Tools
Heterogeneous Computing (http://developer.amd.com/tools/he computing/)
Aparapi (http://developer.amd.com/tools computing/aparapi/)
APP Kernel Analyzer (http://developer.amd.com/tools computing/app-kernel-analyzer/)
APP Profiler (http://developer.amd.com/tools computing/amd-app-profiler/)
Accelerated Parallel Processing (APP) SDK (http://developer.amd.com/tools computing/amd-accelerated-parallel-processing-app-sdk/)
Downloads (http://developer.amd.com/to computing/amd-accelerated-parallel-processing-app-sdk/downloads/)
System Requirements & Driver Compatibility (http://developer.amd.com/to computing/amd-accelerated-parallel-processing-app-sdk/system-requirements/)
Documentation (http://developer.amd.com/to computing/amd-accelerated-parallel-processing-app-sdk/documentation/)
Intro OpenCL Tutorial (http://developer.amd.com computing/amd-accelerated-parallel-processing-app-sdk/introductory-tutorial-to-opencl/)
Samples & Demos (http://developer.amd.com/to computing/amd-accelerated-parallel-processing-app-sdk/samples-demos/)
Compute System Recommendations (http://developer.amd.com/to computing/amd-accelerated-parallel-

Intro OpenCL Tutorial

Benedict R. Gaster, AMD Architect, OpenCL™

OpenCL™ is a young technology, and, while a specification has been published (www.khronos.org/registry/cl/ (http://www.khronos.org/registry/cl/)), there are currently few documents that provide a basic introduction with examples. This article helps make OpenCL™ easier to understand and implement.

Note that:

- I work at AMD, and, as such, I will test all example code on our implementation for both Windows® and Linux®; however, my intention is to illustrate the use of OpenCL™ regardless of platform. All examples are written in pure OpenCL™ and should run equally well on any implementation.
- I have done my best to provide examples that work out-of-the-box on non-AMD implementations of OpenCL™, but I will not be testing them on non-AMD implementations; therefore, it is possible that an example might not work as expected on such systems. If this is the case, please let me know via our Stream Computing (http://forums.amd.com/devforum/categories.cfm?catid=328&entercat=y) forum, and I will do my best to rectify the code and publish an update.

The following “Hello World” tutorial provides a simple introduction to OpenCL™. I hope to follow up this first tutorial with additional ones covering topics such as:

- Using platform and device layers to build robust OpenCL™
- Program compilation and kernel objects
- Managing buffers
- Kernel execution
- Kernel programming – basics
- Kernel programming – synchronization
- Matrix multiply – a case study
- Kernel programming – built-ins

The “Hello World” program in OpenCL™

Here are some notes of caution on how the OpenCL™ samples are written:

- OpenCL™ specifies a host API that is defined to be compatible with C89 and does not make any mention of C++ or other programming language bindings. Currently, there are several efforts to develop bindings for other languages (see the links at the end of this article), and, specifically, there has been a strong push to develop C++ bindings (http://www.khronos.org/registry/cl/). In this and subsequent tutorials, I use the C++ bindings exclusively and describe OpenCL™ in these terms. See the OpenCL™ 1.0 specification for the corresponding C API. Alternatively, you can view the source for the C++ bindings to see what underlying OpenCL™ function is used, and with what arguments by the particular C++ binding.

- OpenCL™ defines a C-like language for programming compute device programs. These programs are passed to the OpenCL™ runtime via API calls expecting values of type ***char ****. Often, it is convenient to keep these programs in separate source files. For this and subsequent tutorials, I assume the device programs are stored in files with names of the form ***name_kernels.cl***, where ***name*** varies, depending on the context, but the suffix ***_kernels.cl*** does not. The corresponding device programs are loaded at runtime and passed to the OpenCL™ API. There are many alternative approaches to this; this one is chosen for readability.

For this first OpenCL™ program, we start with the source for the host application.

Header files

Just like any other external API used in C++, you must include a header file when using the OpenCL™ API. Usually, this is in the directory ***CL*** within the primary include directory. For the C++ bindings we have (replace the straight C API with ***cl.h***):

```
#include <utility>
#define __NO_STD_VECTOR // Use cl::vector instead of STL version
#include <CL/cl.hpp>
```

For our program, we use a small number of additional C++ headers, which are agnostic to OpenCL™.

processing-app-
sdk/compute-system-
recommendations/)

Open Source Libraries
(http://developer.amd.com/to
computing/amd-
accelerated-parallel-
processing-app-sdk/open-
source-libraries/)

Support
(http://developer.amd.com/to
computing/amd-
accelerated-parallel-
processing-app-
sdk/support/)

Accelerated Parallel
Processing Math Libraries
(APPML)
(http://developer.amd.com/tools
computing/amd-accelerated-
parallel-processing-math-
libraries/)

CodeAnalyst Performance
Analyzer
(http://developer.amd.com/tools
computing/amd-codeanalyst-
performance-analyzer/)

CodeXL
(http://developer.amd.com/tools
computing/codexl/)

gDEBugger
(http://developer.amd.com/tools
computing/amd-gdebugger/)

CPU Development
(http://developer.amd.com/tools/cp
development/)

Graphics Development
(http://developer.amd.com/tools/gr
development/)

Open Source
(http://developer.amd.com/tools/op
source/)

```
#include <cstdio>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <string>
#include <iterator>
```

As we will dynamically request an OpenCL™ device to return the “Hello World\n” string, we define it as a constant to use in calculations.

```
const std::string hw("Hello World\n");
```

Errors

A common property of most OpenCL™ API calls is that they either return an error code (type ***cl_int***) as the result of the function itself, or they store the error code at a location passed by the user as a parameter to the call. As with any API call that can fail, it is important, for the application to check its behavior correctly in the case of error. For the most part we will not concern ourselves with recovering from an error; for simplicity, we define a function, ***checkErr***, to see that a certain call has completed successfully. OpenCL™ returns the value ***CL_SUCCESS*** in this case. If it is not, it outputs a user message and exits; otherwise, it simply returns.

```
inline void
checkErr(cl_int err, const char * name)
{
    if (err != CL_SUCCESS) {
        std::cerr << "ERROR: " << name
                 << " (" << err << ")" << std::endl;
        exit(EXIT_FAILURE);
    }
}
```

A common paradigm for error handling in C++ is through the use of exceptions, and the OpenCL™ C++ bindings provide just such an interface. A later tutorial will cover the use of exceptions and other optional features provided by the C++ bindings. For now, let’s look at the one remaining function, “main,” necessary for our first OpenCL™ application.

OpenCL™ Contexts

The first step to initializing and using OpenCL™ is to create a *context*. The rest of the OpenCL™ work (creating devices and memory, compiling and running programs) is performed within this *context*. A *context* can have a number of associated devices (for example, CPU or GPU devices), and, within a *context*, OpenCL™ guarantees a relaxed memory consistency between devices. We will look at this in detail in a later tutorial; for now, we use a single device, ***CL_DEVICE_TYPE_CPU***, for the CPU device. We could have used *CL_DEVICE_TYPE_GPU* or some other support device type, assuming that the OpenCL™ implementation supports that device. But before we can create a *context* we must first queue the OpenCL runtime to determine which platforms, i.e. different vendor’s OpenCL implementations, are present. The class *cl::Platform* provides the static method cl::Platform::get for this and returns a list of platforms. For now we select the first platform and use this to create a *context*. The constructor *cl::Context* should be successful and, in this case, the value of *err* is ***CL_SUCCESS***.

```
int
main(void)
{
    cl_int err;
    cl::vector< cl::Platform > platformList;
    cl::Platform::get(&platformList);
    checkErr(platformList.size()!=0 ? CL_SUCCESS : -1, "cl::Platform::get");
    std::cerr << "Platform number is: " << platformList.size() << std::endl;

    std::string platformVendor;

    platformList[0].getInfo((cl_platform_info)CL_PLATFORM_VENDOR, &platformVendor);

    std::cerr << "Platform is by: " << platformVendor << "\n";

    cl_context_properties cprops[3] =

        {CL_CONTEXT_PLATFORM, (cl_context_properties)(platformList[0])(), 0};

    cl::Context context(

       CL_DEVICE_TYPE_CPU,

       cprops,

       NULL,

       NULL,

       &err);

    checkErr(err, "Conext::Context()");
```

Before delving into compute devices, where the ‘real’ work happens, we first allocate an OpenCL™ buffer to hold the result of the kernel that will be run on the device, i.e. the string “Hello World\n.” For now we simply allocate some memory on the host and request that OpenCL™ use this memory directly, passing the flag ***`CL_MEM_USE_HOST_PTR`***, when creating the buffer.

```
    char * outH = new char[hw.length()+1];
    cl::Buffer outCL(
        context,
        CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,
        hw.length()+1,
        outH,
        &err);
    checkErr(err, "Buffer::Buffer()");
```

OpenCL™ *Devices*

In OpenCL™ many operations are performed with respect to a given context. For example, buffers (1D regions of memory) and images (2D and 3D regions of memory) allocation are all context operations. But there are also device specific operations. For example, program compilation and kernel execution are on a per device basis, and for these a specific device handle is required. So how do we obtain a handle for a device? We simply query a context for it. OpenCL™ provides the ability to queue information about particular objects, and using the C++ API it comes in the form of ***`object.getInfo<CL_OBJECT_QUERY>()`***. In the specific case of getting the device from a context:

```
    cl::vector<cl::Device> devices;
    devices = context.getInfo<CL_CONTEXT_DEVICES>();
    checkErr(
        devices.size() > 0 ? CL_SUCCESS : -1, "devices.size() > 0");
```

Now that we have the list of associated devices for a context, in this case a single CPU device, we need to load and build the compute program (the program we intend to run on the device, or more generally: devices). The first few lines of the following code simply load the OpenCL™ device program from disk, convert it to a string, and create a ***cl::Program::Sources*** object using the helper constructor. Given an object of type ***cl::Program::Sources*** a ***cl::Program***, an object is created and associated with a context, then built for a particular set of *devices*.

```
    std::ifstream file("lesson1_kernels.cl");
    checkErr(file.is_open() ? CL_SUCCESS:-1, "lesson1_kernel.cl");

    std::string prog(

        std::istreambuf_iterator<char>(file),

        (std::istreambuf_iterator<char>()));

  cl::Program::Sources source(


        1,

       std::make_pair(prog.c_str(), prog.length()+1));

    cl::Program program(context, source);

    err = program.build(devices,"");

    checkErr(err, "Program::build()");
```

A given *program* can have many entry points, called kernels, and to call one we must build a kernel object. There is assumed to exist a straightforward mapping from kernel names, represented as strings, to a function defined with the __***kernel*** attribute in the compute program. In this case we can build a ***cl::kernel*** object, ***kernel***. Kernel arguments are set using the C++ API with ***kernel.setArg()***, which takes the index and value for the particular argument.

```
    cl::Kernel kernel(program, "hello", &err);
    checkErr(err, "Kernel::Kernel()");

    err = kernel.setArg(0, outCL);

    checkErr(err, "Kernel::setArg()");
```

Now that the boiler plate code is done, it is time to compute the result (the output buffer with the string "Hello World\n"). All device computations are done using a command queue, which is a virtual interface for the device in question. Each command queue has a one-to-one mapping with a given device; it is created with the associated *context* using a call to the constructor for the class ***cl::CommandQueue***. Given a ***cl::CommandQueue*** *queue*,kernels can be queued using ***queue.enqueuNDRangeKernel***. This queues a *kernel* for execution on the associated device. The kernel can be executed on a 1D, 2D, or 3D domain of indexes that execute in parallel, given enough resources. The total number of elements (indexes) in the launch domain is called the ***global*** work size; individual elements are known as ***work-items***. ***Work-items*** can be grouped into ***work-groups*** when communication between ***work-items*** is required. ***Work-groups*** are defined with a sub-index function (called the ***local*** work size), describing the size in each dimension corresponding to the dimensions specified for the global launch domain. There is a lot to consider with respect to kernel launches, and we will cover this in more detail in future tutorials. For now, it is enough to note that for Hello World, each work-item computes a letter in the resulting string; and it is enough to launch ***hw.length()+1***, where ***hw*** is the ***const std::string*** we defined at the beginning of the program. We need the extra ***work-item*** to account for the ***NULL*** terminator.

```
cl::CommandQueue queue(context, devices[0], 0, &err);
    checkErr(err, "CommandQueue::CommandQueue()");

    cl::Event event;

    err = queue.enqueueNDRangeKernel(

        kernel,

        cl::NullRange,

        cl::NDRange(hw.length()+1),

         cl::NDRange(1, 1),

        NULL,

        &event);

    checkErr(err, "ComamndQueue::enqueueNDRangeKernel()");
```

The final argument to the ***enqueueNDRangeKernel*** call above was a ***cl::Event*** object, which can be used to query the status of the command with which it is associated, (for example, it has completed). It supports the method ***wait()*** that blocks until the command has completed. This is required to ensure the kernel has finished execution before reading the result back into host memory with ***queue.enqueueReadBuffer()***. With the compute result back in host memory, it is simply a matter of outputting the result to ***stdout*** and exiting the program.

```
    event.wait();
    err = queue.enqueueReadBuffer(
        outCL,
        CL_TRUE,
        0,
        hw.length()+1,
        outH);
    checkErr(err, "ComamndQueue::enqueueReadBuffer()");
    std::cout << outH;
    return EXIT_SUCCESS;
}
```

Finally, to make the program complete an implementation for the device program (lesson1_kernels.cl), requires defining the external entry point, hello. The kernel implementation is straightforward: it calculates a unique index as a function of the launch domain using ***get_global_id()***, it uses it as an index into the string, ***hw***, then writes its value to the output array, ***out***.

```
#pragma OPENCL EXTENSION cl_khr_byte_addressable_store : enable
__constant char hw[] = "Hello World\n";
__kernel void hello(__global char * out)
{
    size_t tid = get_global_id(0);
    out[tid] = hw[tid];
}
```

For robustness, it would make sense to check that the thread id (tid) is not out of range of the hw; for now, we assume that the corresponding call to ***queue.enqueueNDRangeKernel()*** is correct.

Building and running

On Linux, it should be enough to use a single command to build the OpenCL™ program; for example:
gcc –o hello_world –Ipath-OpenCL-include –Lpath-OpenCL-libdir lesson1.cpp –lOpenCL

To run:
LD_LIBRARY_PATH=path-OpenCL-libdir ./hello_world

On Windows, with a Visual Studio command window, an example is:
cl /Fehello_world.exe /Ipath-OpenCL-include lesson.cpp path-OpenCL-libdir/OpenCL.lib

Let's assume that OpenCL.dll is on the path, then, running
.\hello_world

outputs the following string pm stdout:
Hello World

This completes our introductory tutorial to OpenCL™. Your feedback, comments, and questions are requested. Please visit our Stream forum (http://forums.amd.com/devforum/categories.cfm?catid=328&entercat=y).

Useful Links

The following list provides links to some specific programming bindings, other than C, for OpenCL™. I have not tested these and cannot vouch for their correctness, but hope they will be useful:

- OpenCL™ specification and headers:
  http://www.khronos.org/registry/cl/ (http://www.khronos.org/registry/cl/)
- OpenCL™ technical forum:
  http://www.khronos.org/message_boards/viewforum.php?f=28 (http://www.khronos.org/message_boards/viewforum.php?f=28)
- The C++ bindings used in this tutorial can be found on the OpenCL™ web page at Khronos, along with complete documentation:
  http://www.khronos.org/registry/cl/ (http://www.khronos.org/registry/cl/)
- Python bindings can be found here:
  http://pyopencl.next-touch.com/ (http://pyopencl.next-touch.com/)
- C# bindings can be found here:
  http://www.khronos.org/message_boards/viewtopic.php?f=28&t=1932 (http://www.khronos.org/message_boards/viewtopic.php?f=28&t=1932)
- An Introduction to OpenCL™:
  http://ati.amd.com/technology/streamcomputing/intro_opencl.html (http://ati.amd.com/technology/streamcomputing/intro_opencl.html)

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