported configurations, check the [ATI Stream SDK v2 Product Page](#).

4.1.1.1 Supported Operating Systems

Windows	<ul style="list-style-type: none"> Windows® XP SP3 (32-bit), SP2(64-bit) Windows Vista® SP1 (32/64-bit) Windows® 7 (32/64-bit)
Linux	<ul style="list-style-type: none"> openSUSE™ 11.1 (32/64-bit) Ubuntu® 9.10 (32/64-bit) Red Hat® Enterprise Linux 5.3 (32/64-bit)

4.1.1.2 Supported Compilers

Windows	<ul style="list-style-type: none"> Microsoft Visual Studio 2008 Professional Edition
Linux	<ul style="list-style-type: none"> GNU Compiler Collection (GCC) 4.3 or later Intel® C Compiler (ICC) 11.x

4.1.1.3 Supported GPUs and Drivers

Table 4–1 GPUs

ATI Radeon HD	5970, 5870, 5850, 5770, 5670, 5570, 5450, 4890, 4870 X2, 4870, 4850, 4830, 4770, 4670, 4650, 4550, 4350
ATI FirePro™	V8800, V8750, V8700, V7800, V7750 V5800, V5700, V4800, V3800, V3750
AMD FireStream™	9270, 9250
ATI Mobility Radeon™ HD	5870, 5850, 5830, 5770, 5730, 5650, 5470, 5450, 5430, 4870, 4860, 4850, 4830, 4670, 4650, 4500 series, 4300 series
ATI Mobility FirePro™	M7820, M7740, M5800
ATI Radeon Embedded	E4690 Discrete GPU

Table 4–2 Drivers (Minimum Version Requirements)

ATI Radeon HD	ATI Catalyst™ 10.4
ATI FirePro	ATI FirePro Unified Driver 8.723
AMD FireStream	ATI Catalyst 10.4
ATI Mobility Radeon HD	ATI Catalyst Mobility 10.4
ATI Mobility FirePro	Contact notebook manufacturer for appropriate driver.
ATI Radeon Embedded	Contact notebook manufacturer for appropriate driver.

4.1.1.4 Supported Processors

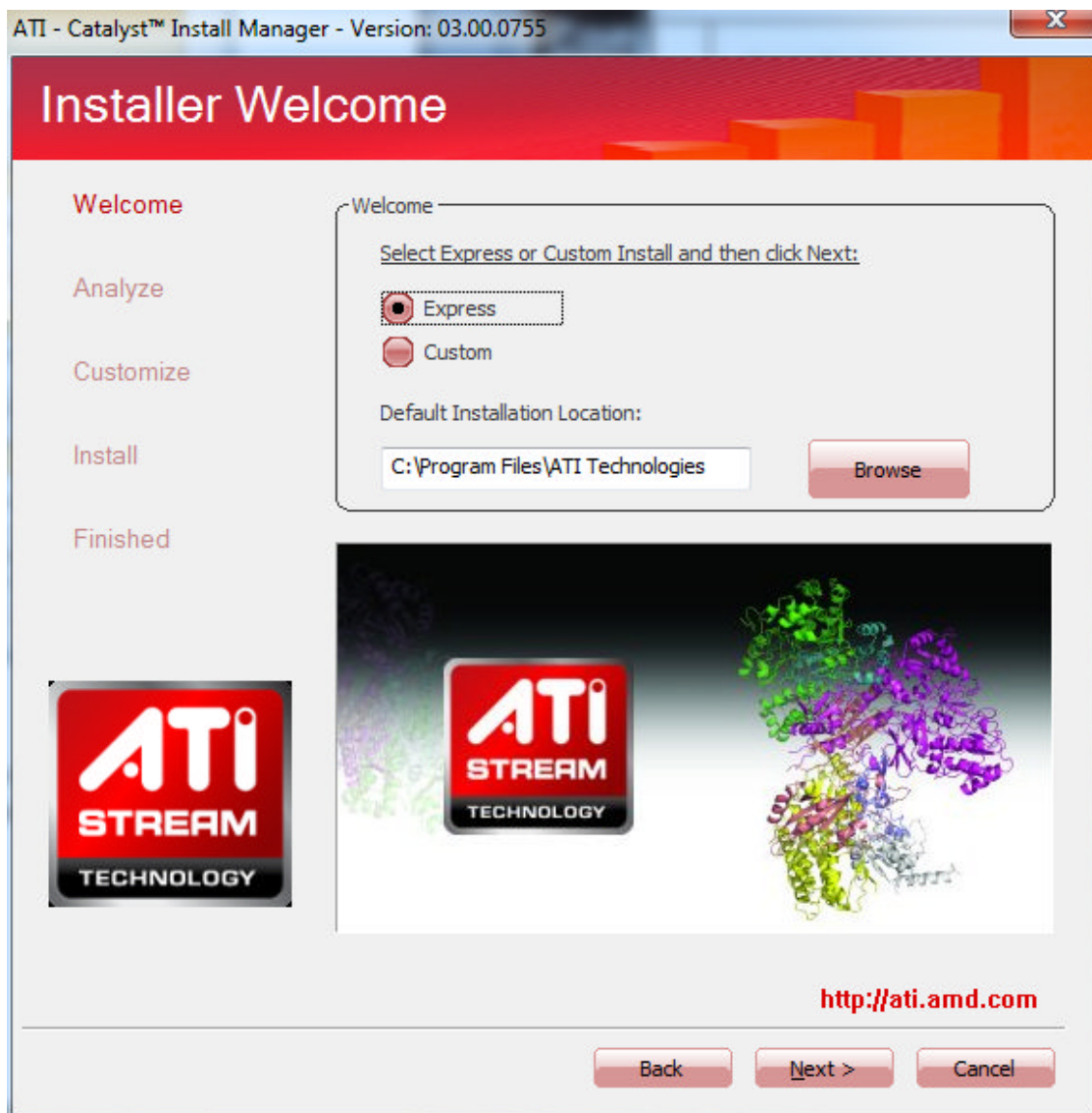
Any X86 CPU with Stream SIMD Extensions (SSE) 3.x or later.

4.1.2 Installing on Windows

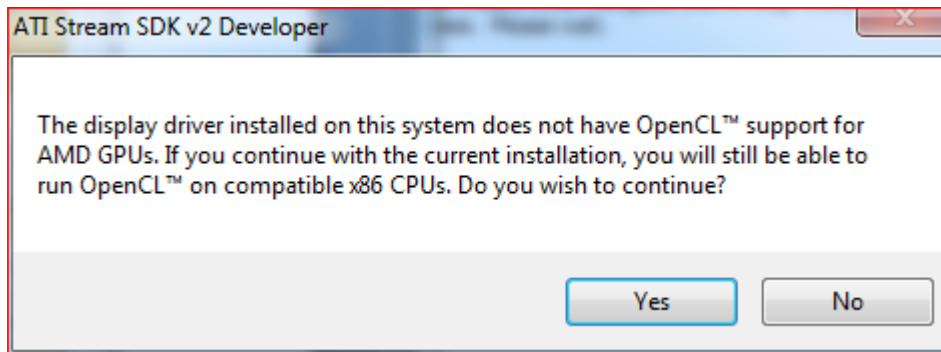
Administrative access is required to install the ATI Stream SDK v2 on a Windows system.

To install, run the ATI Stream SDK v2 executable installation program and follow the instructions in the setup Wizard.

Selecting Express Installation (as shown in the following image) installs the ATI Stream SDK v2, sample code, and the ATI Stream Profiler.



During the installation the installer may analyze the system to detect the currently installed graphics hardware and software drivers. If either the graphics hardware or the software drivers do not support OpenCL, a warning popup may appear as shown in the following image.



4.1.3 Installing on Linux

Root permission is required to install the ATI Stream SDK v2 on Linux.

To install the downloaded ATI Stream SDK v2:

1. Untar the SDK to a location of your choice:

```
tar -zxvf ati-stream-sdk-vX.XX-lnxYY.tgz
```

where X.XX is the downloaded SDK version, and YY is either 32 or 64 (representing 32-bit or 64-bit).

2. Add `ATISTREAMSDKROOT` to your environment variables:

```
export ATISTREAMSDKROOT=<location in which the SDK was extracted>
```

3. If the sample code was installed, add `ATISTREAMSDKSAMPLESROOT` to your environment variables:

```
export ATISTREAMSDKSAMPLESROOT=<location in which the SDK was extracted>
```

4. Add the appropriate path to the `LD_LIBRARY_PATH`:

On 32-bit systems:

```
export LD_LIBRARY_PATH=$ATISTREAMSDKROOT/lib/x86:$LD_LIBRARY_PATH
```

On 64-bit systems:

```
export LD_LIBRARY_PATH=$ATISTREAMSDKROOT/lib/  
x86_64:$LD_LIBRARY_PATH
```

5. Register the OpenCL ICD to allow applications to run:

```
sudo -s
```

```
mkdir -p /etc/OpenCL/vendors
```

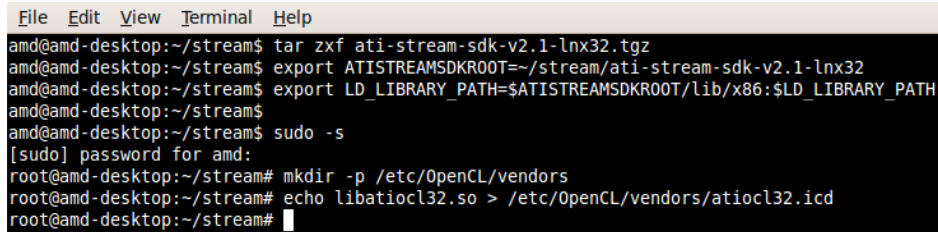
On all systems:

```
echo libatiocl32.so > /etc/OpenCL/vendors/atiocl32.icd
```

On 64-bit systems, also perform the following:

```
echo libatiocl64.so > /etc/OpenCL/vendors/atiocl64.icd
```

The following image shows the steps for installing the ATI Stream SDK v2 on a 32-bit Linux system:



```
File Edit View Terminal Help
amd@amd-desktop:~/stream$ tar xzf ati-stream-sdk-v2.1-lnx32.tgz
amd@amd-desktop:~/stream$ export ATISTREAMSDKROOT=~/stream/ati-stream-sdk-v2.1-lnx32
amd@amd-desktop:~/stream$ export LD_LIBRARY_PATH=$ATISTREAMSDKROOT/lib/x86:$LD_LIBRARY_PATH
amd@amd-desktop:~/stream$
amd@amd-desktop:~/stream$ sudo -s
[sudo] password for amd:
root@amd-desktop:~/stream# mkdir -p /etc/OpenCL/vendors
root@amd-desktop:~/stream# echo libatiocl32.so > /etc/OpenCL/vendors/atiocl32.icd
root@amd-desktop:~/stream#
```

4.2 "Hello World" in OpenCL

A typical OpenCL application starts by querying the system for the availability of OpenCL devices. When devices have been identified, a context allows the creation of command queues, creation of programs and kernels, management of memory between host and OpenCL devices, and submission of kernels for execution.

The following example code illustrates the fundamentals of all OpenCL applications. It takes a input buffer, squares the values, and stores the results in an output buffer, illustrating the relationship between device, context, program, kernel, command queue, and buffers.


```

001 #include <stdio.h>
002 #include "CL/cl.h"
003
004 #define DATA_SIZE 10
005
006
007 const char *KernelSource =
008 "__kernel void hello(__global float *input, __global float *output)\n"
009 "{\n"
010 "    size_t id = get_global_id(0);\n"
011 "    output[id] = input[id] * input[id];\n"
012 "}\n"
013 "\n";
014
015 int main(void)
016 {
017     cl_context context;
018     cl_context_properties properties[3];
019     cl_kernel kernel;
020     cl_command_queue command_queue;
021     cl_program program;
022     cl_int err;
023     cl_uint num_of_platforms=0;
024     cl_platform_id platform_id;
025     cl_device_id device_id;
026     cl_uint num_of_devices=0;
027     cl_mem input, output;
028     size_t global;
029
030     float inputData[DATA_SIZE]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
031     float results[DATA_SIZE]={0};
032
033     int i;
034
035     // retrieves a list of platforms available
036     if (clGetPlatformIDs(1, &platform_id, &num_of_platforms)!= CL_SUCCESS)
037     {
038         printf("Unable to get platform_id\n");
039         return 1;
040     }
041
042     // try to get a supported GPU device
043     if (clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU, 1, &device_id,
044         &num_of_devices) != CL_SUCCESS)
045     {
046         printf("Unable to get device_id\n");
047         return 1;
048     }
049
050     // context properties list - must be terminated with 0
051     properties[0]= CL_CONTEXT_PLATFORM;
052     properties[1]= (cl_context_properties) platform_id;
053     properties[2]= 0;
054
055     // create a context with the GPU device
056     context = clCreateContext(properties,1,&device_id,NULL,NULL,&err);
057
058     // create command queue using the context and device
059     command_queue = clCreateCommandQueue(context, device_id, 0, &err);
060
061     // create a program from the kernel source code

```

```

061     program = clCreateProgramWithSource(context,1,(const char **)
        &KernelSource, NULL, &err);
062
063     // compile the program
064     if (clBuildProgram(program, 0, NULL, NULL, NULL, NULL) != CL_SUCCESS)
065     {
066         printf("Error building program\n");
067         return 1;
068     }
069
070     // specify which kernel from the program to execute
071     kernel = clCreateKernel(program, "hello", &err);
072
073     // create buffers for the input and output
074     input = clCreateBuffer(context, CL_MEM_READ_ONLY,
        sizeof(float) * DATA_SIZE, NULL, NULL);
075     output = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
        sizeof(float) * DATA_SIZE, NULL, NULL);
076
077     // load data into the input buffer
078     clEnqueueWriteBuffer(command_queue, input, CL_TRUE, 0,
        sizeof(float) * DATA_SIZE, inputData, 0, NULL, NULL);
079
080     // set the argument list for the kernel command
081     clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
082     clSetKernelArg(kernel, 1, sizeof(cl_mem), &output);
083     global=DATA_SIZE;
084
085     // enqueue the kernel command for execution
086     clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, &global,
        NULL, 0, NULL, NULL);
087     clFinish(command_queue);
088
089     // copy the results from out of the output buffer
090     clEnqueueReadBuffer(command_queue, output, CL_TRUE, 0,
        sizeof(float) * DATA_SIZE, results, 0, NULL, NULL);
091
092     // print the results
093     printf("output: ");
094     for (i=0;i<DATA_SIZE; i++)
095     {
096         printf("%f ",results[i]);
097     }
098
099     // cleanup - release OpenCL resources
100     clReleaseMemObject(input);
101     clReleaseMemObject(output);
102     clReleaseProgram(program);
103     clReleaseKernel(kernel);
104     clReleaseCommandQueue(command_queue);
105     clReleaseContext(context);
106
107     return 0;
108 }

```

4.3 Compiling OpenCL Source

Note: The ATI Stream SDK v2 must be installed to compile an OpenCL program. See [4.1.1 Requirements](#) for detailed ATI Stream SDK v2 installation procedures.

4.3.1 Compiling on Linux

Ensure the required compiler GNU Compiler Collection (GCC) 4.3 or later is correctly installed. To compile the sample "Hello World" program described earlier in this guide, enter the following at the command line:

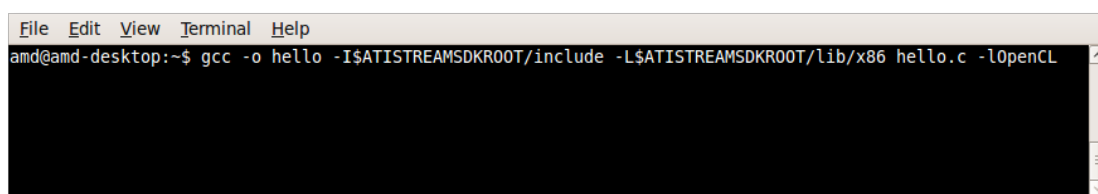
For a 32-bit system:

```
gcc -o hello -I$ATISTREAMSDKROOT/include -L$ATISTREAMSDKROOT/lib/x86  
hello.c -lOpenCL
```

For a 64-bit system:

```
gcc -o hello -I$ATISTREAMSDKROOT/include -L$ATISTREAMSDKROOT/lib/x86_64  
hello.c -lOpenCL
```

Where `ATISTREAMSDKROOT` environment points to the ATI Stream SDK v2 installation location.

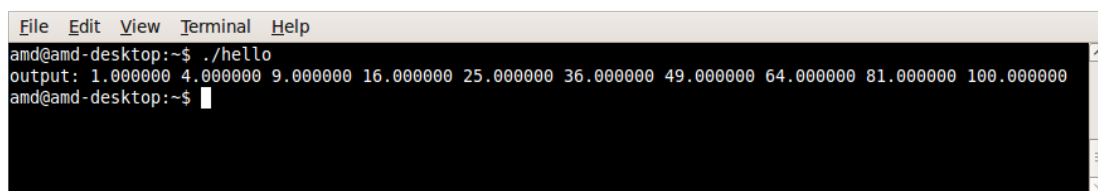


```
File Edit View Terminal Help  
amd@amd-desktop:~$ gcc -o hello -I$ATISTREAMSDKROOT/include -L$ATISTREAMSDKROOT/lib/x86 hello.c -lOpenCL
```

To execute the program, ensure that the `LD_LIBRARY_PATH` environment variable is set to find `libOpenCL.so`, then enter:

```
./hello
```

The following image shows the sample program output:

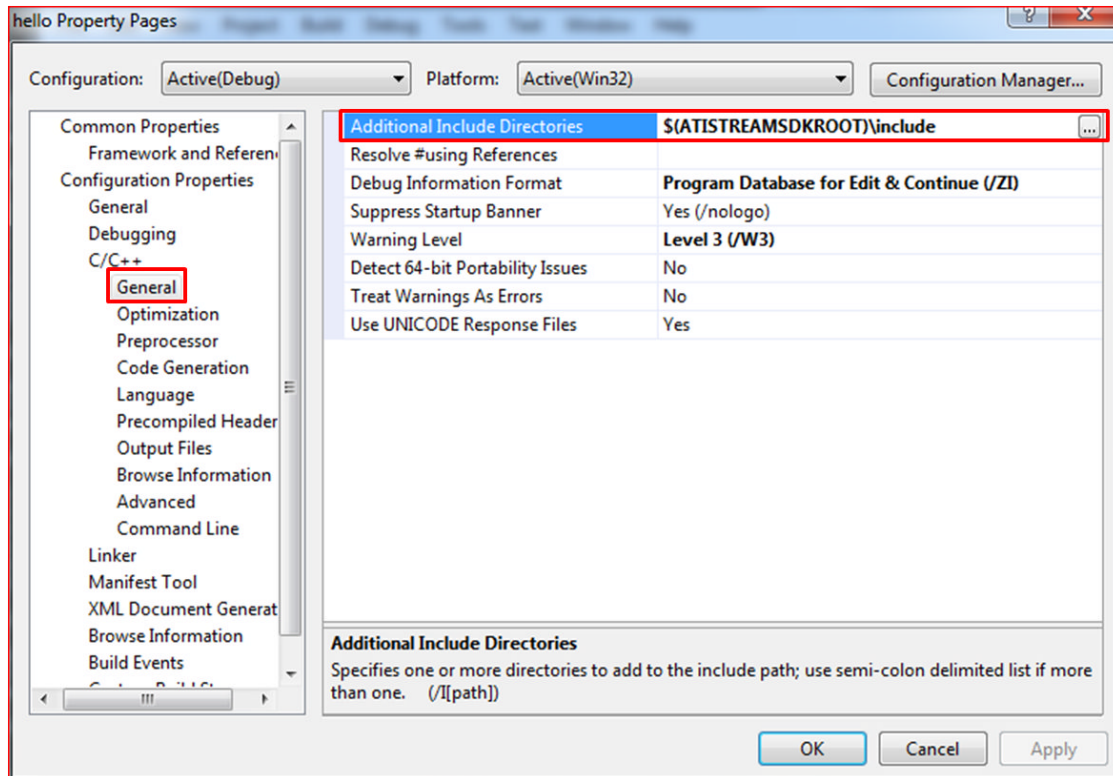


```
File Edit View Terminal Help  
amd@amd-desktop:~$ ./hello  
output: 1.000000 4.000000 9.000000 16.000000 25.000000 36.000000 49.000000 64.000000 81.000000 100.000000  
amd@amd-desktop:~$
```

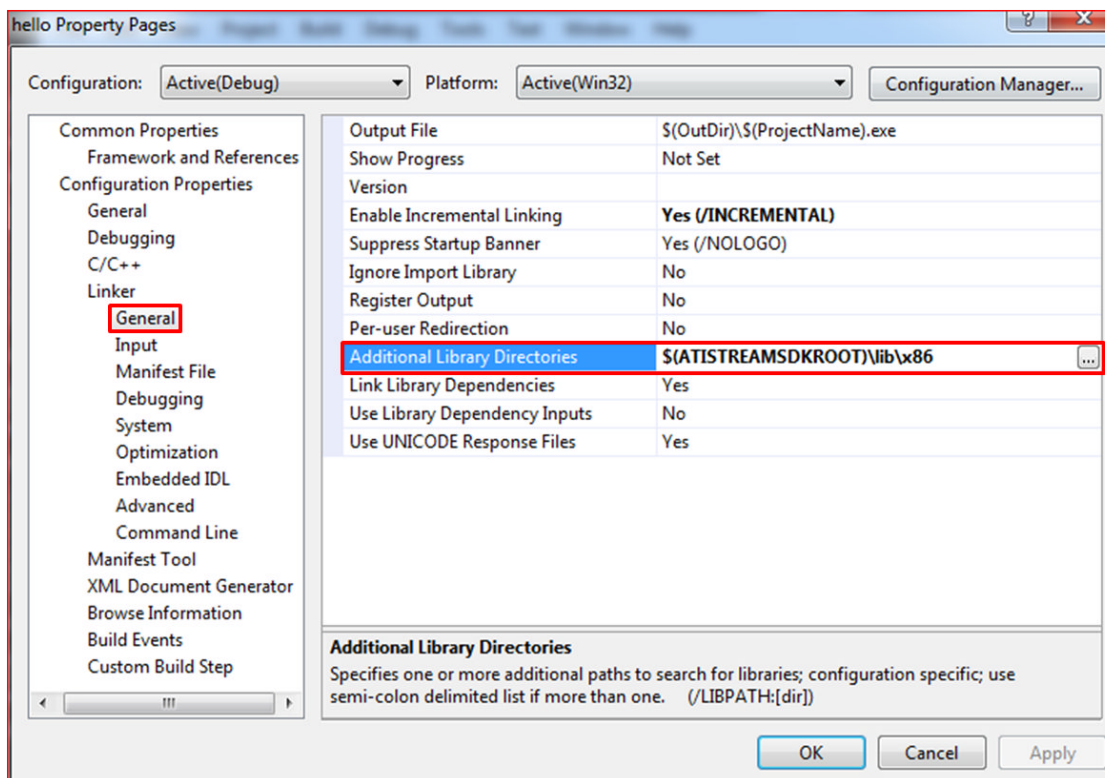
4.3.2 Compiling on Windows

To compile using Microsoft Visual Studio® 2008 Professional Edition, the include path and library path to the ATI Stream SDK v2 must be configured in the project. Compilation steps are:

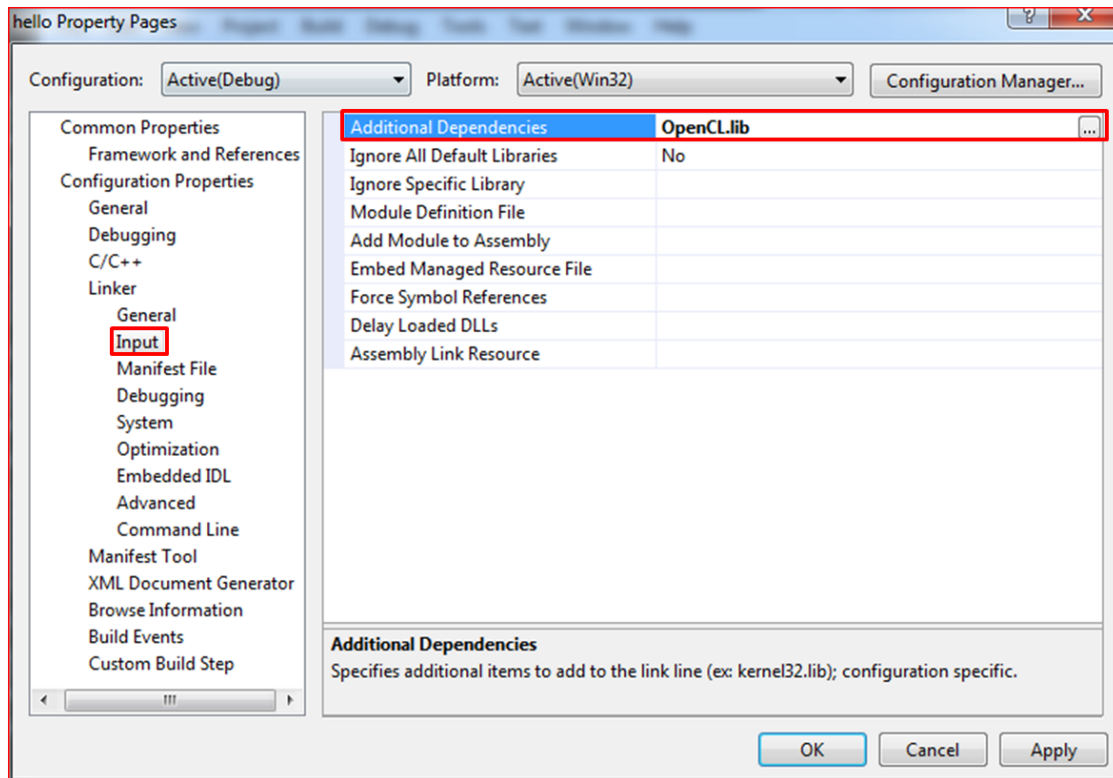
1. Open the project properties page in Visual Studio. On the C/C++ ► **General** page add `$ATISTREAMSDKROOT\include` to the **Additional Include Directories** list.



2. In the **Linker** ► **General** page, add `$ATISTREAMSDKROOT\lib\x86` (`$ATISTREAMSDKROOT\lib\x86_64` for 64-bit systems) to the **Additional Library Directories** list.



3. In the **Linker ▸ Input** page, add `OpenCL.lib` to the **Additional Dependencies** list.



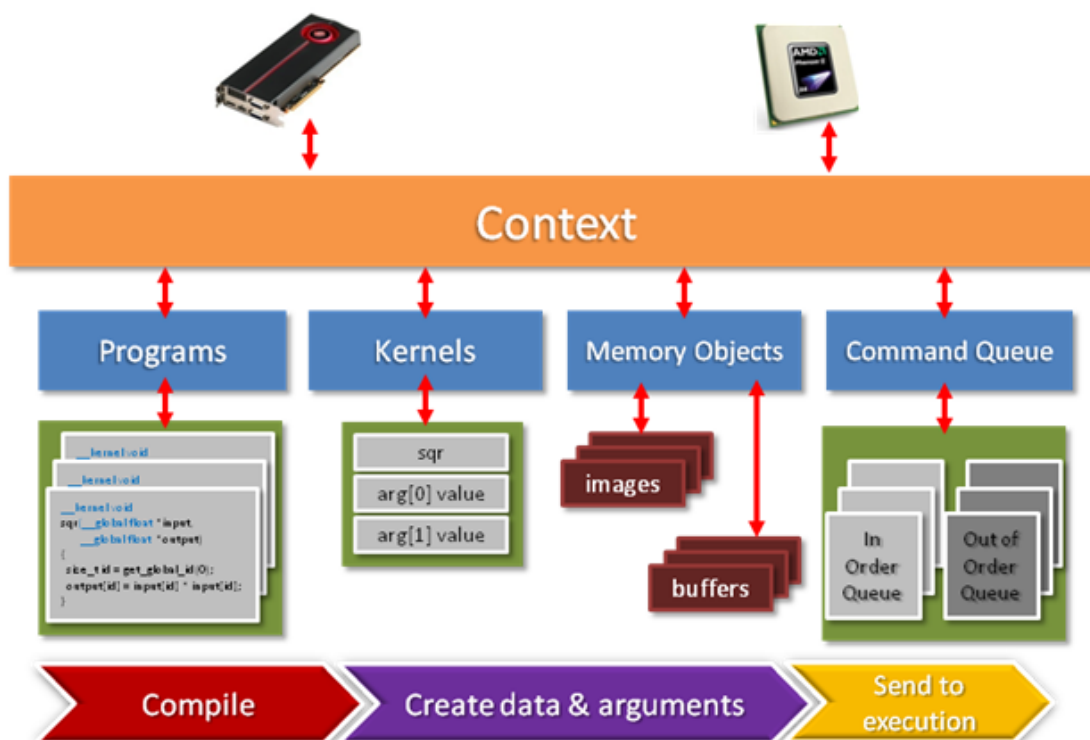
Chapter 5

OpenCL Programming in Detail

This chapter describes the structure of an OpenCL program and the relationship between devices, context, programs, kernels, memory objects, and command queues.

5.1 Executing an OpenCL Program

The OpenCL framework is divided into a platform layer API and runtime API. The platform API allows applications to query for OpenCL devices and manage them through a context. The runtime API makes use of the context to manage the execution of kernels on OpenCL devices. The basic steps involved in creating any OpenCL program are shown in the following figure.



To execute an OpenCL program:

1. Query the host system for OpenCL devices.

2. Create a context to associate the OpenCL devices.
3. Create programs that will run on one or more associated devices.
4. From the programs, select kernels to execute.
5. Create memory objects on the host or on the device.
6. Copy memory data to the device as needed.
7. Provide arguments for the kernels.
8. Submit the kernels to the command queue for execution.
9. Copy the results from the device to the host.

5.2 Resource Setup

The section describes how to setup OpenCL resources such as querying for platform information, device information, creating context and command queue.

5.2.1 Query for Platform Information and OpenCL Devices

The first step in any OpenCL application is to query for platform information and OpenCL devices. The function `clGetPlatformIDs` can be used to retrieve a list of platforms available, as shown in the following code:

```
cl_platform_id platforms;  
cl_uint num_platforms;  
  
// query for 1 available platform  
cl_int err = clGetPlatformIDs(  
    1,                      // the number of entries that can added to platforms  
    &platforms,              // list of OpenCL found  
    &num_platforms);         // the number of OpenCL platforms found
```

`clGetPlatformIDs` return values are:

- **CL_INVALID_VALUE** — Platforms and `num_platforms` is NULL or the number of entries is 0.
- **CL_SUCCESS** — The function executed successfully.

With the `platform_id` determined, the function `clGetPlatformInfo()` can be used to get specific information about the OpenCL platform, including `platform_profile`, `platform_version`, `platform_name`, `platform_vendor`, and `platform_extensions`.

Figure 5–1 Typical OpenCL Platform Info Retrieved from *clGetPlatformInfo()*

```

Number of platforms:      1
Platform Profile:         FULL_PROFILE
Platform Version:         OpenCL 1.0 ATI-Stream-v2.0.1
Platform Name:            ATI Stream
Platform Vendor:          Advanced Micro Devices, Inc.
Platform Extensions:      cl_khr_icd

```

5.2.2 Query for OpenCL Device

The function `clGetDeviceIDs()` can be used to search for compute devices in the system. The following code queries for a single GPU compute device:

```

cl_device_id device_id;
cl_uint num_of_devices;
cl_int err;
err = clGetDeviceIDs(
    platform_id,          // the platform_id retrieved from clGetPlatformIDs
    CL_DEVICE_TYPE_GPU,   // the device type to search for
    1,                   // the number of id add to device_id list
    &device_id,           // the list of device ids
    &num_of_devices        // the number of compute devices found
);

```

`clGetDeviceIDs()` return values are:

- **CL_INVALID_PLATFORM** — Platform is not valid.
- **CL_INVALID_DEVICE_TYPE** — The device is not a valid value.
- **CL_INVALID_VALUE** — `num_of_devices` and `devices` are NULL.
- **CL_DEVICE_NOT_FOUND** — No matching OpenCL of `device_type` was found.
- **CL_SUCCESS** — The function executed successfully.

The example above shows how to query for one GPU device on which the code will be executed. If the host system physically contains two GPUs and it is determined that the program will execute more efficiently on the two GPUs, simply ask for two device IDs instead. We can restrict the query to different device types by passing the appropriate parameter to the `clGetDeviceIDs()` function. The valid values for `device_type` are shown in the table below. For example, to query for CPU devices, pass `CL_DEVICE_TYPE_CPU` as the `device_type` argument.

Table 5–1 List of device types supported by *clGetDeviceIDs()*

<i>cl_device_type</i>	Description
CL_DEVICE_TYPE_CPU	An OpenCL device that is host processor. The host processor runs the OpenCL implementation and is a single or multi-core CPU.
CL_DEVICE_TYPE_GPU	An OpenCL device that is a GPU. The same device that can be also used to accelerate a 3D API such as OpenGL® or DirectX®.
CL_DEVICE_TYPE_ACCELERATOR	Dedicated OpenCL accelerators (such as IBM CELL Blade). These devices communicate with the host processor using a peripheral interconnect such as PCI or PCIe.
CL_DEVICE_TYPE_DEFAULT	The default OpenCL device in the system.
CL_DEVICE_TYPE_ALL	All OpenCL devices available in the system.

Similarly, specific capabilities can be retrieved on the compute device using the *device_id* returned from the query using the `clGetDeviceInfo()` function. Refer to the OpenCL Specification for a full list of capabilities that can be retrieved with the `clGetDeviceInfo()` function. The figure below shows capabilities of the ATI Radeon HD 4770 GPU using `clGetDeviceInfo()`.

Figure 5–2 Device Capabilities of the ATI Radeon HD 4770 GPU

Device Type:	CL_DEVICE_TYPE_GPU
Device ID:	4098
Max compute units:	8
Max work items dimensions:	3
Max work items[0]:	256
Max work items[1]:	256
Max work items[2]:	256
Max work group size:	256
Preferred vector width char:	16
Preferred vector width short:	8
Preferred vector width int:	4
Preferred vector width long:	2
Preferred vector width float:	4
Preferred vector width double:	0
Max clock frequency:	750Mhz
Address bits:	32
Max memeory allocation:	134217728
Image support:	No
Max size of kernel argument:	1024
Alignment (bits) of base address:	4096
Minimum alignment (bytes) for any datatype:	128
Single precision floating point capability	
Denorms:	No
Quiet NaNs:	Yes
Round to nearest even:	Yes
Round to zero:	No
Round to +ve and infinity:	No
IEEE754-2008 fused multiply-add:	No
Cache type:	None
Cache line size:	0
Cache size:	0
Global memory size:	134217728
Constant buffer size:	65536
Max number of constant args:	8
Local memory type:	Global
Local memory size:	16384
Profiling timer resolution:	1
Device endianness:	Little
Available:	Yes
Compiler available:	Yes
Execution capabilities:	
Execute OpenCL kernels:	Yes
Execute native function:	No
Queue properties:	
Out-of-Order:	No
Profiling :	Yes
Platform ID:	0xb7e06488
Name:	ATI RV770
Vendor:	Advanced Micro Devices, Inc.
Driver version:	CAL 1.4.519
Profile:	FULL_PROFILE
Version:	OpenCL 1.0 ATI-Stream-v2.0.1
Extensions:	cl_khr_icd

Note: In early releases of the ATI Stream SDK v2, the `clGetDeviceIDs()` function allowed `platform_id` to be NULL. This is no longer supported.

5.2.3 Creating a Context

Once the compute devices and their corresponding `device_id(s)` have been identified, the `device_id(s)` can be associated with a context. The context is used by the OpenCL runtime API to manage command queues, program objects, kernel objects, and the sharing of memory objects for the compute devices associated with the context. `clCreateContext()` is used to create a context, as shown in the following code:

```

cl_context context;

// context properties list - must be terminated with 0
properties[0]= CL_CONTEXT_PLATFORM; // specifies the platform to use
properties[1]= (cl_context_properties) platform_id;
properties[2]= 0;

context = clCreateContext(
    properties,      // list of context properties
    1,              // num of devices in the device_id list
    &device_id,      // the device id list
    NULL,           // pointer to the error callback function (if required)
    NULL,           // the argument data to pass to the callback function
    &err            // the return code
);

```

The context properties list is made up of a property name followed immediately by a property value. The list must be terminated with a 0 (zero). Supported properties are:

cl_context_properties_enum	Property Value	Description
CL_CONTEXT_PLATFORM	cl_platform_id	Specifies the platform to use

OpenCL allows an optional error callback function to be registered by the application to report on errors that occur within the context during runtime. OpenCL may call the error callback function asynchronously. It is, therefore, the application's responsibility to ensure the callback function is thread-safe.

If the context is created successfully, `clCreateContext()` returns a non-zero context with an error return code value of `CL_SUCCESS`. Otherwise, a `NULL` value is returned and one of the following error codes:

- **CL_INVALID_PLATFORM** — The property list is `NULL` or the platform value is not valid.
- **CL_INVALID_VALUE** — Either:
 - The property name in the properties list is not valid.
 - The number of devices is 0.
 - The `device_id` list is null.
 - The device in the `device_id` list is invalid or not associated with the platform.
- **CL_DEVICE_NOT_AVAILABLE** — The device in the `device_id` list is currently unavailable.
- **CL_OUT_OF_HOST_MEMORY** — The host is unable to allocate OpenCL resources.

5.2.4 Creating the Command Queue

When the context is established, command queues can be created that allow commands to be sent to the compute devices associated with the context. Commands are placed into the command queue in order. `clCreateCommandQueue()` is used to create a command queue, as shown in the following code:

```
cl_command_queue command_queue;

command_queue = clCreateCommandQueue(
    context,    // a valid context
    device_id, // a valid device associated with the context
    0,         // properties for the queue - not used here
    &err       // the return code
);
```

The context that is passed in must be a valid OpenCL context. If the context has multiple compute devices associated with it, you must choose which device to use by specifying the `device_id`. Thus, if the context contains a CPU and a GPU device type, two separate command queues must be created and work sent to the separate queues.

The third parameter, the command queue properties list, is a bit-field:

Command Queue Properties	Description
<code>CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE</code>	If set, commands in the command queue are executed out of order. Otherwise, commands are executed in order.
<code>CL_QUEUE_PROFILING_ENABLE</code>	If set, profiling of commands in the command queue is enabled. Otherwise, profiling is disabled.

If the command queue is successfully created, `clCreateCommandQueue()` returns a non-zero command queue with an error return code value of `CL_SUCCESS`. Otherwise, a NULL value is returned with one of the following error codes:

- **CL_INVALID_CONTEXT** — The context is not valid.
- **CL_INVALID_DEVICE** — Either the device is not valid or it is not associated with the context.
- **CL_INVALID_VALUE** — The properties list is not valid.
- **CL_INVALID_QUEUE_PROPERTIES** — The device does not support the properties specified in the properties list.
- **CL_OUT_OF_HOST_MEMORY** — The host is unable to allocate OpenCL resources.

5.3 Kernel Programming and Compiling

The previous section described the steps in setting up the resources before any work can be performed on compute devices. In this section, OpenCL program and kernel

objects are described, along with the steps involved in creating and compiling them for execution.

5.3.1 Creating the Program Object

An OpenCL program is made up of a collection of kernel functions. Kernel functions are written using the OpenCL C language and is compiled by the OpenCL runtime for execution on a particular compute device. The kernel function in a program is identified by the qualifier `__kernel` in the program source. The program source may also contain helper functions and data constants used by the `__kernel` functions.

A program object in OpenCL encapsulates the program sources or a binary file, the latest successfully-built program executable, the list of devices for which the executable is built, along with any build options and a build log. The program object can be created either online or offline. To create a program object online (or from source code) the `clCreateProgramWithSource()` function is used as shown in the following example:

```
const char *ProgramSource =
    "__kernel void hello(__global float *input, __global float *output)\n"\
    "{\n"\
    "    size_t id = get_global_id(0);\n"\
    "    output[id] = input[id] * input[id];\n"\
    "}\n";

cl_program program;
program = clCreateProgramWithSource(
    context,          // a valid context
    1,                // the number strings in the next parameter
    (const char **) &ProgramSource, // the array of strings
    NULL,             // the length of each string or can be NULL terminated
    &err               // the error return code
);
```

In the example, the program source is included as an array of strings. If the program source is in a separate file, you must read the source file into a string and pass it to the `clCreateProgramWithSource()` function. If multiple source files are to be read, an array of NULL-terminated strings are used and the `clCreateProgramWithSource()` function is informed about how many strings are in the array. In the list above, the `ProgramSource` array contains only one NULL-terminated string.

If the program object is successfully created, `clCreateProgramWithSource()` returns a non-zero program object with an error return code value of `CL_SUCCESS`. Otherwise, a NULL value is returned with one of the following error return code values:

- **CL_INVALID_CONTEXT** — The context is not valid.
- **CL_INVALID_VALUE** — The string count is 0 (zero) or the string array contains a NULL string.
- **CL_OUT_OF_HOST_MEMORY** — The host is unable to allocate OpenCL resources.

At this point a program object is ready for compilation and linking before it can be executed on the compute device. With the program object created, the function

`clGetProgramInfo()` is used to query specific info regarding the program object. One of the parameters that can be extracted is the program binary reference. Using this reference brings a significant reduction in initialization time in subsequent executions because the program need not be recreated.

The function `clCreateProgramWithBinary()` can be used create a program object by referencing the program binary. For more information on `clGetProgramInfo()` and `clCreateProgramWithBinary()`, see the OpenCL language specification.

5.3.2 Building Program Executables

After the program object is created (either from source using `clCreateProgramWithSource()` or binary using `clCreateProgramWithBinary()`), the next step is to compile and link the program object. The program executable can be built for one or more devices that are encapsulated by the program object. The function `clBuildProgram()` is used to build the executable:

```
err = clBuildProgram(
    program, // a valid program object
    0,       // number of devices in the device list
    NULL,    // device list - NULL means for all devices
    NULL,    // a string of build options
    NULL,    // callback function when executable has been built
    NULL     // data arguments for the callback function
);
```

The `clBuildProgram()` function modifies the program object to include the executable, the build log, and build options. The build option string allows the passing of any additional options to the compiler such as preprocessor options, optimization options, math intrinsic options, and other miscellaneous compiler options. The following example defines `GPU_DUAL_ENABLED` and disables all optimizations:

```
char * buildoptions = "-DGPU_DUAL_ENABLED -cl-opt-disable "
```

For a complete list of supported options, see the *Build Options* section of the OpenCL language specification.

`clBuildProgram()` returns `CL_SUCCESS` if the compilation is successful. Otherwise, the following are common error codes that may be returned by `clBuildProgram()`:

- **CL_INVALID_VALUE** — The number of devices is greater than zero, but the device list is empty.
- **CL_INVALID_VALUE** — The callback function is NULL, but the data argument list is not NULL.
- **CL_INVALID_DEVICE** — The device list does not match the devices associated in the program object.
- **CL_INVALID_BUILD_OPTIONS** — The build options string contains invalid options.
- **CL_OUT_OF_HOST_MEMORY** — The host is unable to allocate OpenCL resources.

For a complete list of return codes for `clBuildProgram()`, see the OpenCL specification.

When compiling source code for any language, compilation errors can occur. In OpenCL, the program object contains the build log. To access the build log, the OpenCL API provides the `clGetProgramBuildInfo()` function to retrieve the latest compilation results embedded in the program object:

```
if (clBuildProgram(program, 0, NULL, buildoptions, NULL, NULL) != CL_SUCCESS)
{
    printf("Error building program\n");

    char buffer[4096];
    size_t length;

    clGetProgramBuildInfo(
        program,                // valid program object
        device_id,              // valid device_id that executable was built
        CL_PROGRAM_BUILD_LOG,   // indicate to retrieve build log
        sizeof(buffer),         // size of the buffer to write log to
        buffer,                 // the actual buffer to write log to
        &length                  // the actual size in bytes of data copied to buffer
    );

    printf("%s\n", buffer);
    exit(1);
}
```

Figure 5–3 Sample Build Log Output

```
platform 1
platform name: ATI Stream
kernel void square(const __global float *input0,
                  const __global float *input1,
                  __global float * out)
{
    const int Width = get_global_size(0);
    const size_t xid = get_global_id(0);
    const size_t yid = get_global_id(1);

    const int idx= id*Width + xid;
    out[idx]=input0[idx]+input1[idx];
}

-- /tmp/OCLZenMa8.cl(9): error: identifier "id" is undefined
    const int idx= id*Width + xid;
                  ^
1 error detected in the compilation of "/tmp/OCLZenMa8.cl".
Failed to get work group info
```

5.3.3 Creating Kernel Objects

The kernel object is created after the executable has been successfully built in the program object. As mentioned earlier, kernel functions are declared with the qualifier `__kernel` in the program object. A kernel object is an encapsulation of the specify `__kernel` function along with the arguments that are associated with the `__kernel`

function when executed. The kernel object is eventually sent to the command queue for execution. The function `clCreateKernel()` is used to create kernel objects:

```
cl_kernel kernel;
kernel = clCreateKernel(
    program, // a valid program object that has been successfully built
    "hello", // the name of the kernel declared with __kernel
    &err // error return code
);
```

If the kernel object is created successfully, `clCreateKernel()` returns a non-zero kernel object with an error return code value of `CL_SUCCESS`. Otherwise, a `NULL` value is returned and the error return code will have one of the following values set:

- **CL_INVALID_PROGRAM** — The program is not a valid program object.
- **CL_INVALID_PROGRAM_EXECUTABLE** — The program does not contain a successfully built executable.
- **CL_INVALID_KERNEL_NAME** — The kernel name is not found in the program object.
- **CL_INVALID_VALUE** — The kernel name is `NULL`.
- **CL_OUT_OF_HOST_MEMORY** — The host is unable to allocate OpenCL resources.

Before the kernel object is submitted to the command queue for execution, input or output buffers must be provided for any arguments required by the `__kernel` function. If the arguments use memory objects such as a buffer or image, the memory objects must be created first and the data must be explicitly copied into the memory object. See [5.5 Memory Objects](#) for instructions on creating memory object data.

5.3.4 Setting Kernel Arguments

When the required memory objects have been successfully created, kernel arguments can be set using the `clSetKernelArg()` function:

```
err = clSetKernelArg(
    kernel, // valid kernel object
    0, // the specific argument index of a kernel
    sizeof(cl_mem), // the size of the argument data
    &input_data // a pointer of data used as the argument
);
```

The *argument index* refers to the specific argument position of the `__kernel` definition that must be set. The position can be 0 (zero) or greater. The following example shows the declaration of the `__kernel` function:

```
__kernel void hello(__global float *input,
                   __global float *output)
```

In this example, the input argument is index 0, and the output argument is index 1.

The last two arguments of `clSetKernelArg()` specify the size of the argument data and the pointer to the actual data.

Note: If a `__kernel` function argument is declared to be a pointer of a built-in or user defined type with the `__global` or `__constant` qualifier, a buffer memory object must be used. If the argument is an `image2d_t` or `image3d_t`, the memory object specified as the argument in `clSetKernelArg` must be a 2D image or 3D image object, respectively. Refer to the OpenCL Specification for more information on `clSetKernelArg()`.

If the function is executed successfully, `clSetKernelArg()` returns `CL_SUCCESS`. Otherwise, the follow are some common error codes returned by `clSetKernelArg()`:

- **CL_INVALID_KERNEL** — The kernel is not a valid kernel object.
- **CL_INVALID_ARG_INDEX** — The index value is not a valid argument index.
- **CL_INVALID_MEMORY_OBJECT** — The argument is declared as a memory object, but the argument value is not a valid memory object.
- **CL_INVALID_ARG_SIZE** — The argument size does not match the size of the data type of the declared argument.

OpenCL provides the helper function `clGetKernelInfo()` to allow the application to query specific information about a kernel object:

```
cl_int clGetKernelInfo (
    cl_kernel kernel,           // valid kernel object
    cl_kernel_info param_name, // the information to query (see below)
    size_t param_value_size,   // size of the mem buffer to hold the result
    void *param_value,         // pointer to mem where result of query is returned
    size_t *param_value_size_ret // actual size of data copied to param_value
)
```

The supported kernel information that can be queried is shown in the following table:

cl_kernel_info	Return Type	Data Returned to param_value
CL_KERNEL_FUNCTION_NAME	char[]	Name of the kernel function
CL_KERNEL_NUM_ARGS	cl_uint	Number of arguments in the kernel function
CL_KERNEL_REFERENCE_COUNT	cl_uint	Number of kernel reference count
CL_KERNEL_CONTEXT	cl_context	The context associated with the kernel
CL_KERNEL_PROGRAM	cl_program	The program object associated with the kernel

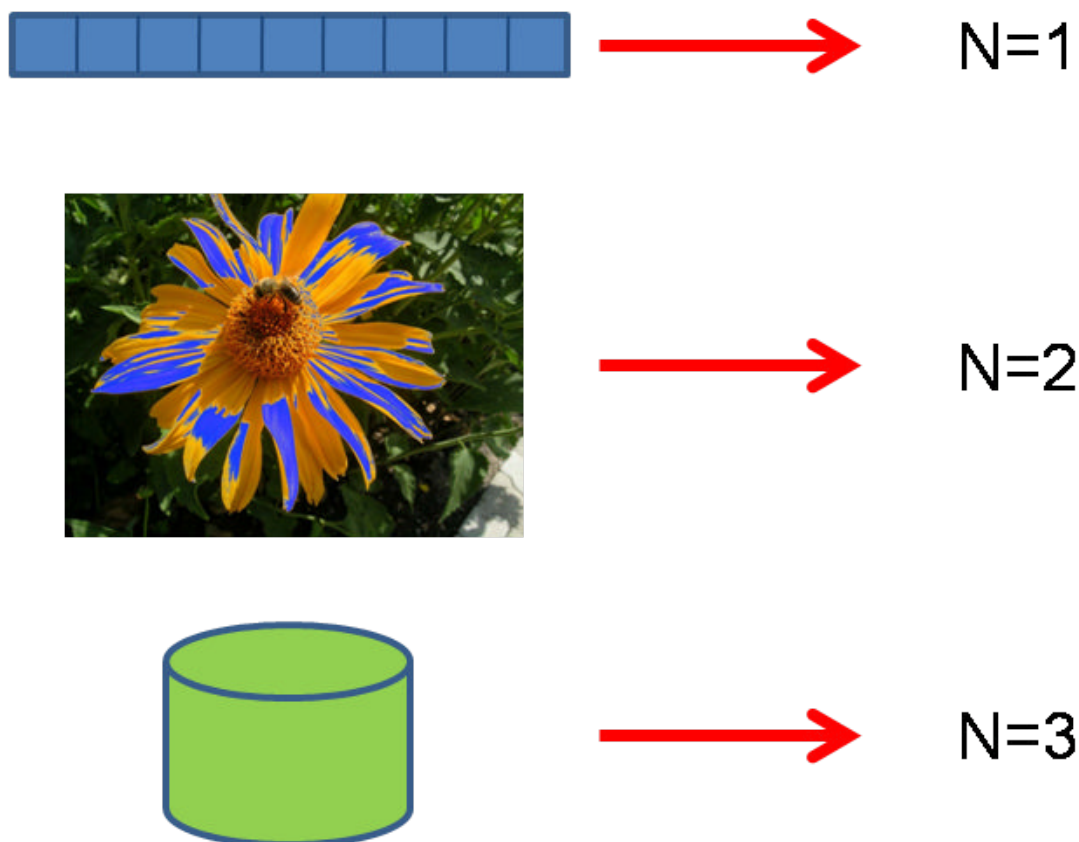
5.4 Program Execution

5.4.1 Determining the Problem Space

As mentioned in [3.3.2 The Execution Model](#), each kernel execution in OpenCL is called a work-item. OpenCL exploits parallel computation of the compute devices by having instances of the kernel execute on different portions of the N -dimensional problem space. In addition, each work-item is executed only with its assigned data. Thus, it is important specify to OpenCL how many work-items are needed to process all data.

Before the work-items total can be determined, the N -dimension to be used to represent the data must be determined. For example, a linear array of data would be a one-dimension problem space, while an image would be a two-dimensional problem space, and spatial data, such as a 3D object, would be a three-dimensional problem space.

Figure 5–4 Representation of Data Set in N -dimensional Space



When the dimension space is determined, the total work-items (also called the global work size) can be calculated. For a one-dimensional data set, the global work size is simply the size of the data. For two-dimensional data—such as an image—the global

work size is calculated by the data's width*height pixel count. For a three-dimensional object with positional data, the global data size is $x*y*z$.

OpenCL allows work-items to be combined into work-groups, and all work-items within a work-group are executed on the same compute unit on the same compute device. When a work-group size is specified, OpenCL divides the global work size among the compute units on the device. In addition, work-items within a work-group execute synchronously and are able to share memory. The total number of work-items in a work-group is known as the local work size. Group sizes cannot be assigned arbitrarily; the provided OpenCL function `clGetKernelWorkGroupInfo()` must be used to query the group size info of a device. For more information on `clGetKernelWorkGroupInfo()`, consult the OpenCL specification.

When the work-items for each dimension and the group size (local work size) is determined, the kernel can be sent to the command queue for execution. To enqueue the kernel to execute on a device, use the function `clEnqueueNDRangeKernel()`:

```
err = clEnqueueNDRangeKernel(
    command_queue, // valid command queue
    kernel,        // valid kernel object
    1,             // the work problem dimensions
    NULL,          // reserved for future revision - must be NULL
    &global,        // work-items for each dimension
    NULL,          // work-group size for each dimension
    0,             // number of event in the event list
    NULL,          // list of events that needs to complete before this executes
    NULL           // event object to return on completion
);
```

The work problem dimensions (the third argument) must be 1, 2, or 3. The fifth argument specifies the work-item size for each dimension. If, for example an image of 512x512 pixels is to be processed, an array must be provided that points to the number of work-item for each dimension:

```
size_t global[2]={512,512};
```

The sixth argument describes the number of work-items that make up a work-group. If, for example, 64 work-items were grouped into an 8x8 work-group, the work-group size for each dimension would be specified as the following array:

```
size_t local[2]={8,8};
```

The total work-items in each work group must be less than the maximum work group size retrieved using the function `clGetKernelWorkGroupInfo()`. If data does not need to be shared among work-items, a NULL value may be specified and the global work-items are broken into appropriate work-group instances by the OpenCL runtime.

The seventh and eighth arguments control the sequence of execution of the kernel command. As mentioned earlier, if the command queue properties are set to allow out-of-order execution during the `clCreateCommandQueue()` process, a list of events must be provided, along with the number of events in the list that need to complete before this particular command can be executed. If the event list is empty, then the number of events must be 0 (zero).

The last argument allows the kernel instance to generate an event on completion. This lets other kernel commands to wait for completion of this command as well as allowing querying of the kernel execution. See [5.6 Synchronization](#) section for more information on this event.

If the function executes successfully, `clEnqueueNDRangeKernel()` returns `CL_SUCCESS`. Otherwise, the following common error codes may be returned:

- **CL_INVALID_PROGRAM_EXECUTABLE** — No executable has been built in the program object for the device associated with the command queue.
- **CL_INVALID_COMMAND_QUEUE** — The command queue is not valid.
- **CL_INVALID_KERNEL** — The kernel object is not valid.
- **CL_INVALID_CONTEXT** — The command queue and kernel are not associated with the same context.
- **CL_INVALID_KERNEL_ARGS** — Kernel arguments have not been set.
- **CL_INVALID_WORK_DIMENSION** — The dimension is not between 1 and 3.
- **CL_INVALID_GLOBAL_WORK_SIZE** — The global work size is NULL or exceeds the range supported by the compute device.
- **CL_INVALID_WORK_GROUP_SIZE** — The local work size is not evenly divisible with the global work size or the value specified exceeds the range supported by the compute device.
- **CL_INVALID_GLOBAL_OFFSET** — The reserved global offset parameter is not set to NULL.
- **CL_INVALID_EVENT_WAIT_LIST** — The events list is empty (NULL) but the number of events arguments is greater than 0; or number of events is 0 but the event list is not NULL; or the events list contains invalid event objects.
- **CL_OUT_OF_HOST_MEMORY** — The host is unable to allocate OpenCL resources.

Refer to the OpenCL Specification for a more comprehensive list of error return codes.

When the kernel command is enqueued for execution, additional kernels can be sent to the command queue, or the results can be copied back from the device to host memory. See [5.5 Memory Objects](#) for a description of how to read data from memory objects.

5.4.2 Cleaning Up

When execution is complete, all OpenCL platform and runtime API resources, including memory objects, kernel, program, command queue, etc., should be released:

```
clReleaseMemObject(input);
clReleaseMemObject(output);
clReleaseProgram(program);
clReleaseKernel(kernel);
clReleaseCommandQueue(command_queue);
clReleaseContext(context);
```

All `clRelease<object>()` functions decrement and return a reference count to the object. If all resources are correctly released, the reference count should be zero. Otherwise, the returned reference counts can be used to track down memory leaks.

5.5 Memory Objects

The OpenCL framework provides a way to package data into a memory object. By using a memory object, OpenCL allows easy packaging of all data and easy transfer to the compute device memory so that the kernel function executing on the device has local access to the data.

Using a memory object minimizes memory transfers from the host and device as the kernel processes data. OpenCL memory objects are categorized into two types: buffer objects and image objects. The buffer object is used to store one-dimensional data such as an *int*, *float*, *vector*, or a user-defined structure. The image object is used to store two- or three-dimensional data such as textures or images.

5.5.1 Creating Buffer Objects

To use the `clCreateBuffer()` function to create a buffer object:

```
cl_mem input;
input = clCreateBuffer(
    context,          // a valid context
    CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, // bit-field flag to specify
                                                // the usage of memory
    sizeof(float) * DATA_SIZE, // size in bytes of the buffer to allocated
    inputsrc, // pointer to buffer data to be copied from host
    &err      // returned error code
);
```

The bit-field flag is used to specify allocation and define how the memory is used. The following table describes the supported values for the flag.

Table 5–2 Supported CL_MEM Flags

CL_MEM Flags	Description
CL_MEM_READ_WRITE	Kernel can read and write to the memory object.
CL_MEM_WRITE_ONLY	Kernel can write to memory object. Read from the memory object is undefined.
CL_MEM_READ_ONLY	Kernel can only read from the memory object. Write from the memory object is undefined.
CL_MEM_USE_HOST_PTR	Specifies to OpenCL implementation to use memory reference by host_ptr (4th arg) as storage for memory object.
CL_MEM_COPY_HOST_PTR	Specifies to OpenCL to allocate the memory and copy data pointed by host_ptr (4th arg) to the memory object.
CL_MEM_ALLOC_HOST_PTR	Specifies to OpenCL to allocate memory from host accessible memory.

If the buffer object is created successfully, `clCreateBuffer()` returns a non-zero buffer object with an error return code value of `CL_SUCCESS`. Otherwise, a NULL value is returned with one of the following error codes:

- **CL_INVALID_CONTEXT** — The context is not valid.
- **CL_INVALID_VALUE** — The value in `cl_mem_flag` is not valid (see table above for supported flags).
- **CL_INVALID_BUFFER_SIZE** — The buffer size is 0 (zero) or exceeds the range supported by the compute devices associated with the context.
- **CL_INVALID_HOST_PTR** — Either: The `host_ptr` is NULL, but `CL_MEM_USE_HOST_PTR`, `CL_MEM_COPY_HOST_PTR`, and `CL_MEM_ALLOC_HOST_PTR` are set; or `host_ptr` is not NULL, but the `CL_MEM_USE_HOST_PTR`, `CL_MEM_COPY_HOST_PTR`, and `CL_MEM_ALLOC_HOST_PTR` are not set.
- **CL_INVALID_OBJECT_ALLOCATION_FAILURE** — Unable to allocate memory for the memory object.
- **CL_OUT_OF_HOST_MEMORY** — The host is unable to allocate OpenCL resources.

5.5.2 Reading, Writing, and Copying Buffer Objects

After the buffer memory object is created, commands can be enqueued to write data to a buffer object from host memory or read data from a buffer object to host memory. OpenCL provides the `clEnqueueWriteBuffer()` and `clEnqueueReadBuffer()` functions for these purposes.

```
err = clEnqueueReadBuffer(
    command_queue,          // valid command queue
    output,                  // memory buffer to read from
    CL_TRUE,                 // indicate blocking read
    0,                       // the offset in the buffer object to read from
    sizeof(float) * DATA_SIZE, // size in bytes of data being read
    results,                 // pointer to buffer in host mem to store read data
    0,                       // number of event in the event list
    NULL,                    // list of events that needs to complete before this executes
    NULL                     // event object to return on completion
);
```

`clEnqueueReadBuffer()` enqueues a command to read data from a buffer object into host memory. This function is typically used to read the results from the kernel execution back to the host program. The `CL_TRUE` parameter indicates that the read command is blocked until the buffer data is completely copied into host memory. If the blocking parameter is set to `CL_FALSE`, the read command is returned as soon as it's enqueued. Another way to determine when the read command has finished copying data to the host memory is to use an event object that is returned by the read command and can be used to check on the execution status.

```
err = clEnqueueWriteBuffer(
    command_queue, // valid command queue
    input,          // memory buffer object to write to
    CL_TRUE,        // indicate blocking write
    0,              // the offset in the buffer object to write to
    sizeof(float) * DATA_SIZE, // size in bytes of data to write
    host_ptr,       // pointer to buffer in host mem to read data from
    0,              // number of event in the event list
    NULL,           // list of events that needs to complete before this executes
    NULL            // event object to return on completion
);
```

`clEnqueueWriteBuffer()` enqueues a command to write data from host memory to a buffer object. This function is typically used to provide data for the kernel for processing. Similar to `clEnqueueReadBuffer()`, the block flag can be set to `CL_TRUE` or `CL_FALSE` to indicate blocking or non-blocking for the write command.

If the functions execute successfully, `clEnqueueReadBuffer()` and `clEnqueueWriteBuffer()` return `CL_SUCCESS`. Otherwise, one of the following common error codes may be returned:

- **CL_INVALID_COMMAND_QUEUE** — The command queue is not valid
- **CL_INVALID_CONTEXT** — The command queue buffer object is not associated with the same context.
- **CL_INVALID_VALUE** — The region being read/write specified by the offset is out of bounds or the host pointer is NULL.
- **CL_INVALID_EVENT_WAIT_LIST** — Either: The events list is empty (NULL), but the number of events argument is greater than 0; or number of events is 0, but the event list is not NULL; or ;the events list contains invalid event objects.
- **CL_OUT_OF_HOST_MEMORY** — The host is unable to allocate OpenCL resources.

The function `clEnqueueCopyBuffer()` can be used to copy data from one memory buffer object to another. For more information on the `clEnqueueCopyBuffer()` function, see the OpenCL specification.

5.5.3 Retaining and Releasing Buffer Objects

Whenever a memory buffer object is created the reference counter for that object is set to 1. The reference counter is used to track how many objects have references to a particular buffer object. At any time, other object can retain a reference to the buffer object by calling `clRetainMemObject()`. This increases the buffer object's reference count by 1. When the memory buffer object is no longer needed, the `clReleaseMemObj()` function can be used to decrement the reference counter by 1. When the reference counter becomes zero, the memory object is freed.

`clRetainMemObject()` and `clReleaseMemObj()` return `CL_SUCCESS` if the function executes successfully. Otherwise, `CL_INVALID_MEM_OBJECT` is returned if the buffer object is not a valid memory object.

5.5.4 Creating Image Objects

OpenCL has built-in support for representing image data in a wide range of formats. This is useful when creating kernel functions that must perform processing on image data. The function `clCreateImage2D()` is used to create a two-dimensional image object.

```
image2d = clCreateImage2D(
    context,          // valid context
    flags,            // bit-field flag to specify usage of memory
    image_format,     // ptr to struct that specifies image format properties
    width,            // width of the image in pixels
    height,           // height of the image in pixels
    row_pitch,        // scan line row pitch in bytes
    host_ptr,         // pointer to image data to be copied from host
    &err              // error return code
);
```

Options such as read-only, write-only, read-write, etc., that are used by `clCreateBuffer()` can also be used by `clCreateImage2D()`. For details, see [Table 5-2 Supported CL_MEM Flags](#).

The row-pitch is the number of bytes necessary to represent one line of the image. Row-pitch is calculated as $\text{width} \times \text{size of each element in bytes}$. If the row-pitch is set to 0, the OpenCL implementation will perform the row-pitch calculation.

The image format is a structure that describes the properties such as number of channels and channel ordering in the image:

```
typedef struct _cl_image_format {
    cl_channel_order image_channel_order;
    cl_channel_type image_channel_data_type;
}
cl_image_format;
```

Supported `channel_data_order` settings are listed in the following table.

Table 5–3 Supported Channel Data Order

<code>channel_data_order</code>
CL_R or CL_A
CL_INTENSITY — Supported only if channel data type = <code>CL_UNORM_INT8</code> , <code>CL_UNORM_INT16</code> , <code>CL_SNORM_INT8</code> , <code>CL_SNORM_INT16</code> , <code>CL_HALF_FLOAT</code> , or <code>CL_FLOAT</code> .
CL_LUMINANCE — Supported only if channel data type = <code>CL_UNORM_INT8</code> , <code>CL_UNORM_INT16</code> , <code>CL_SNORM_INT8</code> , <code>CL_SNORM_INT16</code> , <code>CL_HALF_FLOAT</code> , or <code>CL_FLOAT</code> .
CL_RG or CL_RA
CL_RGB — Supported only if channel data type = <code>CL_UNORM_SHORT_565</code> , <code>CL_UNORM_SHORT_555</code> , or <code>CL_UNORM_INT_101010</code>
CL_RGBA
CL_ARGB or CL_BGRA — Supported if channel data type = <code>CL_UNORM_INT8</code> , <code>CL_SNORM_INT8</code> , <code>CL_SIGNED_INT8</code> , or <code>CL_UNSIGNED_INT8</code>

The `channel_data_type` specifies the size of the channel data type, as listed in the following table:

Table 5–4 Supported Channel Data Types

<code>channel_data_type</code>	Description
<code>CL_SNORM_INT8</code>	Each channel is a signed normalized 8-bit integer.
<code>CL_SNORM_INT16</code>	Each channel is a signed normalized 16-bit integer.
<code>CL_UNORM_INT8</code>	Each channel is a unsigned normalized 8-bit integer.
<code>CL_UNORM_INT16</code>	Each channel is a unsigned normalized 16-bit integer.
<code>CL_UNORM_SHORT_565</code>	3-channel RGB image packed into a single unsigned short of 5-6-5. Bits 15:11=R, 10:5=G, 4:0=B. The channel order must be set to <code>CL_RGB</code> .
<code>CL_UNORM_SHORT_555</code>	3-channel RGB image packed into a single unsigned short of 5-5-5. Bits 14:10=R, 9:5=G, 4:0=B. The channel order must be set to <code>CL_RGB</code> . The channel order must be set to <code>CL_RGB</code> .
<code>CL_UNORM_INT_101010</code>	3-channel RGB image packed into a single unsigned int of 10-10-10. Bits 31:30=undefined, 29:20=R, 19:10=G, 9:0=B. The channel order must be set to <code>CL_RGB</code> .
<code>CL_SIGNED_INT8</code>	Each channel is signed unnormalized 8-bit integer.
<code>CL_SIGNED_INT16</code>	Each channel is signed unnormalized 16-bit integer.
<code>CL_SIGNED_INT32</code>	Each channel is signed unnormalized 32-bit integer.
<code>CL_UNSIGNED_INT8</code>	Each channel is unsigned unnormalized 8-bit integer.

channel_data_type	Description
CL_UNSIGNED_INT16	Each channel is unsigned unnormalized 16-bit integer
CL_UNSIGNED_INT32	Each channel is unsigned unnormalized 32-bit integer
CL_HALF_FLOAT	Each channel is a 16-bit half-float
CL_FLOAT	Each channel is a single precision float

Example: For a floating point data channel with RGBA channel-ordering the `image_format` struct would be set as follows:

```
cl_image_format image_format;
image_format.image_channel_data_type = CL_FLOAT;
image_format.image_channel_order = CL_RGBA;
```

Similar to 2D images, a 3D image object is created using the `clCreateImage3D()` function. For a 3D object, the depth of the object must be provided in addition to the width and height. The `slice_pitch` size must also be provided in addition to `row_pitch` size (`slice_pitch` is the size of each 2D slice in the 3D image, in bytes). See the OpenCL Specification for more details on creating a 3D image object.

As with buffer objects, OpenCL also provides functions to enqueue commands to write 2D or 3D image data to host memory or read back 2D or 3D image objects to host memory.

```
err = clEnqueueReadImage (
    command_queue, // valid command queue
    image,         // valid image object to read from
    blocking_read, // blocking flag, CL_TRUE or CL_FALSE
    origin_offset, // (x,y,z) offset in pixels to read from z=0 for 2D image
    region,        // (width,height,depth) in pixels to read from, depth=1 for 2D image
    row_pitch,     // length of each row in bytes
    slice_pitch,   // size of each 2D slice in the 3D image in bytes, 0 for 2D image
    host_ptr,      // host memory pointer to store write image object data to
    num_events,    // number of events in events list
    event_list,    // list of events that needs to complete before this executes
    &event         // event object to return on completion
);

err = clEnqueueWriteImage (
    command_queue, // valid command queue
    image,         // valid image object to write to
    blocking_read, // blocking flag, CL_TRUE or CL_FALSE
    origin_offset, // (x,y,z) offset in pixels to write to z=0 for 2D image
    region,        // (width,height,depth) in pixels to write to, depth=1 for 2D image
    row_pitch,     // length of each row in bytes
    slice_pitch,   // size of each 2D slice in the 3D image in bytes, 0 for 2D image
    host_ptr,      // host memory pointer to store read data from
    num_events,    // number of events in events list
    event_list,    // list of events that needs to complete before this executes
    &event         // event object to return on completion
);
```

If the function executes successfully, `clEnqueueReadImage()` and `clEnqueueWriteImage()` return `CL_SUCCESS`. Otherwise, one of the following common error codes may be returned:

- **CL_INVALID_COMMAND_QUEUE** — The command queue is not valid.
- **CL_INVALID_CONTEXT** — The command queue and image object are not associated with the same context.
- **CL_INVALID_MEM_OBJECT** — The image object is not valid
- **CL_INVALID_VALUE** — The region being read/write specified by the `origin_offset` and region is out of bounds or the host pointer is NULL.
- **CL_INVALID_VALUE** — The image object is 2D and `origin_offset[2]` (y component) is not set to 0, or `region[2]` (depth component) is not set to 1.
- **CL_INVALID_EVENT_WAIT_LIST** — Either: The events list is empty (NULL), but the number of events argument is greater than 0; or number of events is 0, but the event list is not NULL; or the events list contains invalid event objects.
- **CL_INVALID_OPERATION** — The associated device does not support images.
- **CL_OUT_OF_HOST_MEMORY** — The host is unable to allocate OpenCL resources.

For a complete list of error codes, see the OpenCL specification.

5.5.5 Retaining and Releasing Image Objects

Image objects should be freed when no longer required by the application. They are retained and released using the same `clRetainMemObj()` and `clReleaseMemObj()` functions used by buffer objects. For details, see [5.5.3 Retaining and Releasing Buffer Objects](#).

5.6 Synchronization

When a kernel is queued, it may not execute immediately. Execution can be forced by using a blocking call. The `clEnqueueRead*`() and `clEnqueueWrite*`() functions contain a blocking flag that, when set to `CL_TRUE`, forces the function to block until read/write commands have completed. Using a blocking command forces the OpenCL runtime to flush all the commands in the queue by executing all kernels.

The recommended way to track the execution status of kernels in the command queue is to use events. Events allow the host application to work without blocking OpenCL calls. The host application can send tasks to the command queue and return later to check if the execution is done by querying the event status.

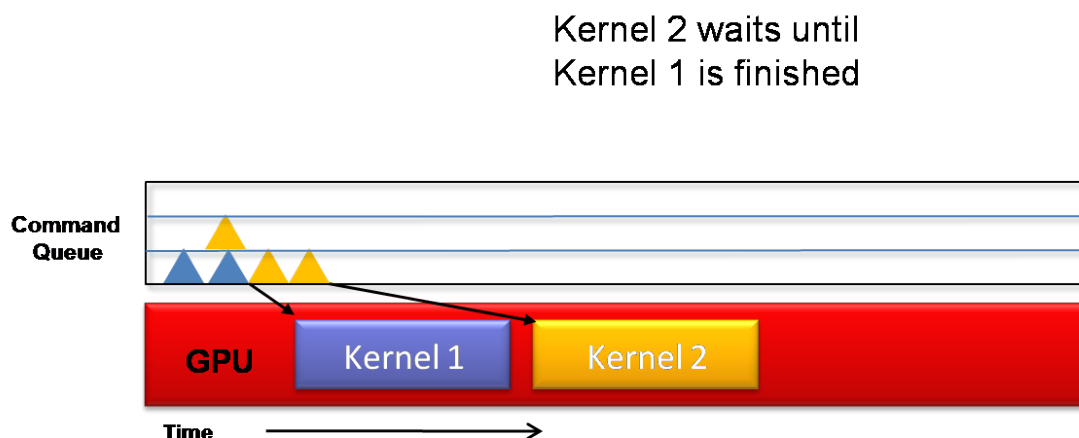
The queue can execute commands in-order or out-of-order, depending on how the command is created. An in-order queue behaves as expected as long the commands are enqueued from a single thread. If there are multiple devices—for example, a GPU and a CPU, each of which have their own queue—commands must be synchronized between the two queues using specific functions such as `clEnqueue*`. All `clEnqueue*` functions take three arguments: The number of events to wait on, a list of

events to wait on, and an event that the command creates that can be used by other commands to perform synchronization:

```
clEnqueue*(..., num_events, events_wait_list, event_return)
```

The following image illustrates a scenario with one OpenCL device and one command queue.

Figure 5-5 Kernel Execution with One Device and One Queue

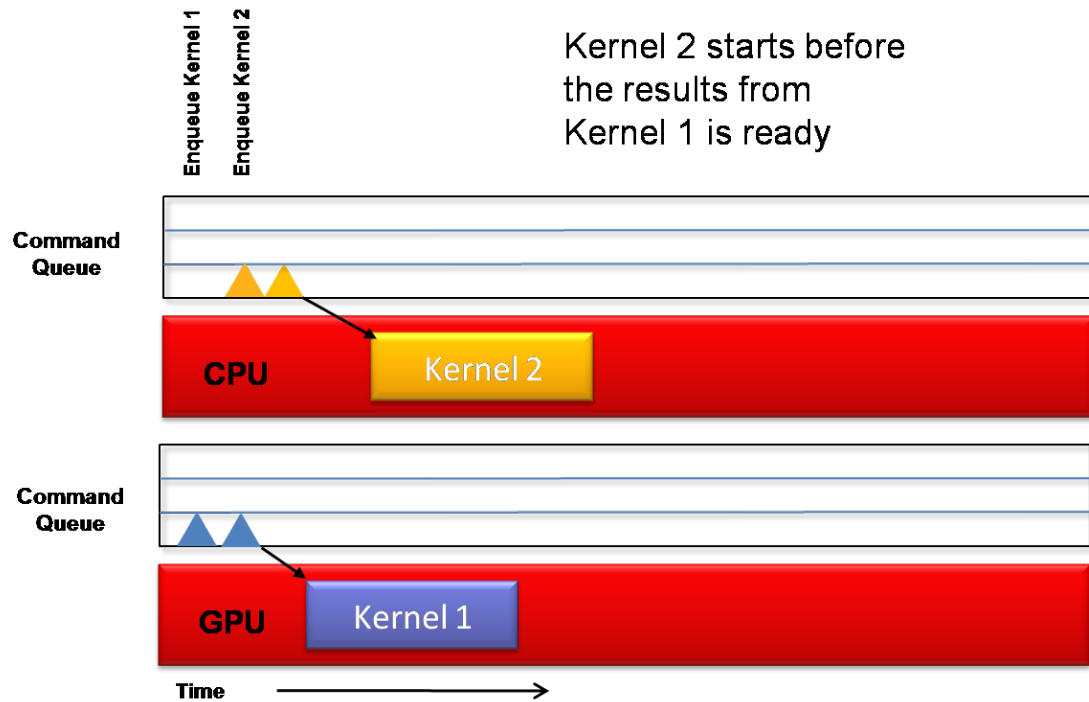


When two kernels must be executed, with the second kernel using the results of the first, the first kernel command enters the queue but may not execute immediately. Since the enqueue function is an asynchronous call and returns when it is called, the second kernel command is enqueued when the function returns. Thus, as a single in-order command queue, the second kernel command will not execute until the first command has finish executing.

The same result can be achieved when using multiple OpenCL devices, each having its own queue, and kernel execution must be synchronized between the two devices. If kernel 1 is running on a GPU, kernel 2 is running on a CPU, and the second kernel must use the results of the first, the second kernel will execute only after the first is finished—but again, only if events are used.

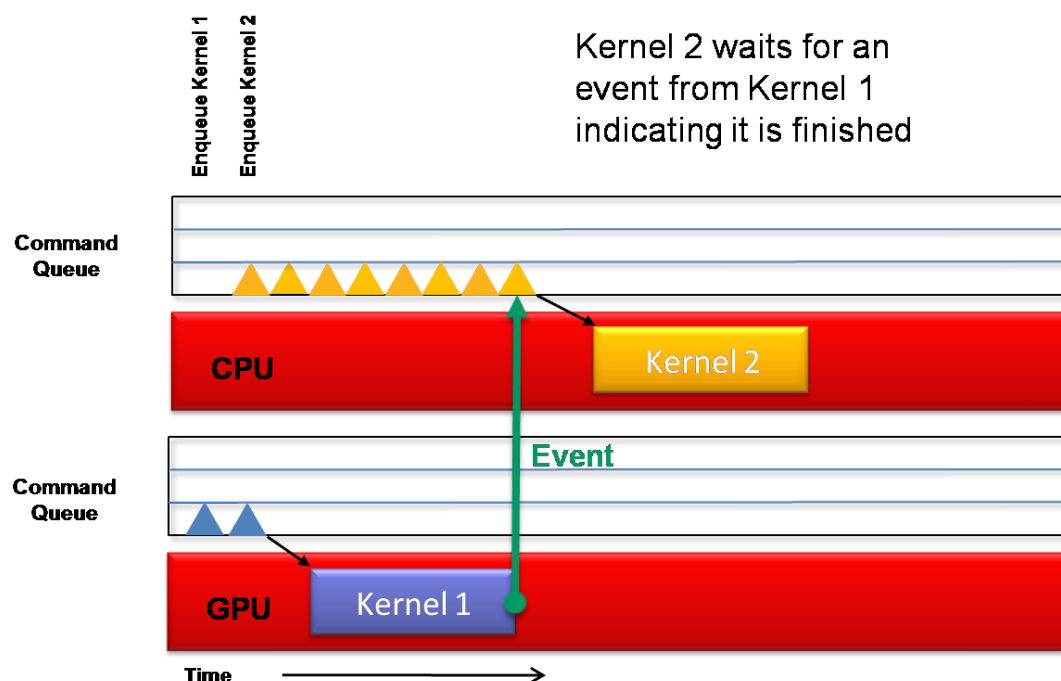
The following figure illustrates what can happen in the same two-devices, two-queue scenario, but without using events to synchronize execution. In this illustration, the first kernel is enqueued into the GPU queue but is not executed immediately. When it is executed, the OpenCL runtime may also detect that the CPU is free, and begin executing the second kernel before results from the first kernel are returned.

Figure 5–6 Two Devices with Two Queues, Unsynchronized



In the same two-devices, two-queue scenario, this time using events to synchronize execution, overlapping execution is avoided, as illustrated in the following figure:

Figure 5-7 Two Devices with Two Queues Synchronized Using Events



In addition to using events in the `clEnqueue` functions to manage how kernels are executed, host application events can also be managed, allowing the application to better manage resources for maximum optimization.

The function `clWaitForEvents(num_events, event_list)` causes the host application to block and wait until all commands identified by the event list are complete (that is, when the execution status of each command is `CL_COMPLETE`). The OpenCL runtime, rather than the host application, can also perform the block by using the `clEnqueueWaitForEvents(command_queue, num_events, event_list)` function to introduce a wait command into the queue to wait for specific events to complete before any future commands in the queue are executed.

An event marker can also be inserted into the command queue. This allows tracking of how quickly commands are moving through the queue. The function `clEnqueueMarker(command_queue, *event_return)` returns an event which can be used to queue a wait on the specified event.

Another useful synchronization function is `clGetEventInfo()`:

```
clGetEventInfo(
    event, // the event object to query
    param_name, // the cl_event_info
    param_size, // the size of the returned info
    param_value, // the value of the returned info
    actual_param_size_ret // actual size of info returned
)
```

This function allows the querying of an event object to determine what command the event is associated with and the execution status of the command (whether it's running or completed). The list of supported `param_name` types and the information returned by `clGetEventInfo()` is described in the following table.

Table 5–5 Supported List `param_name` and Return Values for `clGetEventInfo()`

<code>cl_event_info</code>	Return Type	Info returned in <code>parm_value</code>
<code>CL_EVENT_COMMAND_QUEUE</code>	<code>cl_command_queue</code>	Return the command queue associated with the event
<code>CL_EVENT_COMMAND_TYPE</code>	<code>cl_command_type</code>	Command type is one of the following: <code>CL_COMMAND_NDRANGE_KERNEL</code> <code>CL_COMMAND_TASK</code> <code>CL_COMMAND_NATIVE_KERNEL</code> <code>CL_COMMAND_READ_BUFFER</code> <code>CL_COMMAND_WRITE_BUFFER</code> <code>CL_COMMAND_COPY_BUFFER</code> <code>CL_COMMAND_READ_IMAGE</code> <code>CL_COMMAND_WRITE_IMAGE</code> <code>CL_COMMAND_COPY_IMAGE</code> <code>CL_COMMAND_COPY_BUFFER_TO_IMAGE</code> <code>CL_COMMAND_COPY_IMAGE_TO_BUFFER</code> <code>CL_COMMAND_MAP_BUFFER</code> <code>CL_COMMAND_MAP_IMAGE</code> <code>CL_COMMAND_UNMAP_MEM_OBJECT</code> <code>CL_COMMAND_MARKER</code> <code>CL_COMMAND_ACQUIRE_GL_OBJECTS</code> <code>CL_COMMAND_RELEASE_GL_OBJECTS</code>
<code>CL_EVENT_COMMAND_EXECUTION_STATUS</code>	<code>cl_int</code>	Execution status is one of the following: <code>CL_QUEUED</code> — command has been enqueued into the command queue <code>CL_SUBMITTED</code> — enqueued command has been submitted to the OpenCL device <code>CL_RUNNING</code> — the command is currently executing by the device <code>CL_COMPLETE</code> — the command has completed Error code — negative integer value indicating command was terminated abnormally.
<code>CL_EVENT_REFERENCE_COUNT</code>	<code>cl_unit</code>	Returns the reference counter of the event object.

The following code shows how to get the execution status of an event object:


```
cl_int status;  
size_t return_size;  
err = clGetEventInfo(  
    event,  
    CL_EVENT_COMMAND_EXECUTION_STATUS,  
    sizeof(cl_int),  
    &status,  
    &return_size  
);
```

If the function is executed successfully, `clGetEventInfo()` returns `CL_SUCCESS`. Otherwise, one of the following values are returned:

- **CL_INVALID_EVENT** — The event object is invalid.
- **CL_INVALID_VALUE** — The `param_name` is not valid, or the `param_size` is less than the size of the return type.

Chapter 6

The OpenCL C Language

The OpenCL C programming language is based on the ISO C99 specification with some modifications and restrictions. This chapter highlights important aspects of the OpenCL C language that are used to create kernels for execution on OpenCL devices. For complete details on the language, see the OpenCL specification.

6.1 Restrictions

Key restrictions in the OpenCL C language are:

- Function pointers are not supported.
- Bit-fields are not supported.
- Variable length arrays are not supported.
- Recursion is not supported.
- No C99 standard headers such as `ctype.h`, `errno.h`, `stdlib.h`, etc. can be included.

6.2 Data Types

OpenCL C language has many built-in scalar types that should be familiar to C programmers. A set of vector data types and vector operations were also added for easier manipulation of data sets such as matrices. The OpenCL implement ensures that data is portable, so when data must be moved from one device to another—such as from a GPU to a CPU, OpenCL ensures that the data is always endian safe and properly aligned. This section describes the various data types supported by OpenCL C.

6.2.1 Scalar Data Types

The table below shows the common built-in scalar data types supported by OpenCL C as well as the matching types declared in the OpenCL API that can be used in the host application. Refer to the OpenCL specification for complete list of supported scalar data types.

Table 6–1 Supported Scalar Data Types

OpenCL C Type	OpenCL API Type for host app	Description
bool	n/a	conditional data type can be either true expanding to constant 1 or false expanding to constant 0
char	cl_char	signed two's complement 8-bit integer
unsigned char, uchar	cl_uchar	unsigned 8-bit integer
short	cl_short	signed two's complement 16-bit integer
unsigned short, ushort	cl_ushort	unsigned 16-bit integer
int	cl_int	signed two's complement 32-bit integer
unsigned int, uint	cl_uint	unsigned 32-bit integer
long	cl_long	signed two's complement 64-bit integer
unsigned long, ulong	cl_ulong	unsigned 64-bit integer
float	cl_float	IEEE 754 single precision floating point

6.2.2 Vector Data Types

One addition to the OpenCL C language is support of various vector data types. The vector data is defined with the type name follow by a value *n* that specifies the number of elements in the vector. For example a 4 component floating point vector would be `float4`. The supported components are: 2, 4, 8, and 16. The table below shows the built-in vector data types supported by OpenCL C as well as the matching types declared in the OpenCL API that can be used in the host application.

Table 6–2 Supported Vector Data Types

OpenCL C Type	OpenCL API Type for host app	Description
charn	cl_charn	signed two's complement 8-bit integer vector
ucharn	cl_ucharn	unsigned 8-bit integer vector
shortn	cl_shortn	signed two's complement 16-bit integer vector
ushortn	cl_ushortn	unsigned 16-bit integer vector
intn	cl_intn	signed two's complement 32-bit integer vector
uintn	cl_uintn	unsigned 32-bit integer vector
longn	cl_longn	signed two's complement 64-bit integer vector
ulongn	cl_ulongn	unsigned 64-bit integer vector
floatn	cl_floatn	floating point vector

Note: Even if the OpenCL device does not support any of all of the above vector data types, the OpenCL compiler will convert the vector types to the appropriate types supported built-in types supported by the OpenCL device.

There are several ways to access the components of a vector data type depending on the how many components are in the vector. Vectors types with 2 components such as `char2`, `uint2`, etc. can access the different components using `<vector2>.xy`. Vector

types with 4 components such as `long4`, `float4`, etc. can access components using `<vector4>.xyzw`. An example:

```
float2 pos;
pos.x = 1.0f;
pos.y = 1.0f;
pos.z = 1.0f; // illegal since vector only has 2 components

float4 c;
c.x = 1.0f;
c.y = 1.0f;
c.z = 1.0f;
c.w = 1.0f;
```

Vector components can also be accessed using numeric index to address the particular component. Table below shows the numeric indices used for accessing. When using number index to access the components, the indices must be prefixed with the letter `s` or `S`.

Vector components	Numeric indices
2 components	0, 1
4 components	0, 1, 2, 3
8 components	0, 1, 2, 3, 4, 5, 6, 7
16 components	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F

Examples:

```
float8 f;

f.s0 = 1.0f; // the 1st component in the vector
f.s7 = 1.0f; // the 8th component in the vector

float16 x;

f.sa = 1.0f; // or f.sA is the 10th component in the vector
f.sF = 1.0f; // or f.sF is the 16th component in the vector
```

OpenCL also provides quick addressing to a grouping of components in a vector:

Vector access suffix	Description
<code>.lo</code>	Returns the lower half of a vector
<code>.hi</code>	Returns the upper half of a vector
<code>.odd</code>	Returns the odd components of a vector
<code>.even</code>	Returns the even components of a vector

Examples:

```
float4 f = (float4) (1.0f, 2.0f, 3.0f, 4.0f);
float2 low, high;
float2 o, e;

low = f.lo; // returns f.xy (1.0f, 2.0f)
high = f.hi; // returns f.zw (3.0f, 4.0f)
o = f.odd; // returns f.yw (2.0f, 4.0f)
e = f.even; // returns f.xz (1.0f, 3.0f)
```

6.2.3 Vector Operations

In addition to all the typical operator C programmers are used to, such as `+`, `-`, `*`, `/`, `&`, `|`, etc., the OpenCL C language allows the same operators to be perform on vectors. Vector operations are perform on each component in the vector independently. The following examples illustrate this:

Example 1:

```
int4 vi0, vi1;
int v;

vi1 = vi0 + v;
```

is equivalent to:

```
vi1.x = vi0.x + v;
vi1.y = vi0.y + v;
vi1.z = vi0.z + v;
vi1.w = vi0.w + v;
```

Example 2:

```
float4 u, v, w;
w = u + v
w.odd = v.odd + u.odd;
```

is equivalent to:

```
w.x = u.x + v.x;
w.y = u.y + v.y;
w.z = u.z + v.z;
w.w = u.w + v.w;

w.y = v.y + u.y;
w.w = v.w + u.w;
```

6.3 Type Casting and Conversions

OpenCL C language allows implicit conversion of scalar data types and pointer types as described in the C99 specification. The implicit conversion process converts the value of a type into an equivalent value in the new type. The built-in scalar types

shown in [Table 6-1 Supported Scalar Data Types](#) all support implicit conversion. For built-in vector types, implicit conversions are not allowed. Thus, the following is illegal:

```
int4 v4;
float4 f = v4; // not allowed
```

Explicit conversions through casting for built-in data types are also supported. For example, the following will convert the floating point value of `x` into an integer value:

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

If, however, explicit casting is applied between vector data types such as those used in the following example, an error is generated:

```
int4 i;
float4 f = (float4) i; // not allowed
```

OpenCL provides a set of built-in function to explicitly convert between the data types. These functions can operate on both scalar and vector data types. The conversion functions take the form:

```
convert_<destination_type>(source_type)
```

Examples:

```
int4 i;
float4 f = convert_float4(i); // converts an int4 vector to float4

float f;
int i = convert_int(f); // converts a float scalar to an integer scalar
```

When converting between vector types, the number of components in each vector must be the same. The following example is illegal.

```
int8 i;
float4 f = convert_float4(i); // illegal
```

Sometimes the conversion may not produce the expected results due to rounding or out-of-range values. OpenCL allows us to specify how out-of-range values and rounding should be treated by using two optional modifiers on the conversion functions.

```
convert_<destination_type><_sat><_roundingMode>(source_type)
```

The saturation (`_sat`) modifier specifies how an out-of-range conversion is handled. When the value to be converted is beyond the range that can be represented by the target type, the values are clamped to the nearest representable value. The `_sat` modifier is only supported when converting to an integer type. Conversion to a floating point type follows IEEE754 rules, and may not use the saturation modifier.

The rounding modifier specifies the rounding mode to use in the conversion. The table below list the supported rounding modes.

Rounding Modifier	Description
<code>_rte</code>	Round to nearest even
<code>_rtz</code>	Round towards zero
<code>_rtp</code>	Round towards positive infinity
<code>_rtn</code>	Round towards negative infinity
No modifier	Defaults to <code>_rtz</code> for conversions to integer Defaults to <code>_rte</code> for conversions to floating point

Explicit conversion examples:

```
float4    f = (float4)(-1.0f, 252.5f, 254.6f, 1.2E9f);
uchar4    c = convert_uchar4_sat(f);
// c = (0, 253, 255, 255)
// negative value clamped to 0, value > TYPE MAX is set to the type MAX
// -1.0 clamped to 0, 1.2E9f clamped to 255

float4    f = (float4)(-1.0f, 252.5f, 254.6f, 1.2E9f);
uchar4    c = convert_uchar4_sat_rte(f);
// c = (0, 252, 255, 255)
// 252.5f round down to near even becomes 252

int4 i;
float4 = convert_float4(i);
// convert to floating point using the default rounding mode

int4 i;
float4 = convert_float4_rtp(i);
// convert to floating point. Integer values not representable as float
// are rounded up to the next representable float
```

Built-in scalar and vector data types can also be reinterpreted as another data type as long as they are the same size. The function `as_<typen>(value)` reinterprets the bit pattern in the source to another type without any modification. For example, the value `1.0f` is represented as `0x3f800000` in an IEEE754 value. The value `1.0f` can be reinterpreted as a `uint` this way:

```
uint i = as_uint(1.0f);
// i will have value 0x3f800000
```


6.4 Qualifiers

6.4.1 Address Space Qualifiers

3.3.3 The Memory Model briefly described the OpenCL memory model defines four regions of memory for work-items to access when executing a kernel. The different address spaces allow memory to be shared between work-items, or allow us to choose the best memory space for the operations to be performed. The four disjointed address spaces are: `__global`, `__local`, `__constant`, and `__private`. To allocate memory for a specific region of memory, one of the above qualifiers may be used in the variable declaration.

All functions (including the `__kernel` function) and their arguments or local variables are in the `__private` address space. Arguments of a `__kernel` function declared as a pointer type can point to only one of the following memory spaces: `__global`, `__local`, or `__constant`. Assigning a pointer address to one space to another is not allowed. For example, a pointer to `__global` can only be assigned to a pointer to `__global`. Casting of pointers between different address spaces may cause unexpected behavior. A `__kernel` function with arguments declared as `image2d_t` or `image3d_t` can only point to the `__global` address space.

Examples:

```
__global float *ptr // the pointer ptr is declared in the __private address
                  // space and points to a float that is in the __global
                  // address space

int4 x           // declares an int4 vector in the __private address
```

Global address space

This address space refers to memory objects such as scalars, vectors, buffer objects, or image objects that are allocated in the global memory pool. For a GPU compute device, this is typically the frame buffer memory.

Local address space

This address space is typically used by local variables that are allocated in local memory. Memory in the local address space can be shared by all work-items of a work group. For example, local memory on a GPU compute device could be the local data store for one of the compute units (or core) on the GPU.

Constant address space

This address space describes variables that allocated in global memory pool but can only be accessed as read-only variables. These variables are accessible by the entire global work-items. Variable that need to have a global scope must be declared in the constant address space.

Private address space

This address space describes variables that are passed into all functions (including `__kernel` functions) or variables declared without a qualifier. A private variable can only be accessed by the work-item in which it was declared.

6.4.2 Image Qualifiers

OpenCL also provides access to qualifiers when passing an image memory object as arguments to `__kernel` functions. The qualifier `__read_only` and `__write_only` can specify whether the image object passed to the kernel is read-only or write-only. Default access is `__read_only` when no qualifier is explicitly provided. Kernel functions cannot have both read and write access to the same image memory object.

The following kernel declaration declares the argument `inputImage` as a read-only image object, and `outputImage` as a write-only image object:

```
__kernel void myfunc(__read_only image2d_t inputImage,
                    __write_only image2d_t outputImage)
```

6.5 Built-in Functions

The OpenCL C language includes a wide set of built-in functions for many types of operations such as math functions, work-item functions, image access functions. Refer to the OpenCL Specification for details of all the built-in function. This section will describes some of the common functions used to operate on work-items, image objects, and work-items synchronization.

6.5.1 Work-item Functions

Work-item functions can be used to query the information relating to the data that the kernel is asked to process. These functions allow querying of dimension size of the data, the global size for each dimension, the local work size, number of work-groups, as well the unique global and local work-item ID of the kernel that is being executed.

```
// returns the number of dimensions of the data problem space
uint get_work_dim()

// returns the number total work-items for the specified dimension
size_t get_global_size(dimidx)

// returns the number of local work-items in the work-group specified by dimension
size_t get_local_size(dimidx)

// returns the unique global work-item ID for the specified dimension
size_t get_global_id(dimidx)

// returns the unique local work-item ID in the work-group for the specified dimension
size_t get_local_id(dimidx)

// returns the number of work-groups for the specified dimension
size_t get_num_groups(dimidx)

// returns the unique ID of the work-group being processed by the kernel
size_t get_group_id(dimidx)
```

The following kernel function illustrates how these functions are used:

```
__kernel void square(__global int *input, __global int *output)
{
    size_t id = get_global_id(0);
    output[id] = input[id] * input[id];
}
```

The following figure shows the input and expected output:

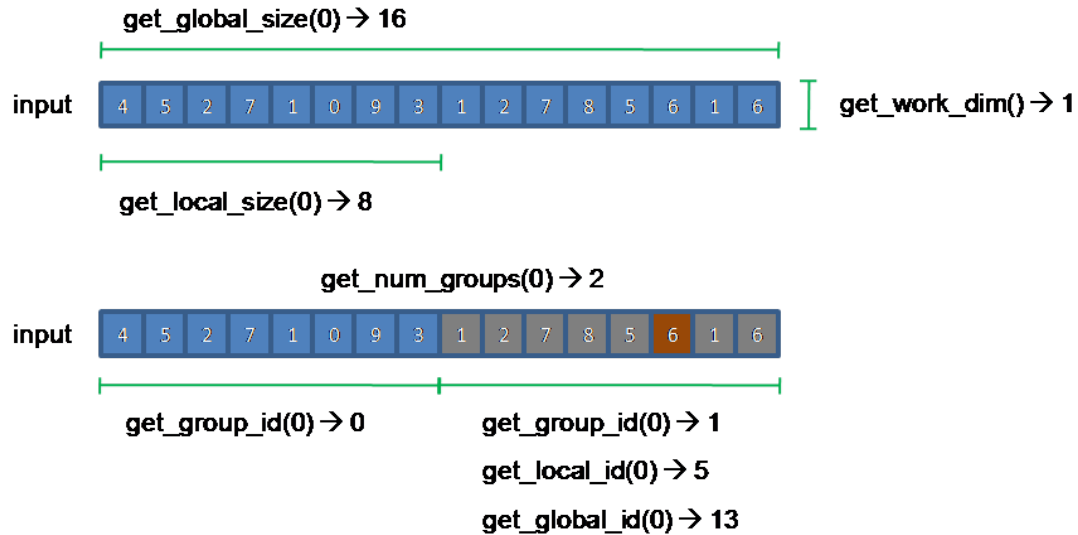
input	4	5	2	7	1	0	9	3	1	2	7	8	5	6	1	6
output	16	25	4	49	1	0	81	9	1	4	49	64	25	36	1	36

Since the kernel function is a data parallel function, it is executed for each work-item. When a work-item calls `get_global_id(0)` the request is for the unique global work-item ID used to index the data. The example below, the work-item instance is returned `id=6` when it makes a call to `get_global_id(0)`. The work-item can then use the `id` to index the data to perform the operation.

get_global_id(0) → 6

input	4	5	2	7	1	0	9	3	1	2	7	8	5	6	1	6
output	16	25	4	49	1	0	81	9	1	4	49	64	25	36	1	36

If, for example, the kernel function is enqueued by indicating that the input will be divided into groups of eight elements, the results of the various work-item functions will appear as shown in the following image.



6.5.2 Image Access Functions

When using image memory objects, the data in the image object can only be read and written using the built-in image access functions. A wide range of read and write functions are available, depending on the image channel data type when the image memory object is created. The following is a list of some of the built-in image access functions. Refer to the OpenCL specification for the details of each function.

```
read_imagef() // for reading image with float-point data channel
read_imagei() // for reading image with integer data channel
read_imageui() // for reading image with unsigned integer data channel

write_imagef() // for writing image with float-point data channel
write_imagei() // for writing image with integer data channel
write_imageui() // for writing image with unsigned integer data channel

get_image_width() // returns width of the 2D or 3D image
get_image_height() // returns height of the 2D or 3D image
get_image_depth() // returns depth of the 3D image
```

6.5.3 Synchronization Functions

The OpenCL C language provides functions to allow synchronization of work-items. The memory model discussion in [Chapter 3 Introduction to OpenCL](#) described how synchronization can only occur between work-items in the same work-group. To achieve that, OpenCL implements a barrier memory fence for synchronization.

The function `barrier(mem_fence_flag)` creates a barrier that blocks the current work-item until all other work-items in the same group has executed the barrier before allowing the work-item to proceed beyond the barrier. It is important to note that when using barrier, all work-items in the work-group must execute the barrier function. If the barrier function is called within a conditional statement, it is important to ensure that all work-items in the work-group enter the conditional statement to execute the barrier.

For example, the following function is an illegal use of barrier because the barrier will not be encountered for more than five work-items:

```
__kernel void read(__global float *input, __global float *output)
{
    size_t id = get_global_id(0);
    if (id < 5)
        barrier(CLK_GLOBAL_MEM_FENCE);
    else
        ...
}
```

The `mem_fence_flag` can be either `CLK_LOCAL_MEM_FENCE`, or `CLK_GLOBAL_MEM_FENCE` is use by the barrier function to either flush any variable in local or global memory or setup a memory fence to ensure that correct ordering of memory operations to either local or global memory.

OpenCL C language implements memory fence functions to provide ordering between memory operations of a work-item. Memory fence can be useful when work-items needs to write data to a buffer and then read back the updated data. Using memory fence ensures that reads or writes before the memory fence have been committed to memory. OpenCL allows explicit setup of memory fence by using one of the following functions.

```
// ensures all reads and writes before the memory fence have committed to memory
void mem_fence(mem_fence_flag)

// ensures all reads before memory fence have completed
void read_mem_fence(mem_fence_flag)

// ensures all writes before memory fence have completed
void write_mem_fence(mem_fence_flag)
```


Chapter 7

Application Optimization and Porting

This chapter describes OpenCL kernel debugging and provides performance and optimization tips for kernel execution on AMD GPU and CPU devices. Also included are language equivalents for use in porting applications from NVIDIA's C for CUDA language to OpenCL.

7.1 Debugging OpenCL

The ATI Stream SDK v2 provides debugging features that allow the debugging of OpenCL kernels on Linux using GDB. This section covers only some of the basic GDB commands; for complete GDB documentation, see [GDB: The GNU Project Debugger](#).

7.1.1 Setting Up for Debugging

To enable debugging, an OpenCL program passes the "-g" option to the build `clBuildProgram()` function:

```
err = clBuildProgram(program, 1, devices, "-g", NULL, NULL);
```

To avoid changes to the source code, the following environment variable can be used to force debugging when compiling for a CPU device:

```
export CPU_COMPILER_OPTIONS=-g
```

When debugging, the kernel program must be set to execute on a CPU device. It is also important to set the environment variable so the kernel program is executed deterministically:

```
export CPU_MAX_COMPUTE_UNITS=1
```

7.1.2 Setting Breakpoints

If no breakpoint is set during a GDB session, the program does not stop until execution has completed. To set a breakpoint using GDB:

```
b linenumber
```

or

```
b function_name | kernel_function_name
```

A breakpoint can be set at a particular line in the source code by simply specifying the line number. A break point can also be set at the start of a function by specifying the `function_name`. To set a break point for a kernel function, use the construct `__OpenCL_function_kernel`. For example, to set a breakpoint for the following kernel function:

```
__kernel void square(__global int *input, __global int *output)
```

Use the following code:

```
b __OpenCL_square_kernel
```

Note: The OpenCL kernel symbols are not visible to the debugger until the kernel is loaded, when a question is asked about making the breakpoint pending on future library load:

```
(gdb) b __OpenCL_hello_kernel
Function "__OpenCL_hello_kernel" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (__OpenCL_hello_kernel) pending.
```

To check that the symbol is correct, set a breakpoint in the host code at the `clEnqueueNDRangeKernel` function, then list all OpenCL symbols using the GDB command:

```
info functions __OpenCL
```


Figure 7–1 Listing OpenCL Debugging Symbols

```
(gdb) b clEnqueueNDRangeKernel
Function "clEnqueueNDRangeKernel" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (clEnqueueNDRangeKernel) pending.
(gdb) r
Starting program: /home/amd/hello
[Thread debugging using libthread_db enabled]
[New Thread 0xb7df86d0 (LWP 14926)]
[New Thread 0xa60e3b90 (LWP 14929)]
[New Thread 0xa60a2b90 (LWP 14930)]
[New Thread 0xa6091b90 (LWP 14931)]
[New Thread 0xa6080b90 (LWP 14932)]
[New Thread 0xa606fb90 (LWP 14933)]
size of workgroup 1024
[Switching to Thread 0xb7df86d0 (LWP 14926)]

Breakpoint 1, 0xb80b897b in clEnqueueNDRangeKernel () from /home/amd/ati-stream-sdk-v2.01-linux32/lib/x86/libOpenCL.so
(gdb) info functions __OpenCL
All functions matching regular expression "__OpenCL":

File OCLLevidZh.cl:
void __OpenCL_hello_kernel(void *, void *);

Non-debugging symbols:
0xa5aed59c __OpenCL_hello_kernel@plt
0xa5aed630 __OpenCL_hello_stub
(gdb)
```

A conditional breakpoint can also be set to stop the program at a particular work-item. This is done by setting a conditional breakpoint when `get_global_id == ID`. For example, to set a breakpoint at the kernel function for the work-item with *global id*==5:

```
b __OpenCL_square_kernel if get_global_id(0) == 5
```

Figure 7–2 Setting a Conditional Breakpoint

```
(gdb) b __OpenCL_hello_kernel if get_global_id(0)==5
Function "__OpenCL_hello_kernel" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (__OpenCL_hello_kernel if get_global_id(0)==5) pending.
(gdb) r
Starting program: /home/amd/hello
[Thread debugging using libthread_db enabled]
[New Thread 0xb7d556d0 (LWP 15699)]
[New Thread 0xa6040b90 (LWP 15702)]
[New Thread 0xa5ffffb90 (LWP 15703)]
size of workgroup 1024

Breakpoint 1, __OpenCL_hello_kernel (input=0x0, output=0x0) at OCLdzJyoD.cl:2
2      {
(gdb) n
3      size_t id = get_global_id(0);
(gdb) n
4      output[id] = input[id] * input[id];
(gdb) p id
$1 = 5
(gdb)
```

7.2 Performance Measurement

The OpenCL runtime provides a built-in mechanism for timing the execution of kernels. This allows the developer to evaluate kernel performance for maximum

optimization. Kernel command profiling is done by specifying the `CL_QUEUE_PROFILING_ENABLE` properties when creating the command queue with `clCreateCommandQueue()`. The command queue properties can also be set after the queue has been created by using the `clSetCommandQueueProperties()` function.

When queue profiling is enabled, the OpenCL runtime automatically records timestamp information for every kernel and memory operation submitted to the queue. The profiling function and supported data are as follows:

```
err = clGetEventProfilingInfo(
    event,           // the event object to get info for
    param_name       // the profiling data to query - see list below
    param_value_size // the size of memory pointed by param_value
    param_value       // pointer to memory in which the query result is returned
    param_actual_size // actual number of bytes copied to param_value
);
```

Table 7–1 Supported Profiling Data for `clGetEventProfilingInfo()`

Profiling data	Return Type	Information returned
<code>CL_PROFILING_COMMAND_QUEUE</code>	<code>cl_ulong</code>	A 64-bit counter in nanoseconds when the command is enqueued in a command queue.
<code>CL_PROFILING_COMMAND_SUBMIT</code>	<code>cl_ulong</code>	A 64-bit counter in nanoseconds when the command that has been enqueued is submitted to compute the device for execution
<code>CL_PROFILING_COMMAND_START</code>	<code>cl_ulong</code>	A 64-bit counter in nanoseconds when the command started execution on the compute device
<code>CL_PROFILING_COMMAND_END</code>	<code>cl_ulong</code>	A 64-bit counter in nanoseconds when the command has finished execution on the compute device

If the function executes successfully, `clGetEventProfilingInfo()` returns `CL_SUCCESS`. Otherwise, one of the following error codes are returned:

- **`CL_INVALID_EVENT`** — The event object is not valid.
- **`CL_PROFILING_INFO_NOT_AVAILABLE`** — The command queue does not have the `CL_QUEUE_PROFILING_ENABLE` properties flag set, or the event associated with the command is not completed.
- **`CL_INVALID_VALUE`** — The query data `param_name` is not valid, or the size `param_actual_size` is greater than `param_value_size`.

The following sample code determines the execution time for a kernel command:

```

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), &endTime, NULL);
cl_ulong elapsedTime = endTime-startTime;

```

7.2.1 Using the ATI Stream Profiler

The ATI Stream Profiler provides a Microsoft® Visual Studio® integrated view of key static kernel characteristics such as work-group dimensions and memory transfer sizes, as well as kernel execution time, dynamic hardware performance counter information (ALU operations, local bank conflicts), and kernel disassembly. The ATI Stream Profiler is installed if the SDK was installed using the express option.

To confirm that the profiler is installed:

1. Open a Visual Studio solution file.
2. Select **Help ► About Microsoft Visual Studio** from the Visual Studio's main menu bar. Under Installed products, you should find ATI Stream Profiler 1.1. If you don't see it, open a command window using the **Run as Administrator** option and run **C:\Program Files (x86)\Microsoft Visual Studio 9.0\Common7\IDE\devenv.exe /setup** for a 64-bit system or **C:\Program Files\Microsoft Visual Studio 9.0\Common7\IDE\devenv.exe /setup** for a 32-bit system from a command line prompt (with "Run as Administrator" option).

To use the profiler plugin:

1. Open an OpenCL solution or project file.
2. Set a startup project by right-clicking on a project and selecting **Setup as StartUp Project**.
3. Compile the project and run it. Verify that it compiles and runs successfully.
4. To start profiling, click on the **Start Profiling** button under the **OpenCL Session List** panel. If you can't find the **OpenCL Session List Panel**, enable it by selecting on **View ► Other Windows ► OpenCL Session List Window** from the Visual Studio's main menu bar. An OpenCL Session panel will be docked to the main document panel. This panel shows the profiling result.
5. Hover the mouse pointer over the profile header in the OpenCL Session panel to get the description of the counters.
6. Click on the kernel name (the first column) to see the ISA and IL code of the kernel. An OpenCL Code Viewer panel will be docked to the main document panel.

Note:

- If you run the kernel in the CPU mode, only the global work size, work group size, local memory, memory transfer size, and kernel time statistics will be available.
- Under the project directory, a new ProfilerOutput directory is created. This directory holds the profiling result of each session in a .csv file.

The following table lists the performance counters available through the ATI Stream Profiler.

Name	Description
Method	The kernel name or the memory operation name.
ExecutionOrder	The order of execution for the kernel and memory operations from the program.
GlobalWorkSize	The global work-item size of the kernel.
GroupWorkSize	The work-group size of the kernel.
KernelTime	Time spent executing the kernel in milliseconds (does not include the kernel setup time).
LocalMem	The amount of local memory in bytes being used by the kernel.
MemTransferSize	The memory transfer size in bytes.
ALU	The average ALU instructions executed per thread (affected by flow control).
Fetch	The average Fetch instructions (from global memory) executed per thread (affected by flow control).
Write	The average Write instructions (to global memory) executed per thread (affected by flow control).
Wavefront	Total wavefronts.
ALUBusy	The percentage of time ALU instructions are processed relative to GPUPTime.
ALUFetchRatio	The ratio of ALU to Fetch instructions.
ALUPacking	The ALU vector packing efficiency (in percentage). This value indicates how well the Shader Compiler packs the scalar or vector ALU in your kernel to the 5-way VLIW instructions. Values below 70 percent indicate that ALU dependency chains may be preventing full utilization of the processor.
FetchUnitBusy	The percentage of time the Fetch unit is active relative to GPUPTime. This is measured with all extra fetches and any cache or memory effects taken into account.
FetchUnitStalled	The percentage of time the Fetch unit is stalled relative to GPUPTime. Try reducing the number of fetches or reducing the amount per fetch if possible.
WriteUnitStalled	The percentage of time Write unit is stalled relative to GPUPTime.

Additional counters available for the ATI Radeon HD 5000 Series graphics cards:

Name	Description
ALUStalledByLDS	The percentage of time ALU is stalled by LDS input queue being full and output queue is not ready relative to GPUBusy. If there are LDS bank conflicts, reduce it. Otherwise, try reducing the number of LDS accesses if possible.
LDSBankConflict	The percentage of time LDS is stalled by bank conflicts relative to GPUPTime.

For additional information regarding the ATI Stream profiler, consult the [ATI Stream Profiler Knowledge Base](#).

7.3 General Optimization Tips

This section offers general tips for optimizing kernel execution on AMD GPUs and CPUs. For more information, such as how the AMD GPU architecture maps to the OpenCL implementation, see the [ATI Stream SDK OpenCL Programming Guide](#).

7.3.1 Use Local Memory

Using local memory is typically an order of magnitude faster than using global memory. AMD GPU includes a fast, high-bandwidth local memory for each work-group. All work-items in the work-group can efficiently share data using the high-bandwidth local memory. Collaborative read/write to the local memory can lead to highly efficient memory accessing. Collaborative write works by having each work-item write a subsection of an array, and as the work-item execute in parallel, the entire array is written. Before reading the values written collaboratively, the kernel must issue a `barrier()` call to ensure that memory is consistent across all work-items.

The following example calculates the transpose of a matrix using collaborative writes, then reads, from local memory. This implementation is twice as fast as the equivalent implementation that uses only global memory.

```

__kernel
void matrixTranspose(__global float * output,
                    __global float * input,
                    __local float * block,
                    const uint width,
                    const uint height,
                    const uint blockSize)
{
    uint globalIdx = get_global_id(0);
    uint globalIdy = get_global_id(1);

    uint localIdx = get_local_id(0);
    uint localIdy = get_local_id(1);

    /* copy from input to local memory */
    block[localIdy*blockSize + localIdx] =
    input[globalIdy*width + globalIdx];

    /* wait until the whole block is filled */
    barrier(CLK_LOCAL_MEM_FENCE);

    uint groupIdx = get_group_id(0);
    uint groupIdy = get_group_id(1);

    /* calculate the corresponding target location for transpose
    by inverting x and y values*/
    uint targetGlobalIdx = groupIdy*blockSize + localIdy;
    uint targetGlobalIdy = groupIdx*blockSize + localIdx;

    /* calculate the corresponding raster indices of source and target */
    uint targetIndex = targetGlobalIdy*height + targetGlobalIdx;
    uint sourceIndex = localIdy * blockSize + localIdx;

    /* read final data from the local memory */
    output[targetIndex] = block[sourceIndex];
}

```

7.3.2 Work-group Size

The OpenCL data-parallel model allows you to divide work-items into work-groups. Work-group division can be performed explicitly or implicitly.

- **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.

Explicit group division generally offers better performance. To maximize efficiency, however, always choose the largest group size supported by the target device.

AMD GPUs are optimized with work-groups sized in multiple of 64. The maximum supported device work-group size can be determined by querying `CL_DEVICE_MAX_WORK_GROUP_SIZE` with `clGetDeviceInfo()`. Sometimes, depending on kernel resource usage or instructions used by the kernel, it may not be possible to use the maximum work group size supported by the device and specified by

`CL_DEVICE_MAX_WORK_GROUP_SIZE`. In this case, `clGetKernelWorkGroupInfo()` can be used to retrieve information about the kernel object that is specific to a device. The parameter of interest is `CL_KERNEL_WORK_GROUP_SIZE`, which determines the maximum work-group size that can be used to execute a kernel on a specific device based on the resource requirement of the kernel. Refer to the OpenCL specification for more details on `clGetKernelWorkGroupInfo()`.

7.3.3 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 ATI Stream SDK v2 OpenCL compiler performs simple loop unrolling optimizations. However, for more complex loop unrolling, it may be beneficial to do this manually. For more on image convolution using OpenCL and how to further optimize the OpenCL kernel by performing loop unrolling, see the [Image Convolution tutorial](#) at the ATI Stream SDK v2 product page.

7.3.4 Reduce Data and Instructions

If possible, create a smaller version of the data set for easier debugging and faster turnaround. GPUs do not have automatic caching mechanisms and typically scale well as resources are added. In many cases, therefore, performance optimization for the smaller data implementation also benefits the full-size data set.

The profiler reports statistics on a per-kernel granularity. To further narrow down bottlenecks, it can be useful to remove or comment-out sections of code, then re-collect timing data to determine problem areas.

7.3.5 Use Built-in Vector Types

When possible, use built-in vector types such as `float4` and the built-in vector functions (`vload`, `vstore`, etc.). These enable the ATI Stream SDK OpenCL implementation to generate efficiently-packed SSE instructions when running on the CPU. Vectorization can benefit both AMD CPUs and GPUs.

7.4 Porting CUDA to OpenCL

The data-parallel programming model in OpenCL shares some functionality with NVIDIA's C for CUDA programming model, making it relatively straightforward to convert programs from CUDA to OpenCL. This section describes terminology equivalents in C for CUDA and OpenCL.

7.4.1 General Terminology

The following table provides general terminology equivalents for describing computations and memory spaces between C for CUDA and OpenCL.

Table 7–2 General Terminology Equivalents

C for CUDA Terminology	OpenCL Terminology
Thread	Work-item
Thread block	Work-group
Global memory	Global memory
Constant memory	Constant memory
Shared memory	Local memory
Local memory	Private memory

7.4.2 Kernel Qualifiers

The following table provides equivalents for qualifiers used when writing kernel functions in both C for CUDA and OpenCL. The most significant difference is the main entry point to the kernel function. C for CUDA uses the `__global__` qualifier and local functions (not callable by the host) are prefixed with the `__device__` qualifier. In OpenCL, entry functions use the `__kernel` qualifier, and qualifiers are not required for local (non-entry point) functions.

Table 7–3 Kernel Qualifier Equivalents

C for CUDA Terminology	OpenCL Terminology
<code>__global__</code> function	<code>__kernel</code> function
<code>__device__</code> function	function (no qualifier required)
<code>__constant__</code> variable declaration	<code>__constant</code> variable declaration
<code>__device__</code> variable declaration	<code>__global</code> variable declaration
<code>__shared__</code> variable declaration	<code>__local</code> variable declaration

7.4.3 Kernel Indexing

The following table indexing access function equivalents for C for CUDA and OpenCL. CUDA provides indexing via special predefined variables, while OpenCL accomplished the same through function calls. Global indexing in CUDA requires manual computation, while OpenCL provides a global indexing function.

Table 7–4 Indexing Terminology Equivalents Used in Kernel Functions

C for CUDA Terminology	OpenCL Terminology
gridDim	get_num_groups()
blockDim	get_local_size()
blockIdx	get_group_id()
threadIdx	get_local_id()
Global index calculated by combining blockDim, blockIdx, and threadIdx.	get_global_id()
Global size calculated by combining blockDim, gridDim.	get_global_size()

7.4.4 Kernel Synchronization

The following table provides equivalents for functions used for synchronization in kernel functions in C for CUDA and OpenCL. `__syncthreads()` and `barrier()` functions are similar in that they provide synchronization for all work-items in a work-group. In OpenCL, work-items are blocked until all work-items in the work-group have called `barrier()`.

`__threadfence()` and `mem_fence()` force various orderings on outstanding memory transactions, which can allow for more sophisticated sharing of data. For example, `mem_fence()` forces all outstanding loads and stores to be completed before execution proceeds, disallowing the compiler, runtime, and hardware from reordering any loads and stores through the `mem_fence()`. This can be used to ensure that all data produced in a work-group is flushed to global memory before signaling another work-group that execution has completed.

Table 7–5 Synchronization Terminology Equivalents Used in Kernel Functions

C for CUDA Terminology	OpenCL Terminology
<code>__syncthreads()</code>	<code>barrier()</code>
<code>__threadfence()</code>	No direct equivalent.
<code>__threadfence_block()</code>	<code>mem_fence(CLK_GLOBAL_MEM_FENCE CLK_LOCAL_MEM_FENCE)</code>
No direct equivalent.	<code>read_mem_fence()</code>
No direct equivalent.	<code>write_mem_fence()</code>

7.4.5 General API Terminology

The following table provides general API terminology equivalents between C for CUDA and OpenCL. Note that in OpenCL, command queues provide task parallelism by allowing the developer to declare dependences between commands executing on a device. CUDA does not have a direct equivalent. The closest approximation would be the CUDA Stream mechanism, which allows kernels and memory transaction to be placed in independent streams. This is different from the command queue, since it does not provide parallelism within the stream, making synchronization between streams difficult.

Table 7–6 General API Terminology Equivalents

C for CUDA Terminology	OpenCL Terminology
CUdevice	cl_device_id
CUcontext	cl_context
CUmodule	cl_program
CUfunction	cl_kernel
CUdeviceptr	cl_mem
No direct equivalent. Closest approximation would be the CUDA Stream mechanism.	cl_command_queue

7.4.6 Important API Calls

The following table provides equivalents for calls used to set up host programs to execute parallel kernels in C for CUDA and OpenCL. A significant difference is that OpenCL is capable of compiling programs off-line or during runtime. CUDA only allows programs to be compiled off-line.

Table 7–7 API Call Equivalents

C for CUDA Terminology	OpenCL Terminology
cuInit()	No OpenCL initialization required
cuDeviceGet()	clGetContextInfo()
cuCtxCreate()	clCreateContextFromType()
No direct equivalent	clCreateCommandQueue()
cuModuleLoad() <i>Note: Requires pre-compiled binary.</i>	clCreateProgramWithSource() or clCreateProgramWithBinary()
No direct equivalent. CUDA programs are compiled off-line	clBuildProgram()
cuModuleGetFunction()	clCreateKernel()
cuMemAlloc()	clCreateBuffer()
cuMemcpyHtoD()	clEnqueueWriteBuffer()
cuMemcpyDtoH()	clEnqueueReadBuffer()
cuFuncSetBlockShape()	No direct equivalent; functionality is part of clEnqueueNDRangeKernel()
cuParamSeti()	clSetKernelArg()
cuParamSetSize()	No direct equivalent; functionality is part of clSetKernelArg()
cuLaunchGrid()	clEnqueueNDRangeKernel()
cuMemFree()	clReleaseMemObj()

7.4.7 Additional Tips

- Pointers in OpenCL kernels must be prefixed with their memory space. For example, a pointer to local memory would be declared as `__local int* p;`. This applies to kernel arguments as well; data passed to a kernel are usually arrays represented by `__global` pointers.
- CUDA encourages the use of scalar code in kernels. While this works in OpenCL as well, but depending on the target architecture it may be more efficient to write programs operating on OpenCL's vector types (such as `float4`) rather than pure scalar types. This is useful for both AMD CPUs and AMD GPUs, which can operate efficiently on vector types. OpenCL also provides flexible swizzle/broadcast primitives for efficient creation and rearrangement of vector types.
- CUDA does not provide rich facilities for task parallelism, so it may be beneficial to think about how to take advantage of OpenCL's task parallelism when porting.

Chapter 8

Exercises

This chapter offers exercises to help further your understanding of OpenCL programming. Solutions are available in the Appendix.

8.1 Matrix Transposition Exercise

This exercise involves creating an OpenCL application that takes as input a matrix (width × height) and determines the transposition of the input matrix.

In the following example, the transposition of matrix A is obtained by swapping the writing of the rows in A with the columns of A^T:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}^T = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

8.1.1 Completing the Code

Given the source skeleton below, complete the necessary OpenCL host portions in the main application as well as the kernel function that perform the matrix transposition using parallelism. The kernel source is to be placed in a file called `transposeMatrix_kernel.cl` and read into the `kernel_source` string by the main program.

Skeleton OpenCL host application to find a matrix transposition:

```

001 // transposeMatrix.c
002 #include <stdio.h>
003 #include <stdlib.h>
004 #include "CL/cl.h"
005
006 #define DATA_SIZE 65536
007 #define ROWS 16
008 #define COLS 16
009
010 int main()
011 {
012
013     cl_float *inputMatrix;
014     cl_float *results;
015     cl_uint width = COLS;
016     cl_uint height = ROWS;
017
018     // OpenCL host variables go here:
019
020     // variables used to read kernel source file
021     FILE *fp;
022     long filelen;
023     long readlen;
024     char *kernel_src; // char string to hold kernel source
025
026
027     // initialize inputMatrix with some data and print it
028     int x,y;
029     int data=0;
030
031     inputMatrix = malloc(sizeof(cl_float)*width*height);
032     results = malloc(sizeof(cl_float)*width*height);
033     printf("Input Matrix\n");
034     for(y=0;y<height;y++)
035     {
036         for(x=0;x<width;x++)
037         {
038             inputMatrix[y*height+x]= data;
039             results[y*height+x]=0;
040             data++;
041
042             printf("%.2f, ",inputMatrix[y*height+x]);
043         }
044         printf("\n");
045     }
046
047     // read the kernel
048     fp = fopen("transposeMatrix_kernel.cl","r");
049     fseek(fp,0L, SEEK_END);
050     filelen = ftell(fp);
051     rewind(fp);
052
053     kernel_src = malloc(sizeof(char)*(filelen+1));
054     readlen = fread(kernel_src,1,filelen,fp);
055     if(readlen!= filelen)
056     {
057         printf("error reading file\n");
058         exit(1);
059     }
060
061     // ensure the string is NULL terminated

```

```

062     kernel_src[filelen+1]='\0';
063
064
065     // ----- Insert OpenCL host source here -----
066
067
068
069     // ----- End of OpenCL host section -----
070
071
072     // print out the transposed matrix
073     printf("\nTransposed Matrix \n");
074     for(y=0;y<height;y++)
075     {
076         for(x=0;x<width;x++)
077         {
078             printf("%.2f , ",results[y*height+x]);
079         }
080         printf("\n");
081     }
082
083     free(kernel_src);
084     free(inputMatrix);
085     free(results);
086     return 0;
087 }

```

Skeleton kernel source for calculating transpose of matrix:

```

// Kernel source file for calculating the transpose of a matrix
__kernel void matrixTranspose(.....)
{
    // insert kernel source here
}

```

For the solution to this exercise, see [A.1 Matrix Transposition Solution](#).

8.2 Matrix Multiplication Exercise

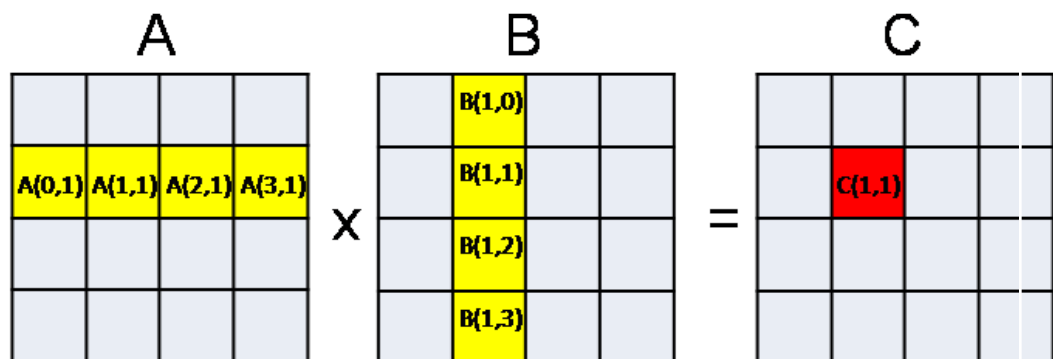
This exercise demonstrates how to improve the performance of a kernel by breaking the work into subgroups. Use the kernel implementation below that performance the multiplication of two matrices and create a new kernel function that decomposes the multiplication into small work-groups working in parallel.

```

001 __kernel void multMatrixSimple(__global float *mO,
002                               __global float *mA,
003                               __global float *mB,
004                               uint widthA, uint widthB)
005 {
006 {
007     int globalIdx = get_global_id(0);
008     int globalIdy = get_global_id(1);
009
010     float sum = 0;
011
012     // multiply each element of the row with each
013     // element of the cols pointed by the current work-item
014     for (int i=0; i< widthA; i++)
015     {
016         float tempA = mA[globalIdy * widthA + i];
017         float tempB = mB[i * widthB + globalIdx];
018         sum += tempA * tempB;
019     }
020     // copy the sum to the device buffer
021     mO[globalIdy * widthA + globalIdx] = sum;
022 }
023 }

```

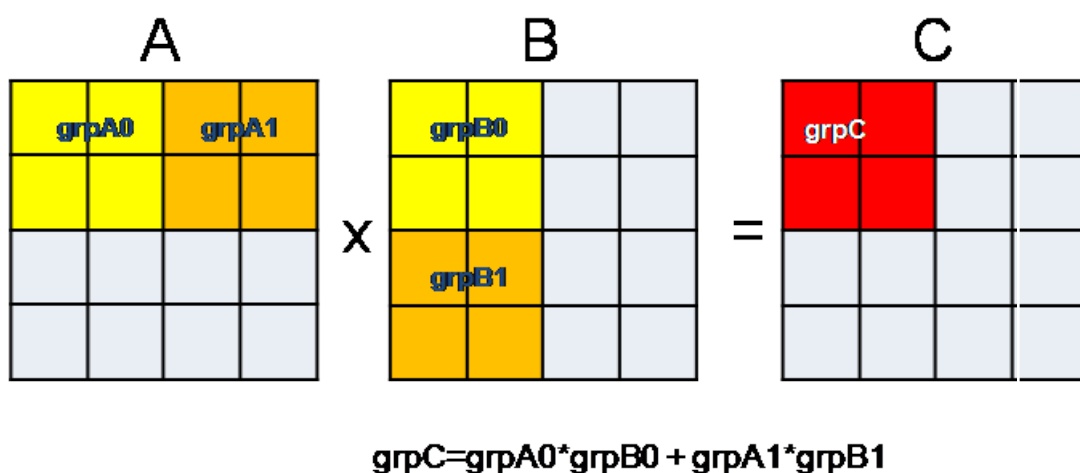
The multiplication of two matrices is shown in the following images. The kernel function simply does what is shown in the figure. For every element in matrix C, a work-item calculates the sums of products for the corresponding rows in image A and columns in image B.



$$C(1,1) = A(0,1) * B(1,0) + A(1,1) * B(1,1) + A(2,1) * B(1,2) + A(3,1) * B(1,3)$$

For each element in image C, the kernel function must access the input data from the global memory in images A and B. For very large matrices, a large amount of access to global memory is required. Access to global memory is slow, however, and should be minimized where possible. Also note that work-items that are on the same row or column access the same data multiple times. In the previous figure, work-items C(1,0), C(1,2), and C(1,3) all need to access the same column data from B as C(1,1). Thus, the number of access to global memory increases.

The number of global memory accesses can be minimized by changing the kernel function to copy a block of data from A and B into fast local memory that can be shared by all work-items in the group. The following figure shows one possible way to decompose the matrix into local sub-matrices that can be shared by all work-items:



This kernel should copy a portion of A and B into local matrices (grpA and grpB) and then calculate the grpC output by using the local matrices. For very large input matrices, this method of decomposition dramatically minimizes the number of access to global memory, since all work-items within grpC can make use of the local memory matrices of A and B.

For the solution to this exercise, see [A.2 Matrix Multiplication Solution](#).

Appendix A

Exercise Solutions

A.1 Matrix Transposition Solution

Solution to the [8.1 Matrix Transposition Exercise](#):

```
001 // tranposeMatrix.c
002 #include <stdio.h>
003 #include <stdlib.h>
004 #include "CL/cl.h"
005
006 #define ROWS 16
007 #define COLS 16
008
009 int main()
010 {
011
012     cl_float *inputMatrix;
013     cl_float *results;
014     cl_uint width = COLS;
015     cl_uint height = ROWS;
016
017     // OpenCL host variables
018     cl_uint num_devs_returned;
019     cl_context_properties properties[3];
020     cl_device_id device_id;
021     cl_int err;
022     cl_platform_id platform_id;
023     cl_uint num_platforms_returned;
024     cl_context context;
025     cl_command_queue command_queue;
026     cl_program program;
027     cl_kernel kernel;
028     cl_mem input_buffer, output_buffer;
029     size_t global[2];
030
031     // variables used to read kernel source file
032     FILE *fp;
033     long filelen;
034     long readlen;
035     char *kernel_src; // char string to hold kernel source
036
037
038     // initialize inputMatrix with some data and print it
039     int x,y;
040     int data=0;
041
042     inputMatrix = malloc(sizeof(cl_float)*width*height);
043     results = malloc(sizeof(cl_float)*width*height);
044     printf("Input Matrix\n");
045     for(y=0;y<height;y++)
046     {
047         for(x=0;x<width;x++)
048         {
049             inputMatrix[y*height+x]= data;
050             results[y*height+x]=0;
051             data++;
052
053             printf("%.2f, ",inputMatrix[y*height+x]);
054         }
055         printf("\n");
056     }
057
058     // read the kernel
059     fp = fopen("transposeMatrix_kernel.cl","r");
060     fseek(fp,0L, SEEK_END);
061     filelen = ftell(fp);
```

```

062     rewind(fp);
063
064     kernel_src = malloc(sizeof(char)*(filelen+1));
065     readlen = fread(kernel_src,1,filelen,fp);
066     if(readlen!= filelen)
067     {
068         printf("error reading file\n");
069         exit(1);
070     }
071
072     // ensure the string is NULL terminated
073     kernel_src[filelen+1]='\0';
074
075
076 // Insert OpenCL host source here ----
077
078     // get a platform id
079     err = clGetPlatformIDs(1,&platform_id,&num_platforms_returned);
080
081     if (err != CL_SUCCESS)
082     {
083         printf("Unable to get Platform ID. Error Code=%d\n",err);
084         exit(1);
085     }
086
087     err = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_CPU, 1,
088                         &device_id, &num_devs_returned);
089     if (err != CL_SUCCESS)
090     {
091         printf("Unable to get Device ID. Error Code=%d\n",err);
092         exit(1);
093     }
094
095     // context properties list - must be terminated with 0
096     properties[0]= CL_CONTEXT_PLATFORM;
097     properties[1]= (cl_context_properties) platform_id;
098     properties[2]= 0;
099
100     // create context
101     context = clCreateContext(properties, 1, &device_id, NULL, NULL, &err);
102     if (err != CL_SUCCESS)
103     {
104         printf("Unable to create context. Error Code=%d\n",err);
105         exit(1);
106     }
107
108     // create command queue
109     command_queue = clCreateCommandQueue(context,device_id, 0, &err);
110     if (err != CL_SUCCESS)
111     {
112         printf("Unable to create command queue. Error Code=%d\n",err);
113         exit(1);
114     }
115
116     // create program object from source.
117     // kernel_src contains source read from file earlier
118     program = clCreateProgramWithSource(context, 1 , (const char **)
119                                       &kernel_src, NULL, &err);
120     if (err != CL_SUCCESS)
121     {
122         printf("Unable to create program object. Error Code=%d\n",err);
123         exit(1);

```

```

124     }
125
126     err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
127     if (err != CL_SUCCESS)
128     {
129         printf("Build failed. Error Code=%d\n", err);
130
131         size_t len;
132         char buffer[2048];
133         // get the build log
134         clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG,
135                               sizeof(buffer), buffer, &len);
136         printf("--- Build Log -- \n %s\n",buffer);
137         exit(1);
138     }
139
140     kernel = clCreateKernel(program, "matrixTranspose", &err);
141     if (err != CL_SUCCESS)
142     {
143         printf("Unable to create kernel object. Error Code=%d\n",err);
144         exit(1);
145     }
146
147     // create buffer objects to input and output args of kernel function
148     input_buffer =
149         clCreateBuffer(context,CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,
150                       sizeof(cl_float) * ROWS*COLS, inputMatrix, NULL);
151     output_buffer = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
152                                   sizeof(cl_float) * ROWS*COLS, NULL ,NULL);
153
154     // set the kernel arguments
155     if ( clSetKernelArg(kernel, 0, sizeof(cl_mem), &output_buffer) ||
156         clSetKernelArg(kernel, 1, sizeof(cl_mem), &input_buffer) ||
157         //clSetKernelArg(kernel, 2, sizeof(cl_uint), &width) ||
158         clSetKernelArg(kernel, 2, sizeof(cl_uint), &width) != CL_SUCCESS)
159     {
160         printf("Unable to set kernel arguments. Error Code=%d\n",err);
161         exit(1);
162     }
163
164     // set the global work dimension size
165     global[0]= width;
166     global[1]= height;
167
168     // Enqueue the kernel object with
169     // Dimension size = 2,
170     // global worksize = global,
171     // local worksize = NULL - let OpenCL runtime determine
172     // No event wait list
173     err = clEnqueueNDRangeKernel(command_queue, kernel, 2, NULL,
174                                  global, NULL, 0, NULL, NULL);
175     if (err != CL_SUCCESS)
176     {
177         printf("Unable to enqueue kernel command. Error Code=%d\n",err);
178         exit(1);
179     }
180
181     // wait for the command to finish
182     clFinish(command_queue);
183
184     // read the output back to host memory
185     err = clEnqueueReadBuffer(command_queue, output_buffer, CL_TRUE, 0,

```

```
186             sizeof(cl_float)*width*height, results, 0, NULL, NULL);
187     if (err != CL_SUCCESS)
188     {
189         printf("Error enqueueing read buffer command. Error Code=%d\n",err);
190         exit(1);
191     }
192
193
194     // print out the transposed matrix
195     printf("\nTransposed Matrix \n");
196     for(y=0;y<height;y++)
197     {
198         for(x=0;x<width;x++)
199         {
200             printf("%.2f , ",results[y*height+x]);
201         }
202         printf("\n");
203     }
204
205     // clean up
206     clReleaseMemObject(input_buffer);
207     clReleaseMemObject(output_buffer);
208     clReleaseProgram(program);
209     clReleaseKernel(kernel);
210     clReleaseCommandQueue(command_queue);
211     clReleaseContext(context);
212     free(kernel_src);
213     free(inputMatrix);
214     free(results);
215     return 0;
216 }
```

A.2 Matrix Multiplication Solution

Solution to the [8.2 Matrix Multiplication Exercise](#):

```
001 // ----- multMatrix.c -----
002
003 #include <stdio.h>
004 #include <stdlib.h>
005 #include "CL/cl.h"
006
007 #define ROWS 2048
008 #define COLS 2048
009
010 #define BLOCK_SIZE 8
011 int main()
012 {
013
014     cl_float *inputMatrix1;
015     cl_float *inputMatrix2;
016     cl_float *results;
017     cl_uint width = COLS;
018     cl_uint height = ROWS;
019
020     // OpenCL host variables
021     cl_uint num_devs_returned;
022     cl_context_properties properties[3];
023     cl_device_id device_id;
024     cl_int err;
025     cl_platform_id platform_id;
026     cl_uint num_platforms_returned;
027     cl_context context;
028     cl_command_queue command_queue;
029     cl_program program;
030     cl_kernel kernel;
031     cl_mem input_buffer1, input_buffer2, output_buffer;
032     size_t global[2];
033     size_t local[2];
034
035     // variables used to read kernel source file
036     FILE *fp;
037     long filelen;
038     long readlen;
039     char *kernel_src; // char string to hold kernel source
040
041
042     // initialize inputMatrix with some data and print it
043     int x,y;
044     int data=0;
045
046     inputMatrix1 = malloc(sizeof(cl_float)*width*height);
047     inputMatrix2 = malloc(sizeof(cl_float)*width*height);
048     results = malloc(sizeof(cl_float)*width*height);
049
050     for (y=0;y<height;y++)
051     {
052         for (x=0;x<width;x++)
053         {
054             inputMatrix1[y*height+x]= data;
055             inputMatrix2[y*height+x]= data;
056             results[y*height+x]=0;
057             data++;
058         }
059     }
060 }
061
```



```

062     // read the kernel
063     fp = fopen("multMatrix_kernel.cl","r");
064     fseek(fp,0L, SEEK_END);
065     filelen = ftell(fp);
066     rewind(fp);
067
068     kernel_src = malloc(sizeof(char)*(filelen+1));
069     readlen = fread(kernel_src,1,filelen,fp);
070     if(readlen!= filelen)
071     {
072         printf("error reading file\n");
073         exit(1);
074     }
075
076     // ensure the string is NULL terminated
077     kernel_src[filelen+1]='\0';
078
079
080 // OpenCL host source starts here ----
081
082     // get a platform id
083     err = clGetPlatformIDs(1,&platform_id,&num_platforms_returned);
084
085     if (err != CL_SUCCESS)
086     {
087         printf("Unable to get Platform ID. Error Code=%d\n",err);
088         exit(1);
089     }
090
091     err = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_CPU, 1, &device_id, &num_devs_returned);
092     if (err != CL_SUCCESS)
093     {
094         printf("Unable to get Device ID. Error Code=%d\n",err);
095         exit(1);
096     }
097
098     // context properties list - must be terminated with 0
099     properties[0]= CL_CONTEXT_PLATFORM;
100     properties[1]= (cl_context_properties) platform_id;
101     properties[2]= 0;
102
103     // create context
104     context = clCreateContext(properties, 1, &device_id, NULL, NULL, &err);
105     if (err != CL_SUCCESS)
106     {
107         printf("Unable to create context. Error Code=%d\n",err);
108         exit(1);
109     }
110
111     // create command queue
112     command_queue = clCreateCommandQueue(context,device_id, 0, &err);
113     if (err != CL_SUCCESS)
114     {
115         printf("Unable to create command queue. Error Code=%d\n",err);
116         exit(1);
117     }
118
119     // create program object from source. kernel_src contains
120     // source read from file earlier
121     program = clCreateProgramWithSource(context, 1 , (const char **) &kernel_src, NULL, &err);
122     if (err != CL_SUCCESS)
123     {

```

```

124         printf("Unable to create program object. Error Code=%d\n",err);
125         exit(1);
126     }
127
128     err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
129     if (err != CL_SUCCESS)
130     {
131         printf("Build failed. Error Code=%d\n", err);
132
133         size_t len;
134         char buffer[4096];
135         // get the build log
136         clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer);
137         printf("--- Build Log -- \n %s\n",buffer);
138         exit(1);
139     }
140
141     //kernel = clCreateKernel(program, "multMatrix", &err);
142     kernel = clCreateKernel(program, "multMatrixSimple", &err);
143     if (err != CL_SUCCESS)
144     {
145         printf("Unable to create kernel object. Error Code=%d\n",err);
146         exit(1);
147     }
148
149     // create buffer objects to input and output args of kernel function
150     input_buffer1 = clCreateBuffer(context,CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR, sizeof(cl_float), NULL, &err);
151     input_buffer2 = clCreateBuffer(context,CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR, sizeof(cl_float), NULL, &err);
152     output_buffer = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(cl_float) * ROWS*COLS, NULL, &err);
153     // set the kernel arguments
154     if ( clSetKernelArg(kernel, 0, sizeof(cl_mem), &output_buffer) ||
155         clSetKernelArg(kernel, 1, sizeof(cl_mem), &input_buffer1) ||
156         clSetKernelArg(kernel, 2, sizeof(cl_mem), &input_buffer2) ||
157         clSetKernelArg(kernel, 3, sizeof(cl_uint), &width) ||
158         clSetKernelArg(kernel, 4, sizeof(cl_uint), &height) != CL_SUCCESS)
159     {
160         printf("Unable to set kernel arguments. Error Code=%d\n",err);
161         exit(1);
162     }
163
164     // set the global & local work size
165     global[0]= width;
166     global[1]= height;
167
168     local[0]=BLOCK_SIZE;
169     local[1]=BLOCK_SIZE;
170
171     // Enqueue the kernel object with
172     // Dimension size = 2,
173     // global worksize = global,
174     // local worksize = local
175     // No event wait list
176     err = clEnqueueNDRangeKernel(command_queue, kernel, 2, NULL, global,local, 0, NULL, NULL);
177     if (err != CL_SUCCESS)
178     {
179         printf("Unable to enqueue kernel command. Error Code=%d\n",err);
180         exit(1);
181     }
182
183     // wait for the command to finish
184     clFinish(command_queue);
185

```

```

186     // read the output back to host memory
187     err = clEnqueueReadBuffer(command_queue, output_buffer, CL_TRUE, 0, sizeof(cl_float)*width
188     if (err != CL_SUCCESS)
189     {
190         printf("Error enqueueing read buffer command. Error Code=%d\n",err);
191         exit(1);
192     }
193
194     // clean up
195     clReleaseMemObject(input_buffer1);
196     clReleaseMemObject(input_buffer2);
197     clReleaseMemObject(output_buffer);
198     clReleaseProgram(program);
199     clReleaseKernel(kernel);
200     clReleaseCommandQueue(command_queue);
201     clReleaseContext(context);
202
203 // ---- End of OpenCL host portion
204
205
206 //  uncomment this block to print out matrix results
207 /*
208     printf("\nMatrix A\n");
209     for(y=0;y<height;y++)
210     {
211         for(x=0;x<width;x++)
212         {
213             printf("%.2f  ",inputMatrix1[y*height+x]);
214         }
215         printf("\n");
216     }
217
218     printf("\nMatrix B\n");
219     for(y=0;y<height;y++)
220     {
221         for(x=0;x<width;x++)
222         {
223             printf("%.2f  ",inputMatrix2[y*height+x]);
224         }
225         printf("\n");
226     }
227     // print out the transposed matrix
228     printf("\n Matrix A + Matrix B \n");
229     for(y=0;y<height;y++)
230     {
231         for(x=0;x<width;x++)
232         {
233             printf("%.2f  ",results[y*height+x]);
234         }
235         printf("\n");
236     }
237 */
238     free(kernel_src);
239     free(inputMatrix1);
240     free(inputMatrix2);
241     free(results);
242     return 0;
243 }

```

