A Beginner's Guide to Computational Computing on the AMD Embedded G-Series APU with OpenCL™

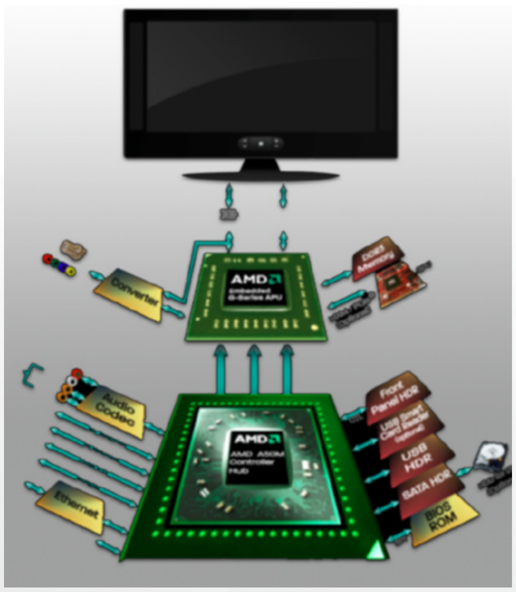
Contents

Abstract

Combining a low power CPU with a multicore high-performance GPU on the AMD Embedded G-Series APU (Accelerated Processing Unit) opens up new processing opportunities for embedded applications. But how do you take full advantage of the processing power? What software is available and what can it do? How does OpenCL™ help you get the most processing performance possible from the AMD G-Series APU? This paper provides an introduction to the AMD G-Series APU and operating systems that support these devices. It also introduces the reader to the OpenCL framework and programming model and contains an in-depth tutorial on working with AMD's Accelerated Parallel Processing Software Development Kit (AMD APP SDK) including creating your first OpenCL program.

APU Computing in Embedded Applications

For some time now, software developers have been using Graphics Processing Units (GPUs) for general purpose processing, often referred to as General Purpose computation on Graphics Processing Units (GPGPUs). The GPUs used for this purpose are usually high-performance multi-core processors capable of very high computation and data throughput. Once specially designed for computer graphics and difficult to program, today's GPUs are general-purpose parallel processors with support for accessible programming interfaces and industry-standard languages such as C and C++. Developers who port their applications to multi-core GPUs often achieve performance increases of orders of magnitude versus optimized CPU-only implementations.



Great advances in processor technology and integration now enable a new class of processors that combine computational processing with graphics processing on a single chip to create an Accelerated Processing Unit (APU). The AMD Embedded G-Series platform is the world's first integrated circuit to combine a low-power CPU and a discrete-level GPU into a single embedded APU. This unprecedented level of graphics integration builds a new foundation for high-performance multi-media content delivery in a small form factor and power efficient platform for a broad range of embedded designs.

Based on a power-optimized core, the AMD Embedded G-Series APU delivers high levels of performance in a compact BGA package that is ideal for low power designs in embedded applications such as digital signage, set-top-box, IP-TV, thin client, information kiosk, point-of-sale, casino gaming, media servers, and industrial control systems.

The AMD G-Series APU brings many advantages to your application:

- DirectX® 11 support lets you enjoy awesome graphics performance, stunning 3D visual effects and dynamic interactivity.
- Advanced discrete-level GPU with OpenGL (Open Graphics Language) support for graphics programming, and OpenCL (Open Computing Language) support for computational programming, in an integrated device provides the ideal platform on which to build the designs of tomorrow, today.
- High level graphics performance/watt through advanced graphics and hardware acceleration delivering over 3X performance per watt over previous generations.
- The integration of the APU reduces the footprint of a traditional three-chip platform to two

chips - the APU and its companion controller hub. This simplifies the design, requiring fewer board layers and a smaller power supply, further driving down system costs.

- APU configurations are available with single or dual x86 cores, at 4.5W, 9W or 18W thermal design power (TDP), and two levels of graphics and video performance. The AMD G-T16R APU with an average power of only 2.3 Watts[1] enables very small form factor, fan-less and portable applications.
- Selective models, such as the G-T56N and G-T40N, have additional boost capability enabled by AMD Turbo CORE technology without additional power draw.

Operating Systems

Most of the applications best suited to a processor of the AMD G-Series APU capability need to work in a deterministic environment; when an event occurs, the processor needs to respond immediately and without hesitation. These situations call for an operating system that has real-time capabilities. In the case of the AMD G-Series APU, there are several choices.

ThreadX

For deterministic real-time operating system (RTOS) support, ThreadX from Express Logic is a good option. ThreadX is a leading RTOS that has a very small memory footprint and fast response times.

ThreadX is implemented as a C library. Only the features used by the application are brought into the final image. The minimal footprint of ThreadX on an AMD G-Series APU can be in the range of a few hundred kilobytes.

The fast response time of ThreadX helps your application respond to external events with a minimal amount of latency. A high priority thread starts responding to an external event on the order of the time it takes to perform a highly optimized ThreadX context switch, about 20 cycles.

ThreadX requires as little as 300 cycles to initialize and start scheduling application threads. This is important for consumer and medical devices that simply can't afford a long boot time.

The following are key highlights of the advanced technology in ThreadX:

- Small, fast picoKernel architecture
- Dynamically downloadable application modules
- Memory protection for downloadable modules
- Automatic scaling (small footprint)
- Preemptive and cooperative scheduling
- Flexible thread priority support (32 - 1024 priority levels)
- Dynamic system object creation
- Unlimited number of system objects
- Optimized interrupt handling
- Preemption-Threshold™
- Priority inheritance
- Event-Chaining™
- Fast software timers
- Flexible memory utilization

Windows® Embedded

Thousands of embedded devices have been built with Windows® Embedded products, from portable ultrasound machines to GPS devices and from ATMs to devices that power large construction machinery. With comprehensive features, easy-to-use tools, free evaluation kits and access to a large network of community support, Windows Embedded may help yield faster time-to-market and decreased embedded development costs.

AMD has developed a board support package for the AMD G-Series APU that is available in Windows Embedded Compact 7 (formerly CE) and Windows Embedded offerings. Windows Embedded provides both a development host and a deployable platform for many embedded applications.

Linux®

Linux® provides an alternative open source solution that is widely used in many applications needing real-time response. Since Linux version 2.6, there has been real-time capability built into the operating system. Linux brings with it the advantage of a large library of software from tools to applications to aid in your software development. Ubuntu and other Linux distributions run on the AMD G-Series APU.

Getting Started with Heterogeneous Computing

In addition to the operating system choices, you need to evaluate what is available for graphics and computational software that will run on the processor. OpenGL and OpenCL Application Programming Interfaces (API) are supported within the hardware, with AMD providing tools for your graphics and computational programming that work with Windows Embedded, Linux, and ThreadX. OpenGL and OpenCL are results of the efforts of the Khronos Group. The Khronos Group is a not-for-profit industry consortium that creates open standards for the authoring and acceleration of parallel computing, graphics, dynamic media, computer vision and sensor processing on a wide variety of platforms and devices.

OpenGL

Since its introduction in 1992, OpenGL has become a widely adopted 2D and 3D graphics API, bringing thousands of applications to a wide variety of computer platforms. It is window-system and operating-system independent as well as network-transparent. OpenGL enables developers of software for PC, workstation, and supercomputing hardware to create high-performance, visually compelling graphics software applications, in markets such as CAD, content creation, energy, entertainment, game development, manufacturing, medical, and virtual reality. OpenGL exposes developers to all of the features of the latest graphics hardware.

OpenCL™

OpenCL is the first open and royalty-free programming standard for accelerating general-purpose computations on heterogeneous systems. OpenCL allows programmers to preserve their expensive source code investment and easily target multi-core CPUs, GPUs, and the new APUs. OpenCL helps to improve speed and responsiveness for a wide spectrum of applications in numerous market categories from gaming and entertainment to scientific and medical software.

Developed in an open standards committee with representatives from major industry vendors, OpenCL gives users what they have been demanding: a cross-vendor, non-proprietary solution for accelerating their applications on CPUs, GPUs and APUs.

AMD, an early supporter of OpenCL and leading innovator and provider of high-performance CPUs, GPUs, and APUs, is uniquely positioned to offer an extensive acceleration platform for OpenCL.

Developer and technology partners have created several applications, libraries and technology demonstrations taking advantage of AMD Accelerated Parallel Processing (APP) technology. These applications and demonstrations showcase how AMD APP technology and OpenCL can help improve performance and overall computational efficiency.

OpenCL™ versus OpenGL

Sometimes there is confusion in choosing between OpenCL and OpenGL. What features make OpenCL unique to choose over OpenGL for calculations? To answer this question, it is important to understand that OpenCL and OpenGL are more complementary than they are competing technologies.

OpenGL is a graphics API; therefore, the work you do in it has to be done from that perspective. It is helpful to structure your code and data in terms of graphics acceleration and rendering functions, then determine how to deal with your data in terms of attributes, frame buffers, textures, and polygons.

OpenCL is NOT a graphics API; it is a computation API, and therefore is more efficient at performing highly parallel computations on GPGPU hardware. OpenCL gives you access to memory levels that are implicit with regard to OpenGL. Certain memory can be shared between threads resulting in OpenCL being able to access and manipulate OpenGL data (vertex and texture buffers) directly inside GPU memory without having to transfer data back and forth.

While you can use OpenGL to do arbitrary computations, in many cases a high overall performance will be achieved by using a combination of OpenGL and OpenCL and partitioning your code to take advantage of each API's unique strengths. For example, OpenCL can help enhance and accelerate graphics-intensive applications originally written and optimized for OpenGL by offloading the front-end setup and calculations, such as transformation and lighting to OpenCL while keeping the back-end rendering and display algorithms in OpenGL. If you are limited to older hardware that can't run OpenCL, you may be limited to using OpenGL for running applications that contain a significant amount of both graphics and computational tasks.

Types of Applications

Many applications can benefit from the APU architecture. These applications are usually graphics-intensive or have need for processing power beyond what is typically available in a traditional general-purpose processor.

- Digital Signage: Graphics display systems that process large images and overlays, plus many have interactive touch inputs and GPS functionality.
- Thin Client: Similar requirements to digital signage but on a smaller scale and with increased network connectivity.
- Information Kiosk: Similar needs to digital signs plus connectivity and a variety of human interfaces.
- Set-Top Box (STB): Scalable choices for various levels of CPU performance, power efficiency, and visual experience.
- IP-TV: The same requirements found in a set-top box, plus network connectivity for content.

- Point-of-Sale: Terminals process large images and overlays with the potential to include video.
- Casino Gaming: Video imaging on many games takes a large amount of processing.
- Media Servers: Complex data routing and media processing for applications such as Digital Rights Management (DRM) and security.
- Industrial Control Systems: Many of these systems now have complex display systems with easy-to-use human-machine interfaces.

Basic Concepts of OpenCL™

Before starting the tutorial part of this document, let's review some basic concepts of OpenCL including the overall framework and programming model.

Platform Model

OpenCL views heterogeneous processors through an abstract, hierarchical platform model. In this model, a Host coordinates execution, transferring data to and from an array of Compute Devices. Each Compute Device is composed of an array of Compute Units, and each Compute Unit is composed of an array of Processing Elements. One of OpenCL's strengths is that this model does not specify exactly what hardware constitutes a Compute Device. Thus, a Compute Device may be a discrete GPU, such as the AMD Radeon™ HD 5870 GPU, a discrete CPU, such as the AMD Phenom™ II x4 processor, or integrated into an APU such as an AMD G-Series APU that contains both a multi-core x86 CPU and an AMD Radeon GPU. Note that multiple cores within a CPU or GPU still only constitute a single Device whereas each CPU or GPU core constitutes a compute unit. A Platform is defined as "a Host plus a collection of Devices managed by the OpenCL framework that allow an application to share resources and execute Kernels on Devices in the Platform". The OpenCL platform model (see Figure 1) is designed to present a uniform view of many different kinds of Compute Devices or parallel processors contained within a single platform.

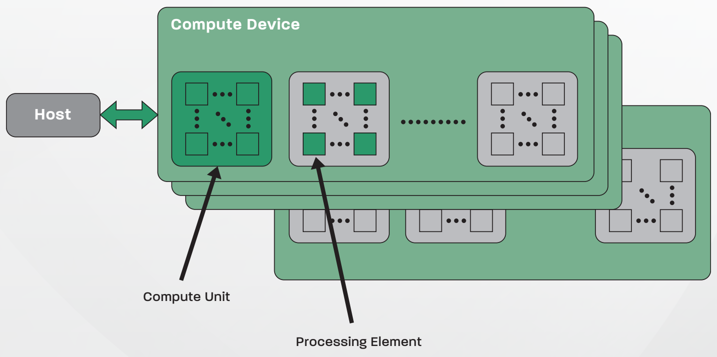


Figure 1: OpenCL™ Platform Model

Execution Model

OpenCL has a flexible execution model that incorporates both task and data parallelism. Tasks themselves are comprised of data-parallel Kernels, which apply a single function over a range of data elements in parallel. The movement of data between the Host and the Compute Devices, as well as the scheduling of OpenCL tasks for execution, is coordinated via Command Queues. Kernels and Command Queues will be explained further in this section.

Kernels

A Kernel can be thought of as a specific OpenCL function (similar to a C function) that is executed on a Compute Device and is called from the Host program. As mentioned, OpenCL Kernels provide data parallelism. The Kernel execution model is based on a hierarchical abstraction of the computation being performed. OpenCL Kernels are executed over an index space (data array), which can be 1, 2 or 3 dimensional. For every element of the Kernel index space, a Work Item will be executed. Work Items are the smallest execution entity and all Work Items within a Kernel perform the same function (but on different data elements).

The Kernel index space is regularly subdivided into Work Groups. Work Groups are arranged into a grid that is based on the amount of memory allocated to the Compute Device (CPU or GPU) and facilitate communication between Work Items. Both Work Items and Work Groups have unique IDs that can be accessed by the Kernel. The ID's are then used to distinguish the data to be processed by each Work Group or Work Item.

The Work Items may only communicate and synchronize locally, within a Work Group, via a barrier mechanism. This provides scalability, traditionally the bane of parallel programming. Because communication and synchronization at the finest granularity are restricted in scope, the OpenCL runtime has great freedom in how Work Items are scheduled and executed.

The data parallel execution model of OpenCL allows programmers to efficiently define a set of instructions that will be executed on a large number of data items at the same time. Figure 2 illustrates the differences between implementing code for processing an image made up of [a] pixels x [b] pixels using a traditional "for" loop and implementing the same function in an OpenCL Kernel utilizing data parallelism. The traditional "for" loop reads in data for both [a] and [b] sequentially and then performs a computation. The OpenCL Kernel streamlines this operation by reading in all of [a], multiplying by a value from [b], and then writing the output. For example, if the image were 1024x1024 pixels, then initiating one Kernel execution per pixel would result in 1024x1024 = 1,048,576 Kernel executions.

```
void
trad_mul(int n,
           const float *a,
           const float *b,
           float*c)

{
   int i;
   for (i=0; i<n; i++)
     c[i]=a[i] * b[i];
}
```

Traditional loop (scalar)

```
kernel void
dp_mul(global const float *a,
           global const float *b,
           global float*c)

{
   int id= get_global _id (0) ;

   c[id]=a[id] * b[id];

} // execute over "n" work-items
```

Data parallelism in OpenCL

Figure 2: Traditional Scalar Loop vs. Data Parallelism in OpenCL™

Command Queues

While the division of a Kernel into Work Items and Work Groups supports data-parallelism, task-parallelism is achieved by "scheduling" Kernels via OpenCL Command Queues. An OpenCL Command Queue is created by the developer through an API call, and associated with a specific Compute Device. A developer may target multiple OpenCL Compute Devices simultaneously by creating multiple Command Queues.

To execute a Kernel, the Kernel must first be "pushed" onto a particular Command Queue. This is known as "enqueueing a Kernel" and is done asynchronously, so that the host program may enqueue many different Kernels without waiting for any of them to complete. When enqueueing a Kernel, the developer optionally specifies a list of events that must occur (such as specific memory transactions) before the Kernel executes. In this way, a developer can specify dependencies between Kernel executions and memory transfers in a particular Command Queue or between Command Queues themselves. Figure 3 illustrates the concept of a task graph. The arrows indicate dependencies between tasks. For example, Kernel A will not execute until Write A and Write B have finished, and Kernel D will not execute until Kernel B and Kernel C have finished.

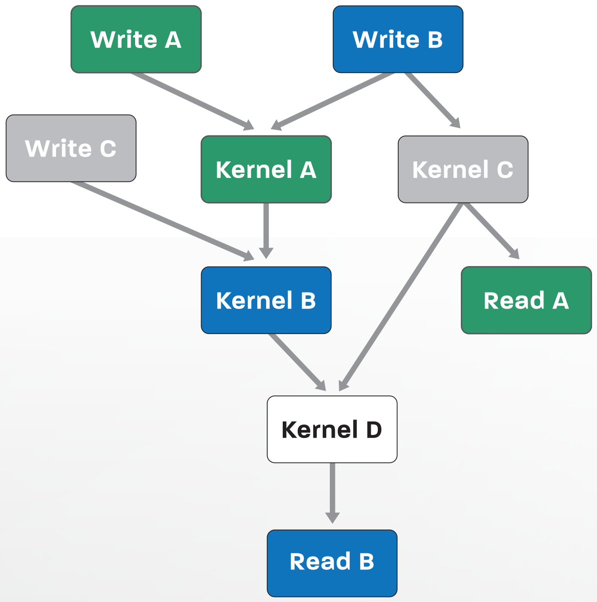


Figure 3: Task Graph Illustrating Task Parallelism within a Command Queue

The ability to construct arbitrary task graphs is a powerful way of constructing task-parallel applications. The OpenCL runtime has the freedom to execute the task graph in parallel, as long as it respects the dependencies encoded in the task graph.

Developers are also free to construct multiple Command Queues, either for parallelizing an application across multiple compute devices, or for expressing more parallelism via completely independent streams of computation. OpenCL's ability to use both data and task parallelism simultaneously is a great benefit to parallel application developers, regardless of the hardware they are intended to run on.

Memory Model

OpenCL has a relaxed consistency memory model made up of four distinct memory regions as shown in Figure 4. Individual Work Items executing a Kernel have access to these four memory regions including private memory, local memory, constant memory, and global memory.

Each Compute Device has a global memory space, which is the largest memory space available to the Device, and typically resides in off-chip DRAM. There is also a read-only, limited-size constant memory space, that allows for efficient reuse of read-only parameters in a computation. Each compute unit on the Device has a local memory, which is typically on the processor die, and therefore has much higher bandwidth and lower latency than global memory. Local memory can be read and written by any Work Item in a Work Group, and thus allows for local communication between Work Groups. Additionally, attached to each processing element is a private memory, which is typically not used directly by programmers, but is used to hold data for each Work Item that does not fit in the processing element's registers.

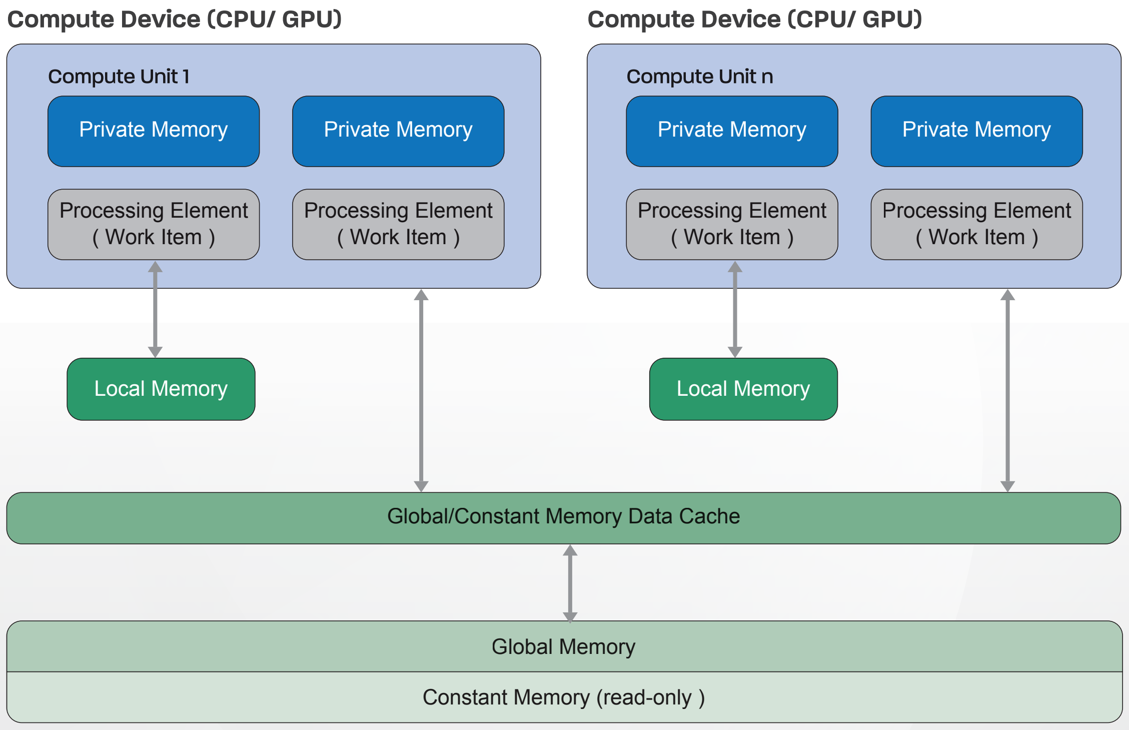

Compute Device (CPU/ GPU)
Compute Device (CPU/ GPU)
Compute Unit 1
Compute Unit n
Private Memory
Private Memory
Private Memory
Private Memory
Processing Element (Work Item)
Processing Element (Work Item)
Processing Element (Work Item)
Processing Element (Work Item)
Local Memory
Local Memory
Global/Constant Memory Data Cache
Global Memory
Constant Memory (read-only)

Figure 4: OpenCL™ Hierarchical Memory Model

As OpenCL has a relaxed consistency model, different Work Items may see a different view of global memory as the computation progresses. Within a Work Item, reads and writes to all memory spaces are consistently ordered, but between Work Items, synchronization is necessary in order to ensure consistency. This relaxed consistency model is a part of OpenCL's efforts to provide parallel scalability: parallel programs that rely on strong memory consistency for synchronization and communication may fail to execute in parallel, because memory ordering requirements force a serialization of such programs during execution, hindering scalability. Requiring explicit synchronization and communication between Work Items encourages programmers to write scalable code, helping to avoid the trap often seen in parallel programming where code looks parallel, but ends up executing in serial due to frequent and implicit synchronization induced by reliance on a strict memory ordering model.

Additionally, OpenCL views the global memory space of each Compute Device as private and separate from host memory. Moving data between Compute Devices and the host requires the programmer to manually manage communication between the host and the Compute Devices. This is done through the use of explicit memory reads and writes between devices.

The OpenCL™ Framework

As Figure 5 shows, the OpenCL Programming Framework is made up of several objects, including the programs, Kernels, memory objects, and Command Queues that define the OpenCL environment. This collection of objects is also known as the Context, and is one of the first items to be defined and initialized in an OpenCL program. Programs, Kernels, Buffers, Images, Events, and Command Queues, are all created in the context while Buffers, Images and Events are shared by all Devices in the context.

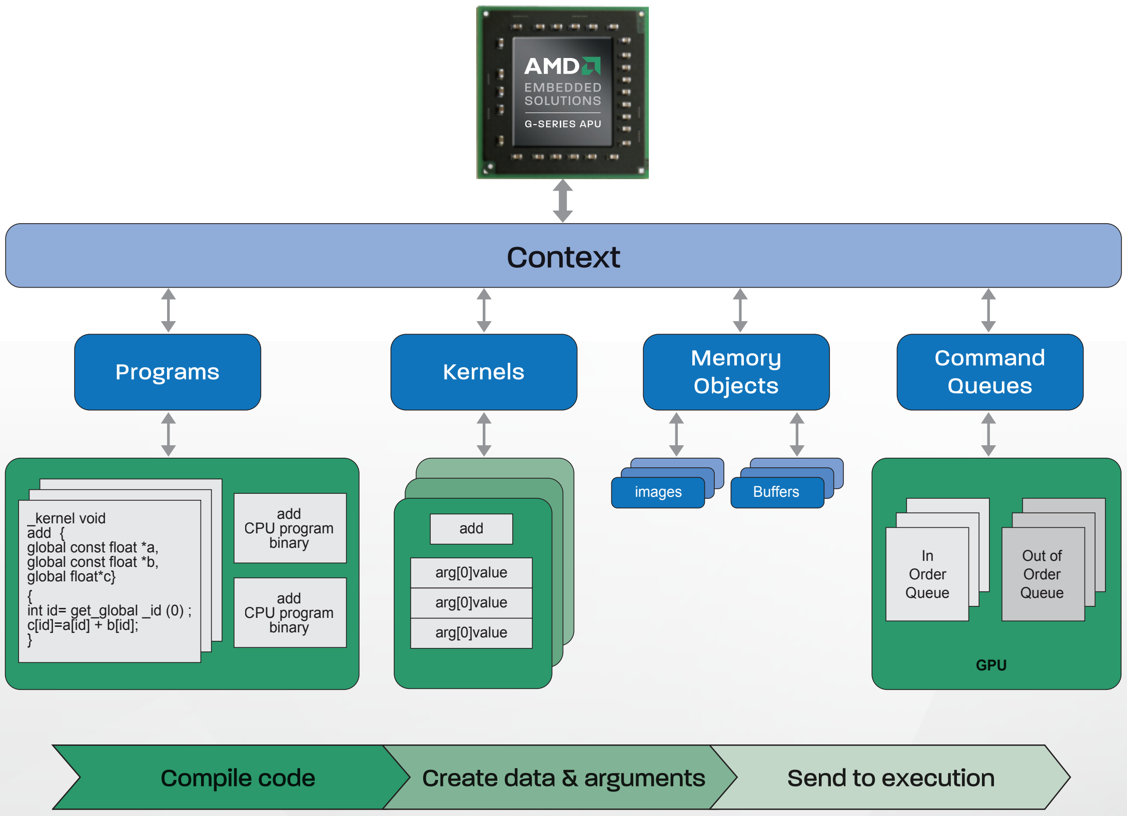


Figure 5: OpenCL™ Framework

OpenCL™ Programming Model

OpenCL programs are divided into three main parts:

1. A Host program (typically written in C/C++) that runs on the CPU and controls the OpenCL environment, or Context, as described above.
2. A C/C ++ header file that contains declarations for all of the functions that need to be visible to the Host program (as well as C++ bindings for the case where the Host is written in C++ as in the examples used in this tutorial).
3. One or more Kernels (written in OpenCL).

There is a basic program example used in this tutorial called “Template” that comes with the AMD APP SDK (AMD’s Software Development Kit for OpenCL) which demonstrates a common structure and flow to creating a complete OpenCL program. The basic programming flow is:

1. Write the Host code (C or C++)

a. Setup
   i. Query the runtime to determine which platforms are present.
   ii. Get devices (CPU + GPU). Note that multiple processing cores within a given device (CPU or GPU) are considered a single device.
   iii. Create a Context.
   iv. Create Command Queues for each Device.
   v. Instantiate all Kernels that will be used.

b. Allocate memory resources including images and buffers.

c. Create the program objects that will run on one or more associated devices.

d. Write data to each Device.

e. Submit the Kernels to the Command Queue for execution.

f. Read data from each Device back to the Host.

g. Clean up buffers, Kernels, Queues, etc.

2. Write the code for the Kernel(s) (OpenCL).

a. Create the Kernel.

b. Allocate Host vectors.

c. Initialize Host memory.

d. Create buffers for Kernel input and output.

e. Set up parameter values.

f. Execute the Kernel.

g. Copy results from the Device back to the Host.

3. Compile and Execute

a. Host program (including OpenCL headers and the OpenCL library) is compiled into an executable.

b. Kernel objects are compiled at runtime for each target Device (CPU or GPU).

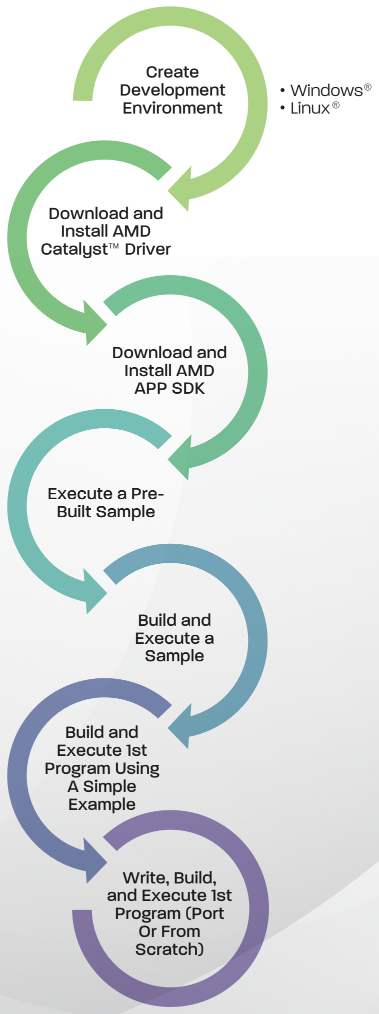
The relationships between Context(s), Device(s), Buffer(s), Program(s), Kernel(s), and Command Queue(s) and the syntax and use of OpenCL objects and functions are better understood by studying the source code of the "Template" sample used later in this tutorial.

A more detailed introduction to OpenCL can be found at
http://blogs.amd.com/developer/2009/09/15/amd-developer-inside-track-episode-2-opencl-introduction/

OpenCL™ Programming 101: A Step-by-Step Tutorial

AMD has extensive documentation available to help you get started with OpenCL on either a Windows or Linux development platform.

This section is intended to consolidate much of that information and help provide novice and expert programmers with a step-by-step guide to getting up and running quickly with OpenCL; from creating a development environment to getting your first OpenCL program compiled and running. While not a "programming tutorial", this tutorial takes a structured and disciplined approach to learning and reinforcing the basic concepts of OpenCL by working with and modifying a sample program.


Create Development Environment
• Windows®
• Linux®
Download and Install AMD Catalyst™ Driver
Download and Install AMD APP SDK
Execute a Pre-Built Sample
Build and Execute a Sample
Build and Execute 1st Program Using A Simple Example
Write, Build, and Execute 1st Program (Port Or From Scratch)

Step 1: Create a Development Environment

The first step in preparing to learn OpenCL is to create a proper development environment including supported AMD hardware and software.

While AMD supports Windows Vista®, Windows 7, and Windows 8, as well as most Linux releases, AMD officially tests its AMD APP (AMD Accelerated Parallel Processing) SDK under Windows 7, Windows 8 and under Linux releases Ubuntu 11.04 and later, OpenSUSE11, and RedHat Enterprise Linux 6. The AMD APP SDK is intended for AMD Radeon GPUs, both integrated in an APU and discrete graphics device, in conjunction with their corresponding AMD Catalyst™ driver (which provides run-time support).

In Linux®

The following instructions assume a clean Ubuntu installation with no prior installation of any AMD or other OpenCL-related software or drivers. If you have attempted to install any OpenCL-related software prior to using this tutorial, it is highly recommended to remove that software and start over with a clean system. Also, note that many of the steps require access to the command line interface. A convenient shortcut to opening a terminal window in Ubuntu is CTL+ALT+T.

1. Verify that you have an up-to-date development system that is fully compatible with AMD's APP SDK. This tutorial was written based on an AMD development platform consisting of an AMD G-Series G-T56N APU that contains a dual-core CPU combined with an integrated AMD Radeon HD 6320 GPU, running both Ubuntu 12.04 LTS with Linux 3.2.0-31-generic, and Windows 7. Click on Check/Install Updates in the Detailed System Settings window or open a terminal window and type:
   ```
   apt-get update
   apt-get upgrade
   ```

2. Users must be able to gain root access on the system to perform certain operations. A convenient method of obtaining root access on a per-command basis is to prepend any command with "sudo" such as "sudo make" (remember, you must enter your user password, NOT the root password). Note that gaining root access can be extremely dangerous for novice developers and should only be used as a last resort to get around errors such as files and/or folders that may be write-protected at the current user level.

3. Install additional Linux packages. Many of these packages are OpenGL libraries needed to build (i.e. "make") and run the entire AMD APP SDK (including the sample programs). Note: you must have an internet connection to do this. Use whatever method is most convenient for you (Synaptic, Ubuntu SW Center, etc.). Command-line instructions using Ubuntu's Advanced Packaging Tool (APT) are shown below:
   a. Install package mesa-common-dev (Mesa-specific OpenGL extensions):
      ```
      sudo apt-get install mesa-common-dev
      ```

   b. Install package libglu1-mesa-dev (Mesa OpenGL utility library development files. Includes headers and static libraries for compiling programs with GLU):
      ```
      sudo apt-get install libglu1-mesa-dev
      ```

   c. Install package freeglut3-dev (OpenGL Utility Toolkit containing libraries, headers, and a simple windowing API):
      ```
      sudo apt-get install freeglut3-dev
      ```

   d. Install the latest versions of the GNU C compiler and GNU C++ compiler (you will also need the build-essential package which contains various programs and utilities necessary for building packages compatible with Ubuntu Linux including gcc compiler, make, and other required tools. The build-essentials package will allow you to compile OpenCL and C/C++ software using standard C/C++ compilers):
      ```
      sudo apt-get install build-essential
      sudo reboot
      ```

4. If any boot issues arise related to detecting the AMD Radeon graphics hardware and/or loading the Ubuntu Desktop during the process of upgrading/updating your development system (including updating the Linux kernel), it may be necessary to reinstall the AMD Catalyst driver. To reinstall the existing AMD Catalyst driver from the command prompt, perform the following steps:
   a. Navigate to usr/share/ati

   b. Uninstall the current driver: sudo (or sudo sh) ./fglrx-uninstall.sh

   c. sudo reboot

   d. Navigate to home/user/Downloads

   e. Reinstall the current driver: sudo (or sudo sh) ./amd-driver-installer[version].run

f. sudo reboot

g. Note: Instructions for downloading and installing the latest AMD Catalyst driver are provided in Step 2 of this tutorial.

In Windows®

1. Verify the version and type of Windows you are running
2. Install Visual Studio 2010 Professional Edition and/or 2012 Professional or Premium Edition
3. Install Microsoft.NET framework version 2.0 (x64) or later Redistributable Package
4. Note: The Windows portion of this tutorial was created on an AMD development system running Microsoft Windows 7 with Microsoft Visual Studio 2010 Professional Edition SP1 Release, Microsoft Visual Studio 2012 Premium Edition, and Microsoft .NET Framework v. 4.5 SP1 Release. For information regarding Microsoft Visual Studio 2012 including backwards compatibility with Microsoft Visual Studio 2010, please refer to Microsoft's MSDN library at http://msdn.microsoft.com/en-us/library/dd831853.

Step 2: Download and Install the Latest AMD Catalyst™ Driver

You must install the AMD Catalyst 12.4 or later graphics drivers before installing the AMD APP SDK 2.7 or later as critical parts of the AMD OpenCL solution are now contained within the AMD Radeon graphics drivers (including GPU runtime support). This tutorial was created based on AMD Catalyst 12.12 and AMD APP SDK 2.8.

In Linux®

1. Browse to the AMD Accelerated Processing (APP) SDK with OpenCL Support home page on the AMD Developer Central website (http://developer.amd.com/tools/hc/AMDAPPSDK/Pages/default.aspx).
2. Click on Downloads and then Download AMD Catalyst Drivers under the Tested Drivers heading.
3. Make the proper selections according to your system configuration and then click on Display Results to download the driver package (delivered as a gzip compressed tar archive or a zip archive). There is also a utility available for download on this page that will automatically detect and install the proper driver. The description of the package should reference version 12.4 or later of the AMD Catalyst driver.
4. Extract and execute the .run file.
5. When the .run file completes, a log file of the installation called fglrx-install.log will be created in /usr/share/ati.
6. Create a symlink (symbolic link) under /usr/lib (this is required because the GPU runtime is installed from the AMD Catalyst driver):
   sudo ln –s libOpenCL.so.1 libOpenCL.so
   a. Note: You will also set the environment variable "LD_LIBRARY_PATH" to point to

these files under Step 3: Download and Install the AMD APP SDK below.

b. Note: If upgrading the AMD Catalyst driver from a previous version, an already established symbolic link between these two libraries should not need to be recreated.
To view all symbolic links, type sudo ls –li from the command line.

7. An additional method of verifying that the AMD Kernel mode driver loaded correctly:
Enter fglrxinfo, which should display something similar to:
OpenGL vendor string: Advanced Micro Devices, Inc.
OpenGL renderer string: AMD Radeon HD 6320 Graphics
OpenGL version string: 4.2.11631 Compatibility Profile Context

In Windows®

1. Browse to the AMD Accelerated Processing (APP) SDK with OpenCL Support home page on the AMD Developer Central website (http://developer.amd.com/tools/hc/AMDAPPSDK/Pages/default.aspx).
2. Click on Downloads and then Download AMD Catalyst Drivers under the Tested Drivers heading.
3. Make the proper selections according to your system configuration and then click on Display Results to download the driver package (delivered as a gzip compressed tar archive). There is also a utility available for download on this page that will automatically detect and install the proper driver. The description of the package should reference version 12.12 or later of the AMD Catalyst driver.
4. Extract all files and run Setup (AMD Catalyst Install Manager).
5. Reboot.

Step 3: Download and Install the AMD APP SDK

AMD's APP SDK version 2.8 includes significant new functionality compared to previous versions including tools and libraries such as Aparapi, C++ Amp, and a preview of AMD's new Bolt C++ template library. Performing a standard installation of AMD APP SDK version 2.8 will install this additional software; however, this additional functionality is outside the scope of this beginner's tutorial and so will not be discussed further in this document.

Additional AMD documentation related to AMD APP SDK 2.8 such as Installation Notes, Developer Release Notes, Getting Started Guide, and FAQs can be found on AMD's APP SDK Documentation page at http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/documentation/ and in the docs folder after installing the SDK as discussed below.

In Linux®

1. Browse to the AMD Accelerated Processing (APP) SDK with OpenCL Support home page on the AMD Developer Central website (http://developer.amd.com/tools/hc/AMDAPPSDK/Pages/default.aspx).

2. Click on Downloads in the menu on the left.

3. Click on Current Version near the top of the page.

4. Click on AMD-APP-SDK-v2.8-lnx32.tgz for 32-bit systems or AMD-APP-SDK-v2.8-lnx64.tgz for 64-bit systems. These files are delivered as a gzip compressed tar archive.

5. Copy and extract/unzip to a location of your choice on your development platform.

6. Navigate to the location specified in the previous step and execute sudo ./Install-AMD-APP.sh. This automatically completes the following:
    a. Registers the Installable Client Driver (ICD).

    b. Globally sets most of the needed environmental variables which are then reflected to all users.

    c. Installs the Linux AMD APP SDK developer and sample files /binaries under /opt/AMDAPP .

    d. Installs the Linux AMD APP SDK runtime files under /usr/lib(64).
    Notes:
        i. If the error "path not found" while clearing the cache is generated on, ignore it.

        ii. If you want to install under a different path than /opt/AMDAPP, update the default-install_lnx.pl file with the path to your desired location.

7. Reboot per the recommendation of the Installer.

8. Check that the AMDAPPSDKROOT and LD_LIBRARY_PATH (for the OpenCL dynamic libraries) environment variables have been set in etc/profile by the installer.
    a. To determine the values of the environment variables, type:
    printenv AMDAPPSDKROOT
    printenv LD_LIBRARY_PATH

    If any of the above environment variables are not set (i.e. printenv doesn't return a value), then you should confirm that etc/profile was modified properly by the installer. If the following lines do not exist in etc/profile, this may be an indication of an installation issue. It is recommended that you check the log file and rerun the installation shell script as appropriate.

        - Navigate to the /etc folder
        - edit the file: sudo gedit profile
        - Check for the following lines:
        AMDAPPSDKROOT=/opt/AMDAPP (if using the installer default)
        LD_LIBRARY_PATH=/opt/AMDAPP/samples/opencl/SDKUtil/build/debug/x86_64 (for a 64-bit system)
        LD_LIBRARY_PATH=/opt/AMDAPP/samples/opencl/SDKUtil/build/debug/x86_32 (for a 32-bit system)

        Note: Due to a known issue where certain environment variables may be inadvertently overwritten during Xsession login in Ubuntu distributions, it is highly recommended that you implement the following workaround after the install script has added the LD_LIBRARY_PATH variable to etc/profile:
        Navigate to /etc/X11
        sudo gedit Xsession.options
        Change use-ssh-agent to no-use-ssh-agent

Save and close
reboot

9. The AMD APP SDK installer installs the following packages on your system by default (unless you choose to customize the install):
   a. AMD APP SDK CPU Runtime package.

   b. AMD APP SDK Developer package. This includes:
      i. the OpenCL compiler,

      ii. Pointers to the latest versions of the developer documentation. (See the AMD APP SDK v2 folder in the All Programs panel of Windows Start. This also contains links to the AMD Math Libraries.)

   c. AMD APP SDK v2 Samples package. This includes:
      i. sample applications

      ii. sample documentation.
      Note: Additional samples not included in the AMD APP SDK Samples package can be found at on AMD Developer Central at :
      http://developer.amd.com/sdks/AMDAPPSDK/samples/Pages/default.aspx

10. The AMD APP SDK installation creates the following folder structure as shown in the Figure 6:

```
/opt/AMDAPP
 ├──── docs
 ├──── include
 │        ├──── CL
 │        ├──── GL
 │        └──── OpenVideo
 │
 ├──── lib
 │        ├──── x86
 │        └──── x86_64
 ├──── make
 └──── samples
          ├────opencl
          │       ├──── benchmark
          │       ├──── bin
          │       │       ├──── x86
          │       │       └──── x86_64
          │       │   cl
          │       │       └──── app
          │       │               └──── sample folders
          │       ├──── cpp_cl
          │       ├──── SDKUtil
          │       └──── SyntaxHighlighting
          └────Aparapi
```

Figure 6: Ubuntu Linux®, AMD APP SDK Folder Structure After Installation

   a. lib - contains the base OpenCL runtime library to which the applications link on Windows systems, as well as import libraries to author CAL applications.

   b. include - contains the header files (as provided by Khronos) for the OpenCL and CAL runtimes.

   c. make – contains rules and definitions used by the make files for building the sample programs.

   d. samples – contains all of the sample program source files and executables including makefiles for building the samples.

   e. docs - contains developer documentation for the AMD APP SDK. Additional developer documentation is available online at: http://developer.amd.com/gpu/AMDAPPSDK/documentation/Pages/default.aspx

11. If you wish to review the sample applications at this time, browse to /opt/AMDAPP/samples/opencl/cl/app and open any of the folders associated with the sample you are interested in. Documentation for each sample can be found in the docs subfolder of each sample.
    a. Note: Developers familiar with earlier versions of the AMD APP SDK may notice that the sample "HelloWorld", missing from version 2.7, has been included again in version 2.8. While this program is very useful for developers new to OpenCL, the rest of this tutorial will focus on an alternate program called "Template".

In Windows®

1. If you have a previously installed version of the AMD APP SDK, you must uninstall it before continuing. It is recommended that you follow all of these steps and check each one thoroughly.
   a. Reboot and use Programs and Features (Windows Vista or Windows 7 SP1) to uninstall the prior SDK files.

   b. Manually remove any AMD APP directories under My Documents and under Program Files or Program Files (x86).

   c. Search for, and remove all atiocl*, and amdocl*, and OVDecode* files on the system.

   d. Ensure that the previously generated temporary folder is also deleted. Paths to this folder are:
      - Vista and Win7 SP1 32-bit: C:\AMD\SUPPORT\streamsdk_2-7_win732
      - Vista and Win7 SP1 64-bit: C:\AMD\SUPPORT\streamsdk_2-7_win764

2. Browse to the AMD Accelerated Processing (APP) SDK with OpenCL Support home page on the AMD Developer Central website (http://developer.amd.com/tools/hc/AMDAPPSDK/Pages/default.aspx).

3. Click on Downloads in the menu on the left.

4. Scroll to the Current Version heading near the middle of the page.

5. Click on AMD-APP-SDK-v2.8-Windows-32.exe (106 MB) for 32-bit systems or AMD-APP-SDK-v2.8-Windows-64.exe (180 MB) for 64-bit systems.

6. Copy, extract, and run from a location of your choice on your development platform.

7. Follow the prompts, choose “Express Install”, and accept the End User License Agreement.
    a. A temporary “Analyzing System” screen will appear. This program is detecting the type of graphics hardware and software currently installed in the system. If an older version of one of the components is already installed, a warning appears, indicating that the installation cannot continue before it is removed. In this case, use Program and Features (in Windows Vista or Windows 7 SP1) to remove the older version of the component named above the lower progress bar.

8. When completed, click to view the log file and confirm that the following packages have been installed:
    a. AMD APP SDK Developer
    b. AMD APP CPU SDK Runtime
    c. AMD APP SDK Samples
    d. C++ Amp
    e. Bolt
    f. Aparapi

9. The AMD APP SDK installation creates the following folder structure as shown in the Figure 7:

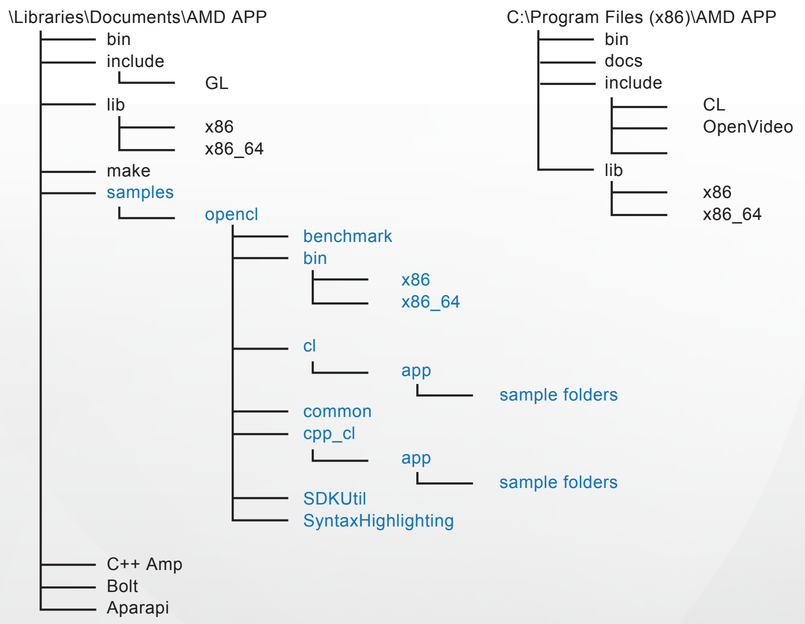
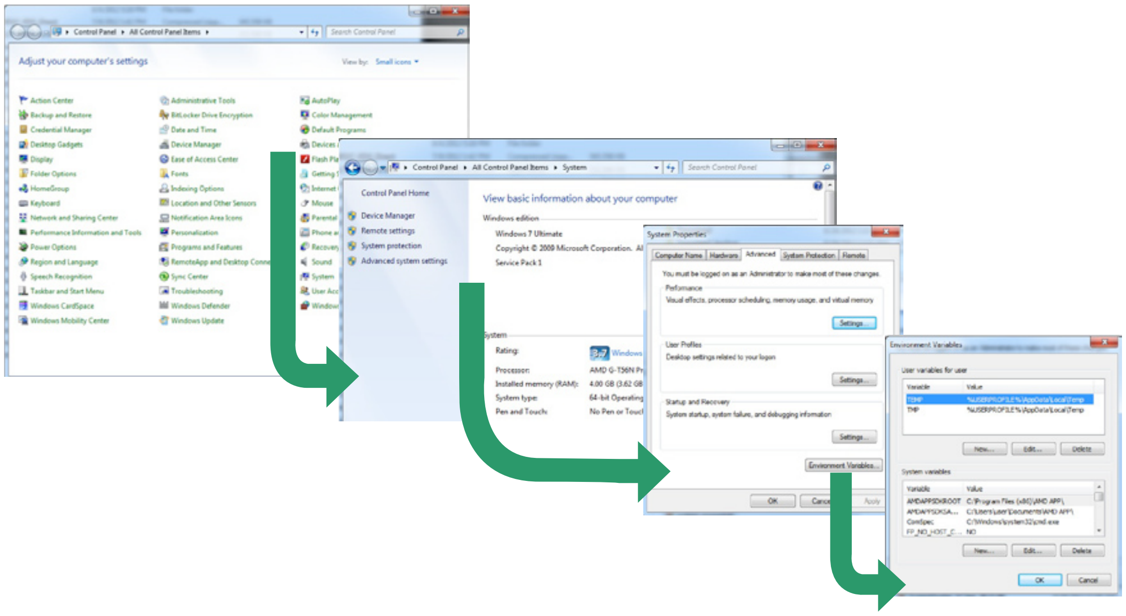


Figure 7: Windows® 7, AMD APP SDK Folder Structure After Installation

10. Note: With SDK 2.7 or later, clinfo.exe is copied under C:\windows\system32\ instead of C:\programfiles\AMD APP\.

11. Verify that the needed environment variables have been set correctly. Click on Start > Control Panel > System > Advanced System Settings > Environment Variables and view the System variables list:

- AMDAPPSDKROOT = c:\program files \ AMD APP (for 32-bit systems)
- AMDAPPSDKROOT = c:\program files (x86) \AMD APP (for 64-bit systems)
- AMDAPPSDKSAMPLESROOT = c:\Users\<username>\Libraries\Documents\AMD APP\



Environment Variables

User variables for user

Variable	Value
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

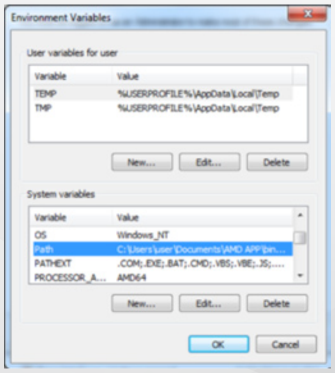
System variables

Variable	Value
AMDAPPSDKROOT	C:\Program Files (x86)\AMD APP\
AMDAPPSDKSA...	C:\Users\user\Documents\AMD APP\
ComSpec	C:\Windows\system32\cmd.exe
FP_NO_HOST_C...	NO

Figure 8: Viewing / Editing Environment Variables in Windows®

12. Verify that the path variable has been updated (also in the System variables list) to include:
    - $(AMDAPPSDKROOT)\bin\x86 (for 32-bit systems)
    - $(AMDAPPSDKROOT)\bin\x86_64 (for 64-bit systems)
    - $(AMDAPPSDKSAMPLESROOT)\bin\x86 (for 32-bit systems)
    - $(AMDAPPSDKSAMPLESROOT)\bin\x86_64 (for 64-bit systems)

    To edit the path variable, highlight the Path variable line in the System variables list and click edit. Additional folders can be added to the list and must be separated by a semicolon. Reboot in order for changes to take effect.



Environment Variables

User variables for user

Variable	Value
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

System variables

Variable	Value
OS	Windows_NT
Path	C:\Users\user\Documents\AMD APP\bin...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;....
PROCESSOR_A...	AMD64

Figure 9: Editing the Path Variable in Windows®

13. If you wish to review the sample applications at this time, browse to c:\Users\<username>\Libraries\Documents\AMD APP\samples\opencl\cl\app and open any of the folders associated with the sample you are interested in. Documentation for each sample can be found in the docs subfolder of each sample.

14. TROUBLESHOOTING TIP: While working through the next couple of sections, if you need to reinstall all or part of the AMD APP SDK (such as reinstalling a sample application that may have stopped working due to file/folder modifications), navigate to C:\AMD\SUPPORT\streamsdk_2-7_win732 or C:\AMD\SUPPORT\streamsdk_2-7_win764 and rerun Setup.exe. To remove and reinstall all sample programs and files (refer to the Figure 10):
    a. After the welcome screen, click on the Uninstall icon and click Next
    b. Select the Custom option and click Next
    c. Scroll down, select AMD APP SDK Samples and click Next
    d. View the log file or click Next to finish
    e. Rerun Setup.exe again, and repeat steps a-d but this time selecting Install, Custom, and AMD SDK Samples (Note that AMD AP SDK Samples will be highlighted by default)
    f. Accept the EULA and click Next to install only the AMD APP SDK samples
    g. View the log file or click Next to finish

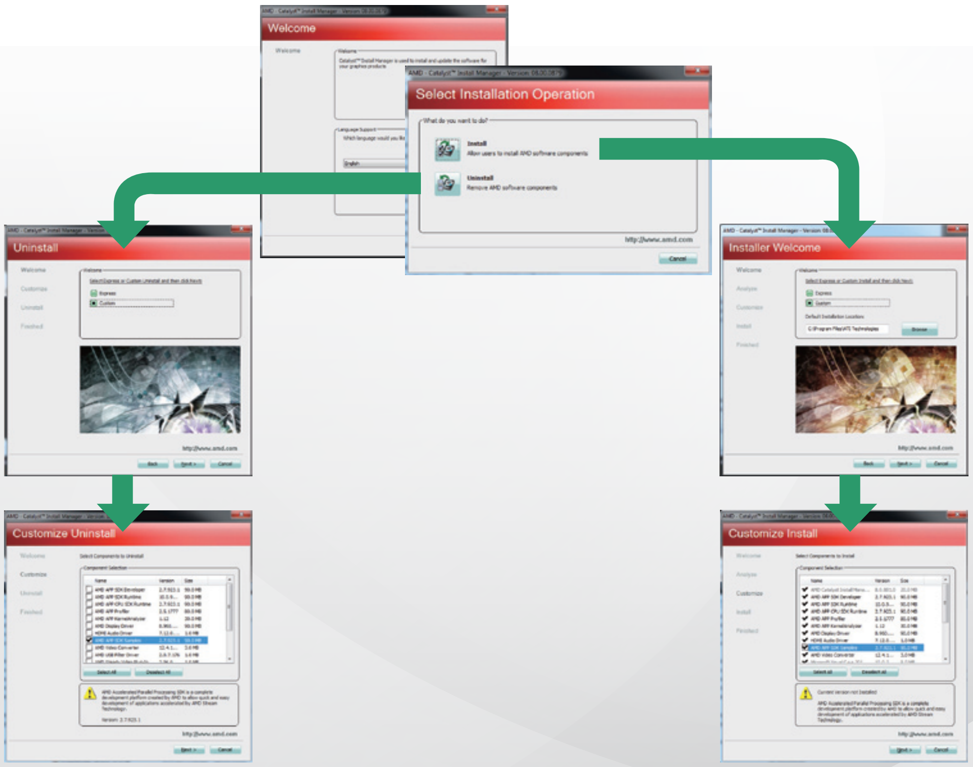


Figure 10: Uninstalling and Reinstalling AMD APP SDK Samples Content

Step 4: Execute a Pre-Built Sample

There is a large set of sample programs included in the AMD APP SDK. The samples vary in complexity and demonstrate a broad range of OpenCL algorithms and capabilities. They provide you with ideas for developing your own code plus allow you to quickly test your development environment. Each sample contains all of the files needed to use as an independent program and includes documentation explaining what the sample code does, how it was implemented, and how to build and run within the AMD APP SDK environment. Source code is also provided for each of the samples so that you can modify for your own needs and/or pull the sample code into your own project.

There are two kinds of OpenCL samples in the AMD APP SDK. One is written using native OpenCL calls (in $(AMDAPPSDKSAMPLESROOT)/samples/opencl/cl); the other is written using the AMD C++ bindings to OpenCL (in $(AMDAPPSDKSAMPLESROOT)/samples/opencl/cpp_cl).

Most of the OpenCL samples make use of a utility library called the AMD APP SDK Utility Library (SDKUtil). This library provides commonly used routines, such as parsing command line options, loading and writing bitmap images, printing formatted output, comparing results, and reading files. This is an object-oriented library that is structured as a set of classes. The SDKUtil cpp files and headers are available in the $(AMDAPPSDKSAMPLESROOT)/samples/opencl/SDKUtil folder.

A simple first step in verifying that your development environment is complete and that the AMD APP SDK has installed correctly is to execute one or more of the pre-built samples included with the SDK:

In Linux®

1. Navigate to opt/AMDAPP/samples/opencl/cl/app and review the various sample programs available. Each application folder has a subfolder named “docs” that contains a description and release notes for that sample. In this tutorial, we’ll start with ImageOverlap because it involves reading two separate image files, applying one to the other as an overlay, and then comparing the result to a third reference image. The program can be forced to generate a failure message by simply manipulating one of the image files.

2. Navigate to opt/AMDAPP/samples/opencl/bin/x86_64 and confirm that ImageOverlap (no extension), ImageOverlap_map.bmp, input512.bmp, and ImageOverlap_verify_map.bmp are contained within the folder. View the ImageOverlap.pdf file for additional details regarding this sample program.

3. Go to the command line, navigate to opt/AMDAPP/samples/opencl/bin/x86_64 and execute./ ImageOverlap –e.
   - The “–e” flag will cause the result to be compared to the reference image.
   - The end of the program output should display “Verifying result – Passed”.
   - Note that by default none of the images are actually displayed on the screen while executing this program.

4. Rename ImageOverlap__verify_map.bmp to ImageOverlap_verify_map_old.bmp and rename BoxFilter_Input.bmp to ImageOverlap_verify_map.bmp. This should force a failure message when the program compares the resulting image to what is now the incorrect reference image.
   a. Note: Use “mv” to rename files from the command line:
      mv ImageOverlap_verify_map.bmp ImageOverlap_verify_map_old.bmp
      mv BoxFilter_Input.bmp ImageOverlap_verify_map.bmp

5. Execute ./ImageOverlap –e again.
   The end of the program output should display “Verifying result – Failed”.

In Windows®

1. The included samples are generally meant to be run in a command-line window (for access to command-line execution options and display output). To open a command-line window with administrator rights in Windows, refer to Figure 11.
    a. Click on the Windows icon and type cmd in the search dialogue box.

    b. Right-click on cmd at the top of the search results screen.

    c. Select Run as administrator.

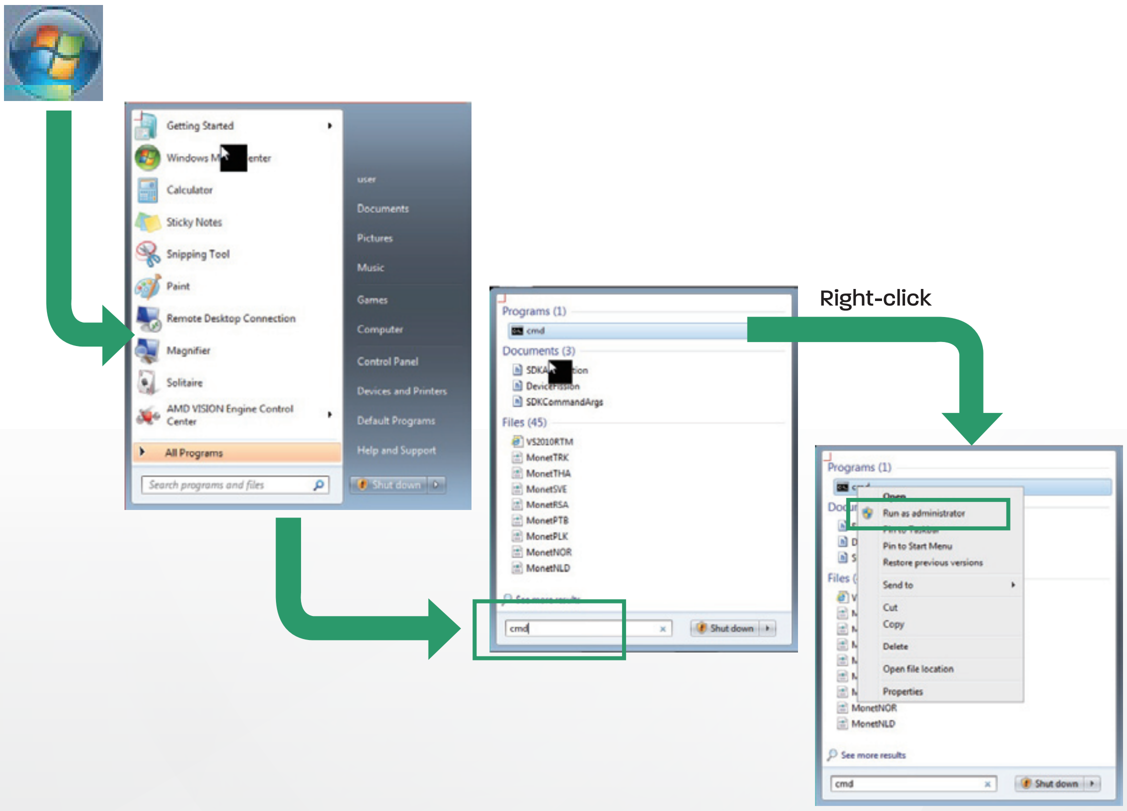


Figure 11: Windows® 7, Open a CMD Window as an Administrator

Note: Samples can be run from within Visual Studio (using F5 or CTL+F5), however, this method does not allow command-line options to be entered which can be extremely important when running many of the sample applications.

2. Navigate to c:\Users\<username>\Libraries\Documents\AMD APP\samples\opencl\cl\app and review the various sample programs available. Each application folder has a subfolder named “docs” that contains a description and release notes for that sample. In this tutorial, we’ll start with ImageOverlap because it involves loading an image file, filling a rectangular block with a specific color, applying the block as an overlay, and then comparing the result to a second reference image. The program can be forced to generate a failure message by simply manipulating one of the image files.

3. Navigate to c:\Users\<username>\Libraries\Documents\AMD APP\samples\opencl\bin\x86_64 and confirm that ImageOverlap.exe, ImageOverlap_map.bmp, input512.bmp, and ImageOverlap_verify_map.bmp are contained within the folder. View the ImageOverlap.pdf file for additional details regarding this sample program.

4. Go to the command line, navigate to c:\Users\<username>\Libraries\Documents\AMD APP\samples\opencl\bin\x86_64 and execute ImageOverlap –e.
   - The "–e" flag will cause the result to be compared to the reference image.
   - The end of the program output should display "Verifying result – Passed".
   - Note that by default none of the images are actually displayed on the screen while executing this program.

5. Rename ImageOverlap__verify_map.bmp to ImageOverlap_verify_map_old.bmp and rename BoxFilter_Input.bmp to ImageOverlap_verify_map.bmp. This should force a failure message when the program compares the resulting image to what is now the incorrect reference image.

6. Execute ImageOverlap –e again. The end of the program output should display "Verifying result – Failed".

Step 5: Build and Execute a Sample

In this section, we will rebuild an executable (ImageOverlap again) using the pre-installed source files, Kernels, and make files, and then verify that our newly-built executable performs exactly like the original executable.

In Linux®

1. Because all included samples are, by default, intended to be built and executed in debug mode, in order to build any of these samples, you must first build a debug library called libSDKUtil.a (which is needed by all of the sample make files) and then add it to your library path:
   a. Navigate to opt/AMDAPP/samples/opencl/SDKUtil.

   b. Type sudo make.

   c. Navigate to opt/AMDAPP/samples/opencl/SDKUtil/build/debug/x86_64 and verify that libSDKUtil.a has been created.

   d. Note that the LD_LIBRARY_PATH environment variable has previously been set to point to this folder in Step 3 above.

2. Choose which sample you wish to build. In this tutorial, we'll use ImageOverlap again.

3. Navigate to opt/AMDAPP/samples/opencl/cl/app/ImageOverlap and type sudo make.

4. Within the opt/AMDAPP/samples/opencl/cl/app/ImageOverlap folder, verify that the folder named "build" has been generated (along with /debug/x86_64 subfolders).

5. Navigate to opt/AMDAPP/samples/opencl/cl/app/ImageOverlap/build/debug/x86_64 and verify that a new ImageOverlap executable and ImageOverlap.o have been created.
   a. Note: the build process in this case does not copy the Kernel program or any of the associated reference image files into this directory so you cannot run your newly built executable from this folder.

6. Copy the executable you just created to the opt/AMDAPP/samples/opencl/bin/x86_64 folder (this will overwrite the executable that originally came with the SDK) and repeat the steps from section 4 above to verify that the executable you just created behaves the same as the executable originally included with the SDK.

7. For subsequent rebuilds of the same program it is recommended that you first delete the corresponding folder named "build" under opt/AMDAPP/samples/opencl/cl/app each time so it is easier to verify that the new files are being built.

In Windows®

An OpenCL application consists of a host program (C/C++), header files (.h, .hpp) and typically at least one Kernel program (.cl). To build an OpenCL application, the host program and header files must be compiled, which can be done using an off-the-shelf compiler such as G++ or Microsoft Visual C++, and the Kernel(s) must be compiled separately into device-specific binaries using the OpenCL compiler. In order to build OpenCL executables from AMD in Windows, you may use any of the following options:

1. Build with Visual Studio (recommended):
   For your convenience, in addition to providing individual source files for each sample program, the AMD APP SDK also contains Microsoft Visual Studio solution files (OpenCLSamplesVS10.sln for VS 2010 and OpenCLSamplesVS12.sln for VS 2012) that encompass all sample project files. Individual solution files for VS 2010 and VS 2012 are also provided for each sample program. You can rebuild a sample program from either an individual solution file or from the "master" solution file but the "master" solution file provides the additional benefit of being able to view, edit, or build any sample programs from a single solution file. For the first part of this tutorial, we will use the master solution files to build individual samples. The master solution files are located in (AMDAPPSDKSAMPLESROOT)\samples\opencl\.
   a. Note: Visual Studio 2010 Express may be used to build samples, however you will be limited to building only one sample at a time. Visual Studio 2010 Express cannot build all samples from the top-level solution/project file.

   b. Navigate to (AMDAPPSDKSAMPLESROOT)\samples\opencl\.

   c. Double-click OpenCLSampleVS10s.sln if Microsoft Visual Studio 2010 Professional Edition is installed or OpenCLSamplesVS12.sln if Microsoft Visual Studio 2012 Professional or Premium Edition is installed.
      i. 80 samples should be loaded (as originally included in AMD APP SDK 2.8).

      ii. By opening a solution file, the following project properties will already be set:
         - Configuration Properties > C/C++ > Additional Include Directories are set to the environment variables that we viewed earlier in this tutorial $(AMDAPPSDKROOT) and $(AMDAPPSDKSAMPLESROOT).
         - Configuration Properties > C/C++ > Preprocessor Definitions is set to ATI_OS_WIN (through an inheritance).
         - Configuration Properties > Linker > Additional Library Directories are set to the environment variables $(AMDAPPSDKROOT) and $(AMDAPPSDKSAMPLESROOT)
         - Configuration Properties > Linker > Input > Additional Dependencies is set to OpenCL.lib and SDKUtil.lib.

      iii. To view/edit the above project properties, right-click on the ImageOverlap project file in Solution Explorer and then click on Properties at the bottom of the menu.

      iv. Note: If you are starting a Visual Studio project/solution from scratch (and importing source code from a sample to start with for a new program/project for example), you will need to manually set the above properties.

d. To build executables for ALL sample programs, Select Build > Build Solution.

e. To build an executable for a SINGLE sample program (let's use ImageOverlap):
   i. Select ImageOverlap in the Solutions Explorer, right-click on the project file, and select Build.

   ii. OR, to use an individual solution file, navigate to (AMDAPPSDKSAMPLESROOT)\samples\opencl\cl\app\ImageOverlap and open the Visual Studio solution file called ImageOverlapVS10.sln or ImageOverlapVS12.sln. Select ImageOverlap in the Solutions Explorer, right-click on the project file, and select Build.

f. The executables and their associated Kernel programs are written into (AMDAPPSDKSAMPLESROOT)\samples\opencl\bin\debug\x86 Navigate to this folder and verify that a new ImageOverlap executable has been created.
   i. Note: For future iterations, it's recommended to either rename the executable (something like ImageOverlap.old) or delete it so that it's easier to see that a new executable is rebuilt.

g. Copy the executable you just created to the (AMDAPPSDKSAMPLESROOT)\samples\opencl\ \bin\x86_64 folder (this will overwrite the executable that originally came with the SDK or the last one you just built) and repeat the process in Step 4 above to verify that the executable you just created behaves the same as the executable originally included with the SDK.
   i. Note: If you already completed the exercises in Step 4 in order to generate a failure message, you must now rename all of the files back to their original names (except for the executable) prior to performing Step G above in order for the ImageOverlap program to execute properly.

h. Note: If you build a sample using make from a command prompt and then try to rebuild it from within Visual Studio, you will likely get an error message related to files being out of date and asking if you would like to rebuild them. Answering 'yes' should cause the application to finish building without further errors.

i. Note: In order to build from within Visual Studio, you may have to implement a workaround to a known documented issue in Microsoft's MSBuild package (the Microsoft.CppCommon.targets file). Without the workaround, upon attempting to build one of the included sample programs, you may receive the message:

   ">C:\Program Files (x86)\MSBuild\Microsoft.Cpp\v4.0\Microsoft.CppCommon.targets(673,5): error MSB3491: Could not write lines to file \\projects\\test\\build\\tmp\\redird\\Debug\\redird.exe.embed.manifest". The network path was not found."

   This issue is associated with Visual Studio Solution files that specify a relative path to the intermediate file directory (found under Project > Properties > Configuration Properties).

   The fix includes editing %Programsfile%\MSBuild\Microsoft.cpp\v4.0\Microsoft.cppcommon.targets

FROM:

```
<WriteLinesToFile
Condition="!EXISTS('$(InputManifest)')"
File="$(InputManifest)"
Lines=""
Overwrite="false"
Encoding ="Unicode"
/>
```

TO:

```
<WriteLinesToFile
Condition="!EXISTS('%(Manifest.OutputManifestFile)')"
File="%(Manifest.OutputManifestFile)
Lines=""
Overwrite="false"
Encoding ="Unicode"
/>
```

2. Build from the Windows command line:
   To build from the command line, you typically need to run make. AMD supplies two support files for make, openclsdkdefs.mk and openclsdkrules.mk, which control the build process for the included sample programs (and are written to invoke the g++ compiler). However, since make is typically installed as part of Linux not Windows, trying to execute make from the command line will most likely generate " 'make' is not a recognized as an internal or external command, operable program or batch file." So, in order to build executables under Windows, you must install a Windows compatible make utility.
   a. Download and install MinGW. MinGW is a GNU-based open source toolset that provides a minimalist development environment for native Microsoft Windows applications. Download from http://www.mingw.org/.
      i. Download standard 32-bit MinGW from http://sourceforge.net/projects/mingw/.

      ii. Download 64-bit MinGW from http://sourceforge.net/projects/mingw-w64/.

      iii. Add MinGW bin and Msys bin to the PATH variable:

      iv. PATH=C:\MinGW\bin;C:\MinGW64\bin;C:\MinGW\msys\1.0\bin

      iv. Note: MinGW is also distributed as part of CYGWIN (http://www.cygwin.com/), a collection of tools which provide a Linux look and feel environment for Windows. It is installed as a DLL (cygwin1.dll), which acts as a Linux API layer providing substantial Linux API functionality.

      v. Reboot to force any changes to the PATH variable to take effect.

      vi. Note: GNU make and GNUWin32 make (an updated version of the standard GNU make utility that is compatible with the latest Windows operating systems including Windows 7) are currently not supported in the AMD APP SDK.

   b. Because all included samples are, by default, intended to be built and executed in debug mode, in order to build any of these samples, you must also build a debug library called libSDKUtil.a (which is needed by all of the sample make files) and then add it to your library path:
      i. Navigate to (AMDAPPSDKSAMPLESROOT)\samples\opencl\SDKUtil.

ii. Type make.

iii. Navigate to (AMDAPPSDKSAMPLESROOT)\samples\opencl\SDKUtil\build\debug\x86 and verify that libSDKUtil.a has been created.

iv. Note that the LD_LIBRARY_PATH environment variable has previously been set to point to this folder.

c. Build a Sample:
   i. Build With MinGW + GCC Using make Files
      1. Open a command prompt window and navigate to the desired sample folder under (AMDAPPSDKSAMPLESROOT)\samples\opencl\cl\app\.

      2. Type make bitness=32.

      3. Verify that a new executable (and associated run-time files) has been created in (AMDAPPSDKSAMPLESROOT)\samples\opencl\bin\x86.

   ii. Build With MinGW-x64 + GCC Using make Files
      1. Open a command prompt window and navigate to the desired sample folder under (AMDAPPSDKSAMPLESROOT)\samples\opencl\cl\app\.

      2. Type make bitness=64.

      3. Verify that a new executable (and associated run-time files) has been created in (AMDAPPSDKSAMPLESROOT)\samples\opencl\bin\x86_64.

   iii. Note: To use Msys under MinGW64-x64:
      1. Open the fstab file (available at msys/1.0/etc/)

      2. Modify according to your MinGW64-w64 path. For example, modify C:\MinGW\ /mingw to C:\MinGW64\ /mingw

3. Build by accessing the command line from within Visual Studio:
   a. This process gains you access to the command prompt from within Visual Studio. It requires the installation of the MinGW make utility and adding make.exe to your PATH environment variable (as described previously in Step 3: Download and Install the AMD APP SDK).

   b. In Visual Studio, click on Tools > Visual Studio Command Prompt to open a command prompt within the Visual Studio environment. This command prompt will be the same as if you were to open a command prompt from the Windows start menu as explained earlier.

   c. Within the command prompt window, navigate to AMDAPPSDKSAMPLESROOT)\samples to build all samples. To build samples individually, navigate to the desired application folder under (AMDAPPSDKSAMPLESROOT)\samples\opencl\cl\app.

   d. For 64-bit builds, simply type make; for 32-bit builds, type make bitness=32. (If your OS is 64-bit, it builds 64-bit targets by default, but to build 32-bit targets, it is necessary to explicitly specify bitness=32.)

4. Build With Intel Compiler Using make Files:
   a. Click on Tools > Visual Studio Command Prompt to open a command prompt within

the Visual Studio environment. This command prompt will be the same as if you were to open a command prompt from the Windows start menu as explained earlier.

b. Within the command prompt window, navigate to AMDAPPSDKSAMPLESROOT)\samples to build all samples. To build samples individually, navigate to (AMDAPPSDKSAMPLESROOT)\samples\opencl\cl\app.

c. At the command line, enter make clean.

d. Build the sample using the command make intel=1 sample_Name.

e. For 64-bit systems use make intel=1 to build 64-bit targets, and make intel=1 bitness=32 to build 32-bit targets. (If your OS is 64-bit, it builds 64-bit targets by default, but to build 32-bit targets, it is necessary to explicitly specify bitness=32.)

Step 6: Modify, Build, and Execute a Simple Program

This section will make you familiar with the various types of files needed to modify, build, and execute a simple program including how to write a host program and how to set up and execute a Kernel on an OpenCL device (in this case, on our integrated AMD Radeon 6320 GPU). There are many simple OpenCL program examples available on the internet, including in AMD published documentation, that showcase different programming techniques and OpenCL functions.

For our purposes, we will build and execute another sample program to get familiar with its functionality. We will then make modifications to the source files, rebuild, and execute our new program to test that the changes we made had the intended effect. The simplest OpenCL sample included in the AMD APP SDK is called “Template” and is included for instructional purposes and specifically to assist developers that are new to OpenCL programming:

- The Template Kernel (Template_kernels.cl) is an OpenCL Kernel that multiplies each element of an input array with a scalar, and then stores it in an output array.
- The C++ host program (Template.cpp) displays the input array, the output array, compares to a known result, and prints a “passed” or “failed” message on the screen.

Build and Execute In Linux®

1. Navigate to opt/AMDAPP/samples/opecl/cl/app/Template and review some of the key files that make up the Template program.
    a. Template.cpp: The host OpenCL program written in C++.

    b. Template.hpp: The C++ bindings header file.

    c. Template_kernels.cl: The Kernel that is called by the host program that performs the mathematical function on the data read in and passed to it by the host.

    d. Template.depend (located in Template/depends/x86_64): a build file that generates the .o file.

    e. Makefile: builds the executable (using the g++ compiler).

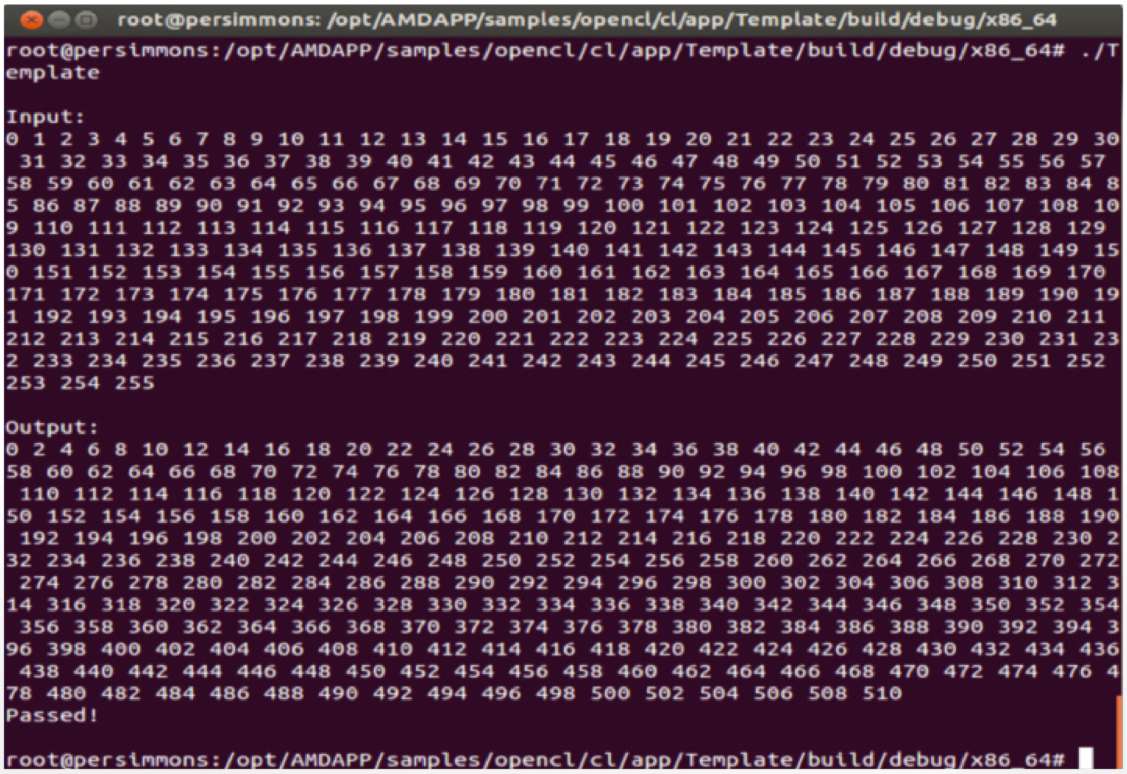
2. Go to the command line and type make.

3. Navigate to opt/AMDAPP/samples/opecl/cl/app/Template/build/debug/x86_64.

4. If the file Template_kernels.cl does not exist, copy it from opt/AMDAPP/samples/opecl/cl/app/Template into this folder (the Kernel file is needed at runtime).

5. Verify that the Template executable and Template.o have been generated.

6. Execute Template. The output should look like Figure 12. Note that the Input string goes from 0 to 255 (256 characters total) and the Output string goes from 0 to 510 (255 * 2), and the result is "Passed!"



```
root@persimmons: /opt/AMDAPP/samples/opencl/cl/app/Template/build/debug/x86_64
root@persimmons:/opt/AMDAPP/samples/opencl/cl/app/Template/build/debug/x86_64# ./T
emplate

Input:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57
58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 8
5 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 10
9 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129
130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 15
0 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170
171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 19
1 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211
212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 23
2 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252
253 254 255

Output:
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56
58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94 96 98 100 102 104 106 108
 110 112 114 116 118 120 122 124 126 128 130 132 134 136 138 140 142 144 146 148 1
50 152 154 156 158 160 162 164 166 168 170 172 174 176 178 180 182 184 186 188 190
 192 194 196 198 200 202 204 206 208 210 212 214 216 218 220 222 224 226 228 230 2
32 234 236 238 240 242 244 246 248 250 252 254 256 258 260 262 264 266 268 270 272
 274 276 278 280 282 284 286 288 290 292 294 296 298 300 302 304 306 308 310 312 3
14 316 318 320 322 324 326 328 330 332 334 336 338 340 342 344 346 348 350 352 354
 356 358 360 362 364 366 368 370 372 374 376 378 380 382 384 386 388 390 392 394 3
96 398 400 402 404 406 408 410 412 414 416 418 420 422 424 426 428 430 432 434 436
 438 440 442 444 446 448 450 452 454 456 458 460 462 464 466 468 470 472 474 476 4
78 480 482 484 486 488 490 492 494 496 498 500 502 504 506 508 510
Passed!

root@persimmons:/opt/AMDAPP/samples/opencl/cl/app/Template/build/debug/x86_64#
```

Figure 12: Ubuntu Linux®, Successful Execution of "Template" Sample Program

Build and Execute In Windows®

1. Navigate to (AMDAPPSDKSAMPLESROOT)\samples\opencl\cl\app\Template and review the various files that make up the Template program.
   a. Template.cpp: The host OpenCL program written in C++.

   b. Template.hpp: The C++ bindings header file.

   c. Template_kernels.cl: The Kernel that is called by the host program that performs the mathematical function on the data read in and passed to it by the host.

   d. Makefile: Builds the executable (using the g++ compiler).

   e. TemplateVS10.*: Solution, project, and configuration files for use in Visual Studio 2010.

   f. TemplateVS12.*: Solution, project, and configuration files for use in Visual Studio 2012.

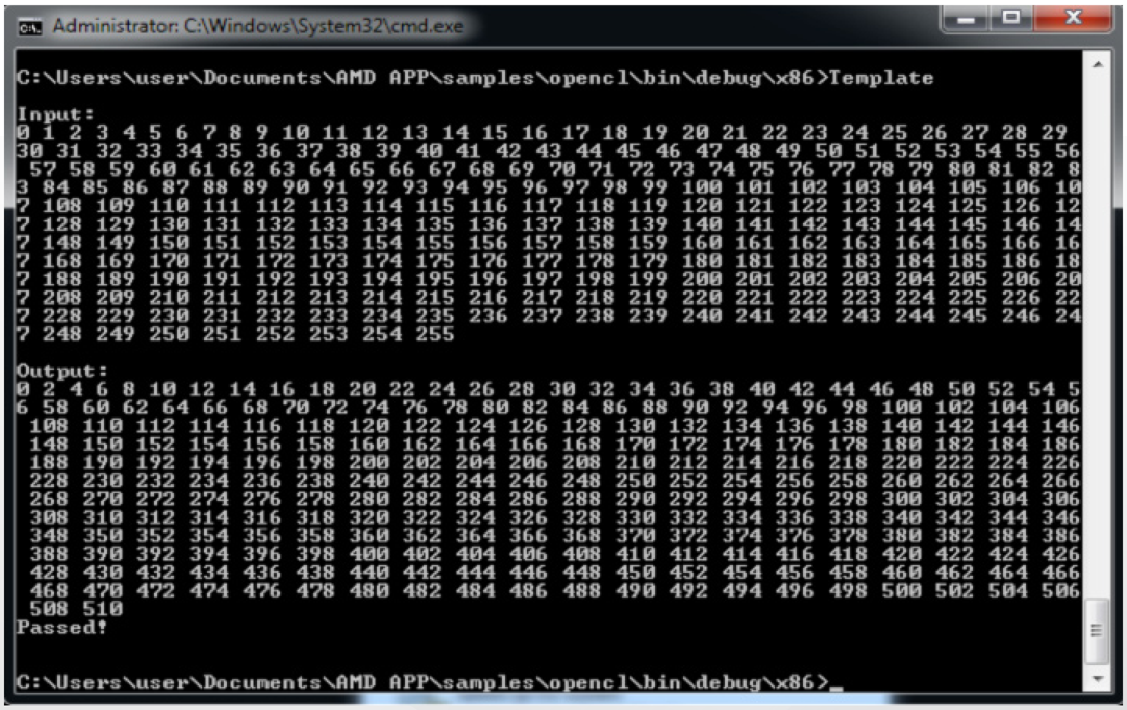
2. Build the executable using one of the options described previously in the Windows section of Step 5.

3. Navigate to (AMDAPPSDKSAMPLESROOT)\samples\opencl\bin\debug\x86.

4. If the file Template_kernels.cl does not exist, copy it from (AMDAPPSDKSAMPLESROOT)\samples\opencl\cl\app\Template into this folder (the Kernel file is needed at runtime as it does not get built into the executable).

5. Verify that a new Template executable has been generated.

6. Open a command window as described previously, navigate to (AMDAPPSDKSAMPLESROOT)\samples\opencl\bin\debug\x86, and execute Template. The output should look like Figure 13. Note that the Input string goes from 0 to 255 (256 characters total) and the Output string goes from 0 to 510 (255 * 2), and the result is "Passed!"



```
C:\Users\user\Documents\AMD APP\samples\opencl\bin\debug\x86>Template

Input:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56
 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 8
3 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 10
7 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 12
7 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 14
7 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 16
7 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 18
7 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 20
7 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 22
7 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 24
7 248 249 250 251 252 253 254 255

Output:
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 5
6 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94 96 98 100 102 104 106
 108 110 112 114 116 118 120 122 124 126 128 130 132 134 136 138 140 142 144 146
 148 150 152 154 156 158 160 162 164 166 168 170 172 174 176 178 180 182 184 186
 188 190 192 194 196 198 200 202 204 206 208 210 212 214 216 218 220 222 224 226
 228 230 232 234 236 238 240 242 244 246 248 250 252 254 256 258 260 262 264 266
 268 270 272 274 276 278 280 282 284 286 288 290 292 294 296 298 300 302 304 306
 308 310 312 314 316 318 320 322 324 326 328 330 332 334 336 338 340 342 344 346
 348 350 352 354 356 358 360 362 364 366 368 370 372 374 376 378 380 382 384 386
 388 390 392 394 396 398 400 402 404 406 408 410 412 414 416 418 420 422 424 426
 428 430 432 434 436 438 440 442 444 446 448 450 452 454 456 458 460 462 464 466
 468 470 472 474 476 478 480 482 484 486 488 490 492 494 496 498 500 502 504 506
 508 510
Passed!

C:\Users\user\Documents\AMD APP\samples\opencl\bin\debug\x86>_
```

Figure 13: Windows® 7, Successful Execution of "Template" Sample Program

Edit/Modify the Program Files

In this section we will modify the source files associated with the Template program. This can be done using any editor present in Windows or Linux. Which OS we use does not affect how we want to rename the files or the changes we need to make within each source file. In Windows, an additional option is to edit the files by opening them from within Microsoft Visual Studio. The edits we are going to make to the source files will result in:

- Performing an addition rather than a multiplication on the input data stream
- Changing the constant from 2 to 4 (so instead of multiplying each vector by 2, we will be adding 4 to each vector)
- Changing the width of the input data stream (which also controls the number of iterations performed by the Kernel) from 256 to 128.

In Visual Studio 2010/2012

Experienced Visual Studio users know that the process of creating a new solution file from an existing solution file is more complicated than simply renaming files from within the Solution Explorer tool window. To simplify the process in this tutorial, we will edit the program files without changing the file names thereby eliminating the need to copy or modify the solution or project files (or their associated attributes).

1. Backup the original Template program files by creating a new folder called Template_back and copying the contents of the Template folder into this new folder.

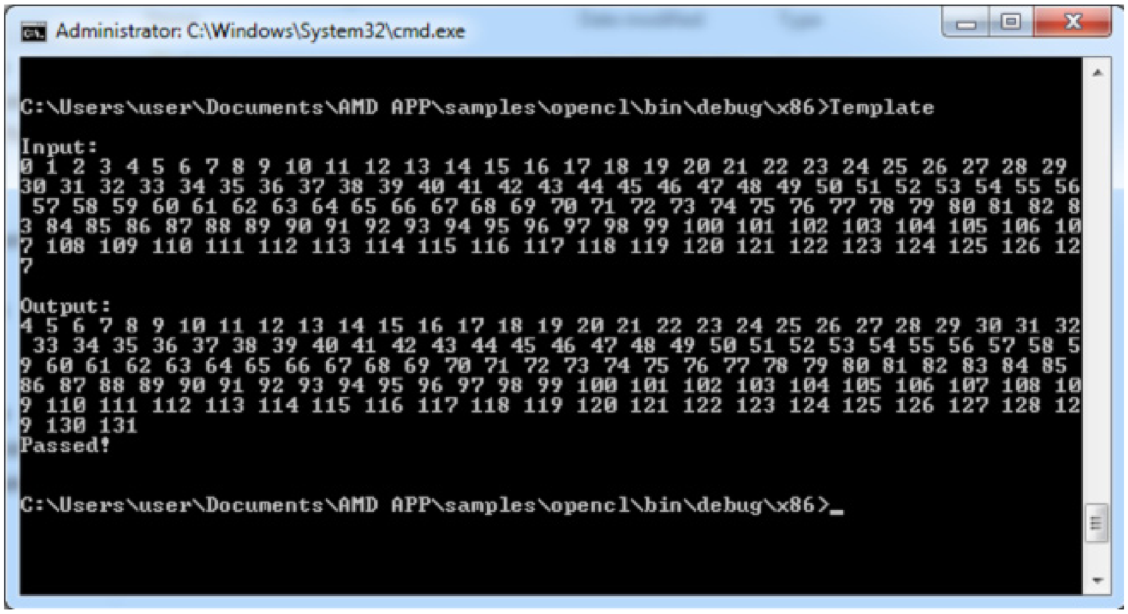
2. Edit the program files in the original Template folder either from within Visual Studio or outside of Visual Studio using the editor of your choice. Refer to the OS Independent Modifications section below for the changes that need to be made in each file. You will be modifying Template.cpp, Template.hpp, and Template_Kernels.cl.

3. Build/Rebuild the solution by clicking on Build > Build Solution or Build > Rebuild Solution.
    a. Note: It is recommended to use the Build > Clean Solution occasionally or as needed to remove all intermediate and output files associated with a previous build operation.

4. Run your new program:
    a. Navigate to (AMDAPPSDKSAMPLESROOT)\samples\opencl\bin\debug\x86 and verify that the Template executable has been created and that Template_kernels.cl has also been copied/updated (it is not built into the executable).
        i. Note: For subsequent builds, the date/time-stamp of the Template executable will always change since it is regenerated every time, however, the date/time-stamp of the Template_Kernels.cl file will be associated with the last time you edited it. This is because it is copied from the Template folder not generated during each build. Therefore it will not be obvious that Template_Kernels.cl is being copied on every build so it is recommended that you delete it from the (AMDAPPSDKSAMPLESROOT)\samples\opencl\bin\debug\x86 folder before a build in order to verify that it is being copied into the target folder.

    b. Open a command prompt (run as administrator), navigate to (AMDAPPSDKSAMPLESROOT)\samples\opencl\bin\debug\x86, and execute Template.

    c. The output should look like the Figure 14. Note that the Input string now goes from 0 to 127 (128 characters total) and the Output string goes from 4 to 131 (127 + 4), and the result is “Passed!”



Administrator: C:\Windows\System32\cmd.exe

```
C:\Users\user\Documents\AMD APP\samples\opencl\bin\debug\x86>Template

Input:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56
 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 8
3 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 10
7 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 12
7

Output:
4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 5
9 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85
86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 10
9 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 12
9 130 131
Passed!


C:\Users\user\Documents\AMD APP\samples\opencl\bin\debug\x86>_
```

Figure 14: Windows® 7, Successful Execution of the Modified Template Program

OS Independent Modifications

This section describes all steps necessary to modify program files associated with the Template sample program in order to change its functionality. The changes are identical regardless of whether your development environment is Linux-based or Windows-based.

1. If you have not done so already, back up the original Template program files by creating a new folder called Template_back and copying the contents of the Template folder into this new folder.

2. Modify Template.cpp (the end of each set of instructions shows the final changes highlighted):
    a. In the Host Initialization section near the top of the file:
        - Change width = 256; to width = 128;
        - Change multiplier = 2; to adder = 4;

```
* \brief Host Initialization
 *       Allocate and initialize memory
 *       on the host. Print input array.
 */
int
initializeHost(void)
{
    width           = 128;
    input           = NULL;
    output          = NULL;
    adder           = 4;
```

b. In the “Step 8 Set appropriate arguments to the Kernel“ section:
   - Change 8.3 Kernel Arg multiplier (cl_uint) to 8.3 Kernel Arg adder (cl_uint).
   - Change //multiplier to //adder.
   - Change (void *)&multiplier); to (void *)&adder);
   - Change std::cout << "Error: Setting Kernel argument. (multiplier)\n"; to std::cout << "Error: Setting Kernel argument. (adder)\n";

```
//////////////////////////////////////////////////////////////////////
// STEP 8 Set appropriate arguments to the kernel
//    8.1 Kernel Arg outputBuffer ( cl_mem object)
//    8.2 Kernel Arg inputBuffer (cl_mem object)
//    8.3 Kernel Arg adder (cl_uint)
//////////////////////////////////////////////////////////////////////

// the output array to the kernel
status = clSetKernelArg(
                kernel,
                0,
                sizeof(cl_mem),
                (void *)&outputBuffer);
if(status != CL_SUCCESS)
{
    std::cout << "Error: Setting kernel argument. (output)\n";
    return SDK_FAILURE;
}

// the input array to the kernel
status = clSetKernelArg(
                kernel,
                1,
                sizeof(cl_mem),
                (void *)&inputBuffer);
if(status != CL_SUCCESS)
{
    std::cout << "Error: Setting kernel argument. (input)\n";
    return SDK_FAILURE;
}

// adder
status = clSetKernelArg(
                kernel,
                2,
                sizeof(cl_uint),
                (void *)&adder);
if(status != CL_SUCCESS)
{
    std::cout << "Error: Setting kernel argument. (adder)\n";
    return SDK_FAILURE;
}
```

c. In the “Step 11 Clean up the OpenCL resources used” section, under the heading “* \brief Print no more than 256 elements of the given array”:
   - Change * \brief Print no more than 256 elements of the given array to * \brief Print no more than 128 elements of the given array.
   - Change cl_uint numElementsToPrint = (256 < length) ? 256 : length; to cl_uint numElementsToPrint = (128 < length) ? 128 : length;
   - Change if(input[i] * multiplier != output[i]) to if(input[i] + adder != output[i]).

```
* \brief Print no more than 128 elements of the given array.
 *
 *        Print Array name followed by elements.
 */
void print1DArray(
         const std::string arrayName,
         const unsigned int * arrayData,
         const unsigned int length)
{
    cl_uint i;
    cl_uint numElementsToPrint = (128 < length) ? 128 : length;

    std::cout << std::endl;
    std::cout << arrayName << ":" << std::endl;
    for(i = 0; i < numElementsToPrint; ++i)
    {
        std::cout << arrayData[i] << " ";
    }
    std::cout << std::endl;

}

void verify()
{
    bool passed = true;
    for(unsigned int i = 0; i < width; ++i)
        if(input[i] + adder != output[i])
            passed = false;
```

3. Modify Template.hpp (the end of each set of instructions shows the final changes highlighted):
   a. In the // GLOBALS section:
      - Change * Multiplier is stored in this variable to * adder is stored in this variable.
      - Change cl_uint multiplier; to cl_uint adder;

```
// GLOBALS
#define SDK_SUCCESS 0
#define SDK_FAILURE 1

/*
 * Input data is stored here.
 */
cl_uint *input;

/*
 * Output data is stored here.
 */
cl_uint *output;
/*
 * Adder is stored in this variable
 */
cl_uint adder;
```

   b. Near the end of the //FUNCTION DECLARATIONS section:
      - Change * Prints no more than 256 elements of the given array to * Prints no more than 128 elements of the given array.
      - Change * Prints full array if length is less than 256 to * Prints full array if length is less than 128.

```
/*
 * Prints no more than 128 elements of the given array.
 * Prints full array if length is less than 128.
 *
 * Prints Array name followed by elements.
 */
void print1DArray(
        const std::string arrayName,
        const unsigned int * arrayData,
        const unsigned int length);
```

4. Modify Template_Kernels.cl (the end of each set of instructions shows the final changes highlighted):
    a. Change * Sample Kernel which multiplies every element of the input array with to * Sample Kernel which adds every element of the input array with.

    b. Change const   unsigned int multiplier) to const   unsigned int adder).

    c. Change output[tid] = input[tid] * multiplier; to output[tid] = input[tid] + adder;

```
/*!
 * Sample kernel which adds every element of the input array with
 * a constant and stores it at the corresponding output array
 */

__kernel void templateKernel(__global  unsigned int * output,
                             __global  unsigned int * input,
                             const     unsigned int adder)
{
    uint tid = get_global_id(0);

    output[tid] = input[tid] + adder;
}
```

Build and Execute the Modified Program in Linux®

1. If you haven't done so already, perform the modifications to the three Template program files described previously in the OS Independent Modifications section.

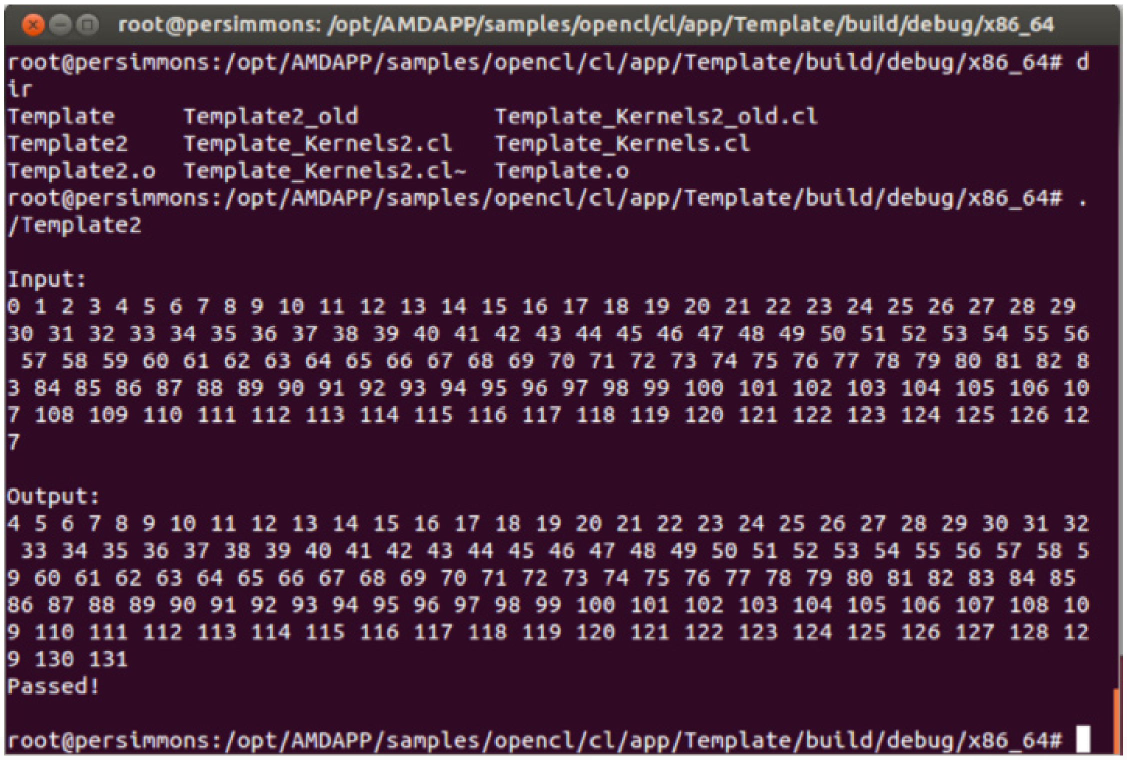
2. Navigate to opt/AMDAPP/samples/opencl/cl/app/Template/build/debug/x86_64 and verify that Template, Template.o, and Template_Kernels.cl do not exist. If they do (from a previous build), delete them before re-running make.

3. Navigate to opt/AMDAPP/samples/opencl/cl/app/Template and run make.

4. Copy Template_kernels.cl to opt/AMDAPP/samples/opencl/cl/app/Template/build/debug/x86_64 (Note: the Kernel needs to be in the same directory as the executable at runtime).

5. Navigate to opt/AMDAPP/samples/opencl/cl/app/Template/build/debug/x86_64 and verify that Template and Template.o have been generated.

6. Execute Template. The output should look like Figure 15. Note that the Input string now goes from 0 to 127 (128 characters total) and the Output string now goes from 4 to 131 (127 + 4), and the result is "Passed!"



```
root@persimmons: /opt/AMDAPP/samples/opencl/cl/app/Template/build/debug/x86_64
root@persimmons:/opt/AMDAPP/samples/opencl/cl/app/Template/build/debug/x86_64# d
ir
Template     Template2_old          Template_Kernels2_old.cl
Template2    Template_Kernels2.cl   Template_Kernels.cl
Template2.o  Template_Kernels2.cl~  Template.o
root@persimmons:/opt/AMDAPP/samples/opencl/cl/app/Template/build/debug/x86_64# .
/Template2

Input:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56
 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 8
3 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 10
7 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 12
7

Output:
4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 5
9 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85
86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 10
9 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 12
9 130 131
Passed!

root@persimmons:/opt/AMDAPP/samples/opencl/cl/app/Template/build/debug/x86_64# 
```

Figure 15: Ubuntu Linux®, Successful Execution of Modified Template Program

Recommended Next Steps

Hopefully this white paper and tutorial have given you a basic understanding of how to work with an OpenCL development environment, and how to get some basic programs compiled and running. The next step in your learning process/development project is to begin developing more advanced programs (such as implementing multiple Kernels) including porting some existing code to OpenCL. Depending on your specific situation and level of expertise, the following recommendations may be helpful to you:

1. Review additional information on AMD's OpenCL web pages including documentation, videos, blogs, forums, and code samples.

2. Download the latest OpenCL specification from Khronos. Khronos also provides developer resources, references pages, and a technical forum. Make sure you download the handy OpenCL API Quick Reference Card.

3. Obtain an OpenCL Programmers Guide.

4. Experiment with the other samples included in the AMD APP SDK, and refer to the documentation for each. These additional samples highlight various OpenCL functions and algorithms, demonstrate additional task and data parallelism, employ multiple Kernels, can be run on both CPU and GPU devices, and can be incorporated into your own source code.

5. Begin porting existing code to OpenCL. Again, additional resources can be found on AMD's OpenCL-related web pages including documentation and discussions around porting from OpenGL and CUDA.

6. If you have a large existing OpenGL code base, investigate OpenCL's interoperability with OpenGL, specifically its ability to enhance and accelerate OpenGL graphical rendering capabilities on GPUs.

7. Learn how to optimize your code for performance. There are many established techniques for accomplishing this including: managing Kernel execution, minimizing external memory accesses, and understanding how certain platform parameters such as bandwidth between Host and GPU and the memory bandwidth of the GPU can affect overall application performance.

8. Learn how to build portable applications that can run on numerous device architectures and configurations (including CPU only, CPU + GPU, or CPU + multiple GPU) without requiring recompilation.

9. Learn about Aparapi, C++ Amp, and AMD's new Bolt C++ template library.

10. Learn about additional tools available from AMD, including AMD's new Code XL Unified Developer Tool Suite (can be used in conjunction with AMD APP SDK 2.8 and available as a separate download) that supports GPU debugging, GPU and CPU profiling, static OpenCL kernel analysis, and provides a standalone user interface on Windows and Linux.

Finding Additional Information

The following table provides links to AMD documentation and other references that have been used in this paper or that provide additional details on the topics covered in this paper. It is highly recommended that you refer to these documents for additional information either during or after completing the programming tutorial part of this paper.

Title	Link
AMD Embedded Applications: Details on common applications using AMD processors	www.amd.com/us/products/embedded/applications/Pages/applications.aspx
AMD Embedded G-Series Platform Website	www.amd.com/us/products/embedded/processors/Pages/g-series.aspx
"AMD Embedded G-Series Processors and the x86 Set-Top Box Revolution: Achieving the Ultimate Fusion of TV, PC and Internet" White paper	www.amd.com/us/Documents/50356_G-Series_xSTB_Whitepaper.pdf
Express Logic press release on ThreadX Support for AMD G-Series platform	http://rtos.com/news/detail/threadx_support_for_amd_g-series_announced/
Windows Embedded Board Support Packages (BSPs)	www.microsoft.com/windowsembedded/en-us/downloads/board-support-packages-for-windows-embedded.aspx
AMD OpenCL Zone home page	http://developer.amd.com/resources/hc/openclzone/Pages/default.aspx
A helpful introduction to OpenCL:	http://blogs.amd.com/developer/2009/09/15/amd-developer-inside-track-episode-2-opencl-introduction/

AMD App Acceleration (formerly ATI Stream) home page	http://www.amd.com/us/products/technologies/amd-app
AMD APP SDK for OpenCL: Overview page	http://developer.amd.com/tools/hc/AMDAPPSDK/Pages/default.aspx
AMD APP SDK documentation page	http://developer.amd.com/tools/hc/AMDAPPSDK/documentation/Pages/default.aspx
“APUs strike the ideal balance of form, function, and power consumption for graphics-intensive portable devices” (article)	www.smallformfactors.com/articles/id/?5556
AMD OpenCL Developers Forum	http://devgurus.amd.com/community/opencl
The Khronos Group - OpenCL Overview	http://www.khronos.org/opencl/
Ubuntu Linux Documentation and Help pages	https://help.ubuntu.com/
GNU Project	http://www.gnu.org/
CYGWIN Project	http://www.cygwin.com/
Microsoft Visual Studio 2010 Information and 30-day Free Trial Download	http://www.microsoft.com/visualstudio/en-us/try

[1] While running a Winbench® 99 business graphics benchmark the AMD G-T16R APU consumed an average of 2.284W. I/O Controller Hub power is estimated based on the measured average power drawn by the I/OCH of .965W during a run of 3DMark 06. System Configuration: AMD G-T16R APU (DVT) at 30°C, “Inagua” Development Board, 4GB 1.35V DDR3, Windows 7 Ultimate. EMB-26