A Basic Sample of OpenCL™ Host Code

By Tom Craver (Intel Corporation Software Applications Engineer)

Contents

Introduction

Programmers new to OpenCL may find that the most complete documentation—the Khronos OpenCL specification—is not the best guide to getting started programming for OpenCL. The specification describes many options and alternatives, which can be confusing at first. Other code samples written for OpenCL may focus on the device kernel code, or may use host code written with an OpenCL "wrapper" library that hides the details of how to directly use the standard OpenCL host API.

The **SampleOCL** sample code described in this document aims to provide a clear and readable representation of the basic elements of a non-trivial OpenCL program. The focus of the sample code is the OpenCL™ code for the host (CPU), rather than kernel coding or performance. It demonstrates the basics of constructing a fairly simple OpenCL application, using the OpenCL v1.2 specification.[1] Similarly, this document focuses on the structure of the host code and the OpenCL APIs used by that code.

About the Sample

This code sample uses the same OpenCL kernel as the ToneMapping sample (see reference below), previously published for the Intel® SDK for OpenCL Applications [2]. This simple kernel attempts to make visible features of an image that would otherwise be too dark or too bright to distinguish. It reads pixels from an input buffer, modifies them, and writes them out to the same position of an output buffer. For more information on how this kernel works, see the document *High Dynamic Range Tone Mapping Post Processing Effect* [3].

OpenCL Implementation

The SampleOCL sample application is not intended to "wrap" OpenCL; that is, it does not try to replace OpenCL APIs with a "higher level" API. Generally I have found that such wrappers are not much simpler or cleaner than using the OpenCL API directly and, while the original programmer of a wrapper may find the wrapper easier to work with, the wrapper will impose a burden on any OpenCL programmer called upon to maintain the code. The OpenCL APIs are a standard. To wrap them in a proprietary "improved" API is to throw away much of the value of having that standard.

With that said, the SampleOCL implementation does make use of a few C++ classes and associated methods to separate the use of OpenCL APIs into a few groups. The application is broken into two main classes to separate generic application elements from elements related to OpenCL. The former is C_SampleOCLApp; the latter is C_OCL.

Limitations

This sample code focuses only on the basics of an OpenCL application, as specified in version 1.2. It provides no insight into differences from other revisions, though most of the information should still be relevant for newer revisions.

The host side application code of this sample is not intended to demonstrate the most optimal performance. For simplicity, several obvious optimizations have been left out.

OpenCL Application Basics

What follows is a fairly complete explanation of a *basic* OpenCL application program sequence. The emphasis is on "basic," as many options are not covered. More information can be found in the OpenCL specification [1].

An OpenCL application should be able to execute with substantial parallelism on a variety of processing ***devices*** such as multi-core CPUs with SIMD instruction support and Graphics Processing Units (GPUs), either discrete or integrated into a CPU. As such, one of the first things an OpenCL application must do is determine what devices are available and select the device or devices that will be used. A single ***platform*** might support more than one type of device, such as a CPU that has an integrated GPU, and more than one platform may be available to the application.

Each platform available to the OpenCL application will have an associated name, vendor, etc. That information can be obtained using the OpenCL APIs `clGetPlatformIDs()` followed by `clGetPlatformInfo()` and can be used to select a desired platform.

Once a platform is selected, a ***context*** must be created to encompass the OpenCL devices, memory, and other resources needed by an application. With the selected platform ID and a specification of the desired device type (CPU, GPU, etc.), an application can call `clCreateContextFromType()` and then use `clGetContextInfo()` to obtain the device IDs. Or, it can directly request device IDs for a given platform ID and device type using `clGetDeviceIDs()` and then use `clCreateContext()` with those device IDs to create the context. This sample uses the latter approach to create a context with a single GPU device.

With the desired device ID(s) and context, one can create a ***command queue*** for each device to be used using `clCreateCommandQueue()`. The command queue is used to "enqueue" operations from the host application to the GPU or other device, for example, requesting that a particular OpenCL kernel be executed. This sample code creates a single command queue for a GPU device.

With that initialization work done, a common next step is to create one or more OpenCL ***program*** objects using `clCreateProgramWithSource()`. Once the program is created, it must still be built (essentially compiled and linked) using `clBuildProgram()`. That API allows setting options to the compiler, such as `#defines` to modify the program source.

Finally, with the program created and built, ***kernel*** objects that link to the functions in that program can be created, calling `clCreateKernel()` for each kernel function name.

Prior to executing an OpenCL kernel, set up the data to be processed—usually done by creating linear memory ***buffers*** using the `clCreateBuffer()` API function. (An ***Image*** is another OpenCL memory object type not used in this sample.) The `clCreateBuffer` function can allocate memory for a buffer of a given size and optionally copy data from host memory, or it can set up the buffer to directly use space already allocated by the host code. (The latter can avoid copying from host memory to the OpenCL buffer, which is a common performance optimization.)

Typically, a kernel will need at least one input and one output buffer as well as other arguments. The arguments need to be set up one at a time for the kernel to access at execution time by calling the `clSetKernelArg()` function for each argument. The function is called with a number indexing a particular argument in the kernel function argument list. The first argument is passed with index 0, the second with index 1, etc.

With the arguments set, call the function `clEnqueueNDRangeKernel()` with the kernel object and a command queue to request that the kernel be executed. Once the kernel is enqueued, the host code can do other things, or it can wait for the kernel (and everything previously enqueued) to finish by calling the `clFinish()` function. This sample calls `clFinish()`, as it includes code to time the total kernel execution (including any enqueue overhead) in a loop needs to wait for each execution to finish before recording the final or contribution to the average time.

That's the bulk of what goes into an OpenCL application. There are some clean-up operations, such as calling `clReleaseKernel`, `clReleaseMemObject`, `clReleaseProgram`, etc. These are included in the sample, even though OpenCL should automatically release all resources when the program exits. A more complex program might wish to release resources in a timely fashion to avoid memory leaks.

A final word of caution: while this sample does not use "events," they can be very useful for more complex applications that wish, for example, to overlap CPU and GPU processing. However, it is very important to note that any `clEnqueueXXXXX()` function (where "XXXXX" is replaced with the name for one of many possible functions) that is passed a pointer to an event will allocate an event, and the calling application code is then responsible for calling `clReleaseEvent()` with a pointer to that event at some point. If this is not done, the program will experience a memory leak as events accumulate.

A common mistake is to use the `clCreateUserEvent()` function to allocate an event to pass to any `clEnqueueXXXX` function, thinking that OpenCL will signal that event when it completes. OpenCL will not use that event, and the `clEnqueueXXXX` will return a new event, overwriting the contents of the event variable passed by pointer. This is an easy way to create a memory leak. User events have a different purpose, beyond the scope of this sample. For more details on OpenCL events, please see the OpenCL specification.[1]

Project Structure

_tmain(argc, argv) - Main entry point function in the *Main.cpp* file.

Creates an instance of class **C_SampleOCLApp**.

Calls `C_SampleOCLApp::Run()` to start application.

That's all it does! See the C_MainApp and C_SampleApp classes below for more details.

class **C_MainApp** - Generic OpenCL application super-class in the *C_MainApp.h* file.

On construction, creates instance of OpenCL class **C_OCL**.

Defines a generic application "run" function:

`Run()`	`Run()` is a good starting point for reading the code to understand how an OpenCL application is initialized, run, and cleaned up. `Run()` calls virtual functions (see below) in a simple representative application sequence.

Declares virtual functions to be defined by **C_SampleOCLApp** (below):

`AppParseArgs ()`	Parse command line options
`AppUsage ()`	Print usage instructions
`AppSetup ()`	Application set up, including OpenCL set up
`AppRun ()`	Application specific operations
`AppCleanup ()`	Application clean up

class **C_SampleOCLApp** - Derived from **C_MainApp**, defines functions specific to this sample.

Implements application specific code for the **C_MainApp** virtual functions in the *SampleApp.cpp* and *SampleApp.h* files. (See class **C_MainApp** (above) for the virtual functions implemented.)

Defines "ToneMap" OpenCL kernel setup and execution functions in the *ToneMap_OCL.cpp* file:

`RunOclToneMap ()`	Does one-time set up for ToneMap, then calls ToneMap().
`ToneMap ()`	Sets ToneMapping kernel arguments and executes the kernel.

class **C_OCL** - Most of the host side OpenCL API set up and clean up code.

On construction, initializes OpenCL. On destruction, cleans up after OpenCL.

Defines OpenCL service functions in the *C_OCL.cpp* and *C_OCL.h* files:

`Start ()`	Sets up OpenCL device for Intel® Iris™ graphics for proper platform.
`ReadAllPlatforms ()`	Obtains all available OpenCL platforms, saving their names.
`MatchPlatformName ()`	Helper function, chooses a platform by name.
`GetDeviceType ()`	Helper function – is device type GPU or CPU?
`CheckExtension ()`	Checks if a particular OpenCL extension is supported on the current device.
`ReadExtensions ()`	Obtains a string listing all OpenCL extensions for the current device.
`SetCurrentDeviceType ()`	Sets desired device type and creates OpenCL context and command queue.
`CreateProgramFromFile ()`	Loads a file containing OpenCL kernels, creates an OpenCL program, and builds it.
`ReadSourceFile ()`	Reads OpenCL kernel source file into a string, ready to build as a program.
`CreateKernelFromProgram ( )`	Creates an OpenCL kernel from a previously built program.
`GetDeviceInfo ()`	Two helper functions to get device specific information: one allocates memory to receive and return results, the other returns results via a pointer to memory provided by the caller.
`ClearAllPlatforms ()`	Releases everything associated with a previously selected platform.
`ClearAllPrograms ()`	Releases all currently existing OpenCL programs.
`ClearAllKernels ()`	Releases all currently existing OpenCL kernels.

OpenCL APIs Used

`clBuildProgram`	`clCreateBuffer`
`clCreateCommandQueue`	`clCreateContext`
`clCreateKernel`	`clCreateProgramWithSource`
`clEnqueueMapBuffer`	`clEnqueueNDRangeKernel`
`clEnqueueUnmapMemObject`	`clFinish`
`clGetDeviceIDs`	`clGetDeviceInfo`
`clGetPlatformIDs`	`clGetPlatformInfo`

`clReleaseCommandQueue`	`clReleaseContext`
`clReleaseDevice`	`clReleaseDevice`
`clReleaseKernel`	`clReleaseMemObject`
`clReleaseProgram`	`clSetKernelArg`

Controlling the Sample

This sample is run from a Microsoft Windows* command line console. It supports the following command line and optional parameters:

ToneMapping.exe [? | --h] [-c|-g] [-list] [-p "platformName] [-i "full image filename"]

? OR --h	Prints this help message
-c	Runs OpenCL on CPU
-g	Runs OpenCL on GPU - default
-list	Displays list of platform name strings
-p "platformName"	Supplies a platform name (in quotes if it has spaces) to check for and use.
-i "full image filename"	Supplies an image file name (in quotes if it has spaces) to process.

References

1. OpenCL Specifications from Khronos.org:
   http://www.khronos.org/registry/cl/
2. Intel® SDK for OpenCL™ Applications: http://software.intel.com/en-us/vcsource/tools/opencl-sdk
3. *High Dynamic Range Tone Mapping Post Processing Effect*:
   http://software.intel.com/en-us/vcsource/samples/hdr-tone-mapping

Legal Information