NVIDIA® OpenCL™ JumpStart Guide

Technical Brief

Version 1.0

February 19, 2010

Introduction

The purposes of this guide are to assist developers who are familiar with CUDA C/C++ development and want to port code to OpenCL, and to assist developers who want to use CUDA C/C++ now in a fashion that provides the easiest transition later to OpenCL.

Examples and explanations are provided to illustrate implementation of a simple GPU-accelerated application using CUDA C/C++ compute kernels and the CUDA Driver API, with direct comparison to the OpenCL implementation. A summary is also provided for the most significant CUDA C/C++ capabilities that are not supported in OpenCL (or may not be supported in all vendors implementations of OpenCL).

Overview

OpenCL (Open Compute Language) is an open standard for parallel programming of heterogeneous systems, managed by the Khronos Group. OpenCL supports a wide range of applications, from embedded and consumer software to HPC solutions, through a low-level, high-performance, portable abstraction. By creating an efficient, close-to-the-metal programming interface, OpenCL will form the foundation layer of a parallel computing ecosystem of platform-independent tools, middleware and applications.

CUDA is NVIDIA's technology for GPU Computing. With the CUDA architecture and tools, developers are achieving dramatic speedups in fields such as medical imaging and natural resource exploration, and creating breakthrough applications in areas such as image recognition and real-time HD video playback and encoding.

Leveraging the massively parallel processing power of NVIDIA GPUs, OpenCL running on the CUDA architecture extends NVIDIA's world-renowned graphics processor technology into the realm of parallel computing. Applications that run on the CUDA architecture can take advantage of an installed base of over one hundred million CUDA-enabled GPUs in desktop and notebook computers, professional workstations, and supercomputer clusters. NVIDIA GPUs enable this unprecedented performance via standard APIs such as OpenCL and DirectX Compute, and high level programming languages such as C/C++, Fortran, Java, Python, and .NET.

The NVIDIA CUDA Driver API and Runtime API are the original API's that allowed programmers to develop applications for the CUDA architecture and are predecessors of OpenCL. The CUDA Driver API is very similar to OpenCL with a high correspondence between functions.

Note: Developers interested in porting between CUDA API's and OpenCL should focus upon the CUDA **Driver** API rather than the the CUDA Runtime API due to similarities between the CUDA Driver API and OpenCL. Choosing the CUDA **Driver** API and the guidelines explained in this document will allow the smoothest transitions to/from OpenCL.

Getting Started

To get started, follow the steps in the CUDA Quickstart Guide for your operating system, and read through the rest of this document. CUDA Quickstart Guides are available at:
http://www.nvidia.com/object/cuda_develop.html

Note: You must have a CUDA-enabled GPU in your system. All recent NVIDIA GPUS have the necessary support, and a full list is available here:
http://www.nvidia.com/object/cuda_learn_products.html

General Differences between OpenCL and the CUDA Driver API

This section describes several key differences between the CUDA Driver API and OpenCL, as guidance to developers who plan to support both or port from one to the other. These differences might shift over time as either OpenCL or the CUDA Driver API are revised. Please also refer to the OpenCL Programming Guide, the OpenCL Specification v1.0 and the CUDA Programming Guide for additional details.

C Language Integration

CUDA C/C++ has features supported via a dedicated compiler that are not part of OpenCL. These provide features such offline compilation of host and device code from a single source file and sharing of user-defined types between the host and device. The CUDA C/C++ language supports the C memory model in a manner C/C++ programmers will find very familiar, exposing device memory via a pointer rather than an opaque handle. This permits arbitrary pointer dereferencing, device memory pointer arithmetic on both the device and the host, and no requirement for address space typed pointers.

Pointer Traversal

Multiple pointer traversals must be avoided on OpenCL, the behavior of such operations is undefined in the specification. Pointer traversals are allowed with CUDA C/C++.

```
struct Node { Node* next; }
n = n->next;    // undefined operation in OpenCL,
                // since 'n' here is a kernel input
```

To do this on OpenCL, pointers must be converted to be relative to the buffer base pointer and only refer to data within the buffer itself (no pointers between OpenCL buffers are allowed).

```
struct Node { unsigned int next; }
…
n = bufBase + n; // pointer arithmetic is fine, bufBase is
                 // a kernel input param to the buffer's beginning
```

Kernel Programs

Using CUDA C/C++, kernel programs are precompiled into a binary format and there are function calls for dealing with module and function loading. In OpenCL, the compiler is built into the runtime and can be invoked on the raw text or a binary can be built and saved for later load. There are slight differences in keywords and syntax of the languages used for kernels.

Kernel Invocation Memory Offsets

The current version of OpenCL does not support stream offsets at the API/kernel invocation level. Offsets must be passed in as a parameter to the kernel and the address of the memory computed inside it. CUDA kernels may be started at offsets within buffers at the API/kernel invocation level.

C++

CUDA C/C++ allows C++ constructs in device/kernel code including Default Parameters, Operator Overloading, Namespaces and Function Templates. These aren't presently supported by OpenCL.

Automatic Demotion for Devices without native Double Precision

With CUDA C/C++, when compiling for devices without native double-precision floating-point support (such as devices of compute capability 1.2 and lower), double variables get converted to single-precision floating-point format (but retain their size of 64 bits) and double-precision floating-point arithmetic gets demoted to single-precision floating-point arithmetic. Current OpenCL compilers do not implement this automatic double-to-float demotion.

Multiple Device support

OpenCL allows multiple device management from within one context using multiple command queues. CUDA C/C++ supports multiple devices using multiple contexts, with one context per device.

API Command ordering

OpenCL supports explicit ordering of API commands within a command queue using event dependencies. CUDA C/C++ enforces API command order as-issued within a stream.

Thread Safety

Applications using OpenCL should not rely on the drivers being thread-safe at present. The OpenCL v1.0 specification does not require vendor implementations to be thread-safe, and there are no conformance tests that verify thread-safety of any particular implementation.

Debugging

CUDA C/C++ allows debugging in device code from Visual Studio, so a developer can set breakpoints and use other debugger features to interactively debug kernels. At this time, NVIDIA's OpenCL implementation and associated tools don't provide for this.

Libraries

Several highly-optimized libraries are available for use with CUDA C/C++, including cublas, cufft, cuddp, npp, nvcuvid, MAGMA, CULA and GPU-VSIPL. Equivalent libraries compatible with existing OpenCL implementations may become available in the future but aren't available at present.

Advanced Features

CUDA C/C++ has more extensive support at present for some advanced features such as linear-memory-bound 1D texture fetches from kernel and warp vote functions from kernel code. Over time, such features in CUDA C/C++ can reasonably be expected to be available in extensions in NVIDIA's OpenCL implementation or in future versions of the OpenCL specification. NVIDIA is actively working with the Khronos organization on future versions of OpenCL.

Vector Addition Example

Here we show the differences between CUDA C/C++ and OpenCL implementations of vector addition. The program adds two arrays of floats. The basic components of this program are identical in CUDA C/C++ and OpenCL:

- A compute kernel, which will be executed in a massively parallel fashion on the compute device (GPU). Each thread (also known as *work item*) executes the same kernel computation on different data, adding an element from each of input arrays a and b and placing the result in a corresponding element of array c.
- A host application drives the kernel execution.

CUDA C/C++ Kernel Code:

```
__global__ void
vectorAdd(const float * a, const float * b, float * c)
{
    // Vector element index
    int nIndex = blockIdx.x * blockDim.x + threadIdx.x;
    c[nIndex] = a[nIndex] + b[nIndex];
}
```

OpenCL Kernel Code

```
__kernel void
vectorAdd(__global const float * a,
          __global const float * b,
          __global       float * c)
{
    // Vector element index
    int nIndex = get_global_id(0);
    c[nIndex] = a[nIndex] + b[nIndex];
}
```

As can be seen from the kernel code, both languages are conceptually very similar. For this program, the differences are mostly in the syntax. Let's look at these differences in detail.

Kernel declaration specifier

CUDA C/C++ kernel functions are declared using the "__global__" function modifier, while OpenCL kernel functions are declared using "__kernel".

Pointer declaration specifiers

With OpenCL, it is mandatory to specify the address space for any pointers passed as arguments to kernel functions. This kernel has three parameters *a, b,* and *c* that are pointers to global device memory. These arrays must be declared using the __global specifier in OpenCL.

Global thread index computation

In CUDA C/C++, all index and threadblock size information is available to kernels in three structures: `threadIdx.{x|y|z}`, `blockIdx.{x|y|z}`, `blockDim.{x|y|z}` and `gridDim.{x|y|z}`. The kernel developer is responsible for implementing the index computations necessary for the kernel to operate on its data.

In contrast, OpenCL provides basic index information to kernels via functions. OpenCL also provides several functions to access derived information such as `get_global_id()`. This function computes a global work item index from work group index, work group size and thread index. OpenCL also provides the function get_local_id() to query the id inside the work group, `get_work_dim()` to query the number of dimension of the work group launched for the kernel and the `get_global_size()` function to query the size of the work group.

CUDA Driver API Host Code:

The vectorAdd example is a very basic CUDA C/C++ program that adds two arrays together. The CUDA driver API is a lower level API that offers a better level of control for CUDA applications. It is language independent since it can deal directly with PTX or CUBIN objects. PTX or CUBIN files generated by NVCC.EXE can be loaded using the CUDA Driver API.

This example assumes that the CUDA C/C++ kernel previously shown has been successfully compiled via NVCC.exe into a CUBIN file named "vectorAdd.cubin".

```
// Kernel launch configuration
const unsigned int cnBlockSize = 512;
const unsigned int cnBlocks    = 3;
const unsigned int cnDimension = cnBlocks * cnBlockSize;

CUdevice    hDevice;
CUcontext   hContext;
CUmodule    hModule;
CUfunction  hFunction;

// create CUDA device & context, and load the kernel
cuInit(0);
cuDeviceGet(&hContext, 0); // pick first device
cuCtxCreate(&hContext, 0, hDevice));
cuModuleLoad(&hModule, "vectorAdd.cubin");
cuModuleGetFunction(&hFunction, hModule, "vectorAdd");
```

```
// allocate host vectors
float * pA = new float[cnDimension];
float * pB = new float[cnDimension];
float * pC = new float[cnDimension];

// initialize host memory (using helper C function called “randomInit”)
randomInit(pA, cnDimension);
randomInit(pB, cnDimension);

// allocate memory on the device
CUdeviceptr pDeviceMemA, pDeviceMemB, pDeviceMemC;
cuMemAlloc(&pDeviceMemA, cnDimension * sizeof(float));
cuMemAlloc(&pDeviceMemB, cnDimension * sizeof(float));
cuMemAlloc(&pDeviceMemC, cnDimension * sizeof(float));

// copy host vectors to device
cuMemcpyHtoD(pDeviceMemA, pA, cnDimension * sizeof(float));
cuMemcpyHtoD(pDeviceMemB, pB, cnDimension * sizeof(float));

// setup parameter values
cuFuncSetBlockShape(hFunction, cnBlockSize, 1, 1);
cuParamSeti(hFunction, 0, pDeviceMemA);
cuParamSeti(hFunction, 4, pDeviceMemB);
cuParamSeti(hFunction, 8, pDeviceMemC);
cuParamSetSize(hFunction, 12);

// execute kernel
cuLaunchGrid(hFunction, cnBlocks, 1);

// copy the result from device back to host
cuMemcpyDtoH((void *) pC, pDeviceMemC, cnDimension * sizeof(float));

// cleanup
delete[] pA;
delete[] pB;
delete[] pC;
cuMemFree(pDeviceMemA);
cuMemFree(pDeviceMemB);
cuMemFree(pDeviceMemC);
```

OpenCL Host Code:

Let’s compare the Host Code from the CUDA Driver API to the OpenCL one below. The code below assumes that the OpenCL kernel code from below is stored in a string named “`sProgramSource`”.

```
// Kernel launch configuration
const unsigned int cnBlockSize = 512;
const unsigned int cnBlocks    = 3;
const unsigned int cnDimension = cnBlocks * cnBlockSize;

// Get OpenCL platform count
cl_uint NumPlatforms;
clGetPlatformIDs (0, NULL, &NumPlatforms);
```

```
// Get all OpenCL platform IDs
cl_platform_id* PlatformIDs;
PlatformIDs = new cl_platform_id[NumPlatforms];
clGetPlatformIDs(NumPlatforms, PlatformIDs, NULL);

// Select NVIDIA platform (this example assumes it IS present)
char cBuffer[1024];
cl_uint NvPlatform;
for(cl_uint i = 0; i < NumPlatforms; ++i)
{
  clGetPlatformInfo (PlatformIDs[i], CL_PLATFORM_NAME, 1024, cBuffer, NULL);
  if(strstr(cBuffer, "NVIDIA") != NULL)
  {
     NvPlatform = i;
     break;
  }
}

// Get a GPU device on Platform (this example assumes one IS present)
cl_device_id cdDevice;
clGetDeviceIDs(PlatformIDs[NvPlatform], CL_DEVICE_TYPE_GPU, 1,
               &cdDevice, NULL);

// Create a context
cl_context hContext;
hContext = clCreateContext(0, 1, &cdDevice, NULL, NULL, NULL);

// Create a command queue for the device in the context
cl_command_queue hCmdQueue;
hCmdQueue = clCreateCommandQueue(hContext, cdDevice, 0, NULL);

// Create & compile program
cl_program hProgram;
hProgram = clCreateProgramWithSource(hContext, 1, sProgramSource, 0, 0);
clBuildProgram(hProgram, 0, 0, 0, 0, 0);

// Create kernel instance
cl_kernel hKernel;
hKernel = clCreateKernel(hProgram, “vectorAdd”, 0);

// Allocate host vectors
float * pA = new float[cnDimension];
float * pB = new float[cnDimension];
float * pC = new float[cnDimension];

// Initialize host memory (using helper C function called “randomInit”)
randomInit(pA, cnDimension);
randomInit(pB, cnDimension);

// Allocate device memory (and init hDeviceMemA and hDeviceMemB)
cl_mem hDeviceMemA, hDeviceMemB, hDeviceMemC;
hDeviceMemA = clCreateBuffer(hContext,
                         CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                         cnDimension * sizeof(cl_float), pA, 0);
hDeviceMemB = clCreateBuffer(hContext,
                         CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                         cnDimension * sizeof(cl_float), pB, 0);
hDeviceMemC = clCreateBuffer(hContext,
                         CL_MEM_WRITE_ONLY,
                         cnDimension * sizeof(cl_float), 0, 0);
```

```
// Setup parameter values
clSetKernelArg(hKernel, 0, sizeof(cl_mem), (void *)&hDeviceMemA);
clSetKernelArg(hKernel, 1, sizeof(cl_mem), (void *)&hDeviceMemB);
clSetKernelArg(hKernel, 2, sizeof(cl_mem), (void *)&hDeviceMemC);

// Launch kernel
clEnqueueNDRangeKernel(hCmdQueue, hKernel, 1, 0, &cnDimension, 0, 0, 0, 0);

// Copy results from device back to host;  block until complete
clEnqueueReadBuffer(hContext, hDeviceMemC, CL_TRUE, 0,
                    cnDimension * sizeof(cl_float), pC, 0, 0, 0);

// Cleanup
delete[] pA;
delete[] pB;
delete[] pC;
delete[] PlatformIDs;
clReleaseKernel(hKernel);
clReleaseProgram(hProgram);
clReleaseMemObj(hDeviceMemA);
clReleaseMemObj(hDeviceMemB);
clReleaseMemObj(hDeviceMemC);
clReleaseCommandQueue(hCmdQueue);
clReleaseContext(hContext);
```

API Differences

Both CUDA C/C++ and OpenCL implementations perform the same steps conceptually. The main differences are the naming schemes and how data gets passed to the API. Both OpenCL and the CUDA Driver API require the developer to manage the contexts and parameter passing.

One noteworthy difference is that CUDA C/C++ programs are compiled with an external tool (the NVCC compiler) before executing on the final application. This compilation step is typically performed when the actual application is built. Typically, the OpenCL compiler is invoked at runtime and the programmer needs to create or obtain the strings with the kernel programs. It is also possible to offline compile OpenCL source in a similar fashion to CUDA C/C++.

The following sections cover the API differences per program section.

Initialization, Context and Device Creation

CUDA Driver API and OpenCL both have to concept of a "Context". Any resources involved in executing compute code using either of the APIs will belong to a Context. One of the first steps for any compute program is to create such a context.

Using the CUDA Driver API:

Before any function calls to the CUDA driver API can be made, CUDA needs to be initialized with a call to `cuInit(0);`

In future versions of CUDA, `cuInit( )` will also include initialization flags as parameters. The current versions of CUDA require 0 (Zero) to be passed.

In CUDA a context is created for a specific device. The typical flow is to first query the CUDA devices available on a given system, get a handle to the device one wants to execute the CUDA C/C++ode on and create a context on that device. The vectorAdd sample uses a simplified version of this workflow and simply picks the first CUDA device (device 0):

```
cuInit(0);
cuDeviceGet(&hContext, 0);
cuCtxCreate(&hContext, 0, hDevice));
```

Using OpenCL:

OpenCL does not require global initialization of the library, but it does require a few additional setup steps due to the diverse scope of device types and driver implementations embraced by the OpenCL standard.

Using OpenCL, one obtains the Platform ID and Device ID, and then creates a Context. Additionally, OpenCL introduces the concept of Command Queues. Commands launching kernels and reading or writing memory are always issued for a specific Command Queue. A Command Queue is created on a specific device in a context.

```
// Get OpenCL platform count
cl_uint NumPlatforms;
clGetPlatformIDs (0, NULL, &NumPlatforms);

// Get all OpenCL platform IDs
cl_platform_id* PlatformIDs;
PlatformIDs = new cl_platform_id[NumPlatforms];
clGetPlatformIDs (NumPlatforms, PlatformIDs, NULL);

// Select NVIDIA platform (this example assumes it IS present)
char cBuffer[1024];
cl_uint NvPlatform;
for(cl_uint i = 0; i < NumPlatforms; ++i)
{
  clGetPlatformInfo (PlatformIDs[i], CL_PLATFORM_NAME, 1024, cBuffer, NULL);
  if(strstr(cBuffer, "NVIDIA") != NULL)
  {
     NvPlatform = i;
     break;
  }
}

//Get a GPU device on Platform (this example assumes one IS present)
cl_device_id cdDevice;
clGetDeviceIDs(PlatformIDs[NvPlatform], CL_DEVICE_TYPE_GPU, 1, cdDevice, NULL);

//Create a context
cl_context hContext;
hContext = clCreateContext(0, 1, &cdDevice, NULL, NULL, NULL);


//Create a command queue for the device in the context
cl_command_queue hCmdQueue;
hCmdQueue = clCreateCommandQueue(hContext, cdDevice, 0, NULL);
```

At this point, the CUDA Driver API and OpenCL programs are ready to create a compute kernel, upload data to the GPU device's memory and process it by launching a compute kernel on the device.

Kernel Creation

The following sections discuss how kernels are created using the CUDA Driver API and OpenCL.

Using the CUDA Driver API:

CUDA kernel code is typically stored in a separate file and compiled to binary format (using the NVCC compiler). This is similar to compiling a C file to object code. The result of this compilation step is a CUBIN file, which is loaded by an application at runtime using the `cuModuleLoad()` function.

A handle to a specific kernel in a CUBIN module is obtained via a string lookup of the kernel function's name. The code for module loading and accessing the kernel function assumes that the `vectorAdd.cu` kernel code has been compiled to `vectorAdd.cubin`:

```
CUmodule hModule;
cuModuleLoad(&hModule, “vectorAdd.cubin”);
cuModuleGetFunction(&hFunction, hModule, "vectorAdd");
```

Using OpenCL:

OpenCL is different from CUDA C/C++ in that OpenCL does not provide a standalone compiler for creating device ready binary code. The OpenCL interface provides methods for compiling kernels given a string containing the kernel code (`clCreateProgramWithSource()`) at runtime. Once a kernel is compiled it can be launched on the device.

> **Note:** The OpenCL API also provides methods to access a program's binaries after successful compilation, as well as methods to create program objects from such binaries. Using those methods, it is theoretically possible for a developer to re-create the tools for a workflow like the CUDA one using the OpenCL API, where a separate compile (implemented based on the OpenCL library) is used to compile binaries which the application loads during runtime. This approach would allow applications to avoid lengthy compilations every time they are launched by caching the kernel binaries on disk and only recompiling if the binaries for a specific device are not already in cache. But it should be noted that NVIDIA's OpenCL implementation does **not** currently support this.

In summary, the most straight forward process is to compile the kernels at runtime and this is what the following code does:

```
// create a program object and compile/build the device code
cl_program hProgram;
hProgram = clCreateProgramWithSource(hContext, 1, sProgramSource, 0, 0);
clBuildProgram(hProgram, 0, 0, 0, 0, 0);

// create a kernel instance
cl_kernel hKernel;
hKernel = clCreateKernel(hProgram, “vectorAdd”, 0);
```

The `clCreateProgramWithSource()` function creates a program object. sProgramSource is a C string containing the kernel source code. `clBuildProgram()` compiles the kernel source into binary code suited for the context's devices (it is possible to restrict compilation to a subset of a context's devices by passing a non-zero pointer to a list of device descriptors).

`clCreateKernel()` returns a handle to an instance of the kernel given a string with the kernel function's name within the program object that has been built.

Device Memory Allocation

This section covers how memory is allocated on the device. The vectorAdd example allocates arrays of float (in global device memory) for the three vectors (A, B, C) involved in the addition C = A+B.

CUDA's device memory for the Driver API's management functions are modeled after the C runtime's malloc(), free(), and memcpy() functions. The following code allocates three buffers of appropriate size to hold the three arrays and fills the two input vectors (A, B) with data prepared on the host via a host-to-device copy.

Using the CUDA Driver API:

We use `cuMemcpyHtoD()` to copy data from host to device.

```
CUdeviceptr pDeviceMemA, pDeviceMemB, pDeviceMemC;
cuMemAlloc(&pDeviceMemA, cnDimension * sizeof(float));
cuMemAlloc(&pDeviceMemB, cnDimension * sizeof(float));
cuMemAlloc(&pDeviceMemC, cnDimension * sizeof(float));

// copy host vectors to device
cuMemcpyHtoD(pDeviceMemA, pA, cnDimension * sizeof(float));
cuMemcpyHtoD(pDeviceMemB, pB, cnDimension * sizeof(float));
```

Using OpenCL:

OpenCL's device memory is managed via "buffer objects". Buffer objects are created via the clCreateBuffer() function, which offers a richer set of parameters than CUDA memory management functions: Buffer objects can be flagged as read and write-only, and it's even possible to specify a host memory region to be used by the device directly.

OpenCL buffer creation also allows for passing a host pointer to the data to be copied into the new buffer, all in one call; the following code shows the buffer creation for the three device memory region for vector A, B, C. A and B are being filled with data from the host, pointed to by pA, and pB. Since vector C is there to receive the results, it is not getting prefilled with data.

```
cl_mem hDeviceMemA, hDeviceMemB, hDeviceMemC;

hDeviceMemA = clCreateBuffer(hContext,
                             CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                             cnDimension * sizeof(cl_float), pA, 0);
hDeviceMemB = clCreateBuffer(hContext,
                             CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                             cnDimension * sizeof(cl_float), pB, 0);
hDeviceMemC = clCreateBuffer(hContext,
                             CL_MEM_WRITE_ONLY,
                             cnDimension * sizeof(cl_float), 0, 0);
```

Kernel Parameter Specification

The next step in preparing the kernels for launch is to establish a mapping between the kernels' parameters, essentially pointers to the three vectors A, B and C, to the three device memory regions, which were allocated in the previous section.

Parameter setting in both APIs is a pretty low-level affair. It requires knowledge of the total number, order, and types of a given kernel's parameters. The order and types of the parameters are used to determine a specific parameter's offset inside the data block made up of all parameters. The offset in bytes for the n^{th} parameter is essentially the sum of the sizes of all (n-1) preceding parameters.

Using the CUDA Driver API:

In CUDA device pointers are represented as `unsigned int` and the CUDA Driver API has a dedicated method for setting that type. Here's the code for setting the three parameters. Note how the offset is incrementally computed as the sum of the previous parameters' sizes.

```
cuParamSeti(hFunction, 0, pDeviceMemA);
cuParamSeti(hFunction, 4, pDeviceMemB);
cuParamSeti(hFunction, 8, pDeviceMemC);
cuParamSetSize(hFunction, 12);
```

Using OpenCL:

In OpenCL parameter setting is done via a single function that takes a pointer to the location of the parameter to be set.

```
clSetKernelArg(hKernel, 0, sizeof(cl_mem), (void *)&hDeviceMemA);
clSetKernelArg(hKernel, 1, sizeof(cl_mem), (void *)&hDeviceMemB);
clSetKernelArg(hKernel, 2, sizeof(cl_mem), (void *)&hDeviceMemC);
```

Kernel Launch

Launching a kernel requires the specification of the dimension and size of the "thread-grid". The CUDA Programming Guide and the OpenCL specification contain details about the structure of those grids. For NVIDIA GPUs the permissible structures are the same for CUDA and OpenCL.

For the vectorAdd sample we need to start one thread per vector-element (of the output vector). The number of elements in the vector is given in the `cnDimension` variable. It is defined to be `cnDimension = cnBlockSize * cnBlocks`. This means that `cnDimension` threads need to be executed. The threads are structured into `cnBlocks` one-dimensional thread blocks of size `cnBlockSize`.

Using the CUDA Driver API:

A kernel's block size is specified in a call separate from the actual kernel launch using `cuFunctSetBlockShape`. The kernel launching function `cuLaunchGrid` then only specifies the number of blocks to be launched.

```
cuFuncSetBlockShape(hFunction, cnBlockSize, 1, 1);
cuLaunchGrid       (hFunction, cnBlocks, 1);
```

Using OpenCL:

The OpenCL equivalent of kernel launching is to "enqueue" a kernel for execution into a command queue. The enqueue function takes parameters for both the work group size (work group is the OpenCL equivalent of a CUDA thread-block), and the global work size, which is the size of the global array of threads.

> **Note:** Where in CUDA the global work size is specified in terms of number of thread blocks, it is given in number of threads in OpenCL.

Both work group size and global work size are potentially one, two, or three dimensional arrays. The function expects pointers of `unsigned ints` to be passed in the fourth and fifth parameters. For the vectorAdd example, work groups and total work size is a one-dimensional grid of threads.

```
clEnqueueNDRangeKernel(hCmdQueue, hKernel, 1, 0,
                       &cnDimension, &cnBlockSize, 0, 0, 0);
```

The parameters of `cnDimension` and `cnBlockSize` must be pointers to unsigned int. Work group sizes that are dimensions greater than 1, the parameters will be a pointer to arrays of sizes.

Cleanup/Teardown

The CUDA Driver API and OpenCL both provide teardown functions for objects created in the application. For OpenCL, there are a few additional objects to deallocate associated with platform

Using the CUDA Driver API:

```
// cleanup
delete[] pA;
delete[] pB;
delete[] pC;
cuMemFree(pDeviceMemA);
cuMemFree(pDeviceMemB);
cuMemFree(pDeviceMemC);
```

Using OpenCL:

```
// Cleanup
delete[] pA;
delete[] pB;
delete[] pC;
delete[] PlatformIDs;
clReleaseKernel(hKernel);
clReleaseProgram(hProgram);
clReleaseMemObj(hDeviceMemA);
clReleaseMemObj(hDeviceMemB);
clReleaseMemObj(hDeviceMemC);
clReleaseCommandQueue(hCmdQueue);
clReleaseContext(hContext);
```

Retrieving Results to Host from Device

Both kernel launch functions (CUDA and OpenCL) are asynchronous, i.e. they return immediately after scheduling the kernel to be executed on the GPU. In order for a copy operation that retrieves the result vector C (copy from device to host) to produce correct results in synchronization with the kernel completion needs to happen.

CUDA memcpy functions automatically synchronize and complete any outstanding kernel launches proceeding. Both API's also provide a set of asynchronous memory transfer functions which allows a user to overlap memory transfers with computation to increase throughput.

Using the CUDA Driver API:

Use `cuMemcpyDtoH()` to copy results back to the host.

```
cuMemcpyDtoH((void *)pC, pDeviceMemC, cnDimension * sizeof(float));
```

Using OpenCL:

OpenCL's `clEnqueueReadBuffer()` function allows the user to specify whether a read is to be synchronous or asynchronous (third argument). For the simple vectorAdd sample a synchronizing read is used, which results in the same behavior as the simple synchronous CUDA memory copy above:

```
clEnqueueReadBuffer(hContext, hDeviceC, CL_TRUE, 0,
                    cnDimension * sizeof(cl_float),
                    pC, 0, 0, 0);
```

When used for asynchronous reads, OpenCL has an event mechanism that allows the host application to query the status or wait for the completion of a given call.

▪

Additional Resources

Resource	URL
Khronos OpenCL Homepage	http://www.khronos.org/opencl
OpenCL 1.0 Specification	http://www.khronos.org/registry/cl
OpenCL at NVIDIA	http://www.nvidia.com/object/cuda_opencl.html
CUDA Driver	http://www.nvidia.com/object/cuda_get.html
CUDA Toolkit	http://www.nvidia.com/object/cuda_get.html
CUDA SDK	http://www.nvidia.com/object/cuda_get.html
CUDA Reference Guide	http://www.nvidia.com/object/cuda_develop.html
CUDA Programming Guide	http://www.nvidia.com/object/cuda_develop.html
CUDA Zone	http://www.nvidia.com/cuda
Developer Forums	http://forums.nvidia.com/index.php?showforum=62
CUDA Visual Profiler	http://www.nvidia.com/object/cuda_get.html
CUDA GDB	http://www.nvidia.com/object/cuda_get.html
NVIDIA Nexus	http://developer.nvidia.com/object/nexus.html

For more information about GPU Computing with OpenCL and other technologies, please visit www.nvidia.com/cuda.

Notice

Trademarks

Copyright

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com