Lecture 26 - Programming with OpenCL

ECE 459: Programming for Performance

Jon Eyolfson

University of Waterloo

March 14, 2012

Introduction

Today, we'll see how to program with OpenCL

- We're using OpenCL 1.1
- There is a lot of initialization and querying
- When you compile your program, make sure to include the `-lOpenCL` flag

You can find the official documentation here:
`http://www.khronos.org/opencl/`
More specifically:
`http://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/`

Let's just dive into an example

Reminder

- All of the data is in an **NDRange**
- The range can be divided into **Work-Groups** (software)
- The work-groups run on **wavefronts/warps** (hardware)
- Each wavefront/warp executes **Work-Items**

All branches in a wavefront/warp should execute the same path

If an iteration of a loop takes `t` if one work-item executes 100 iterations, the total time to complete the wavefront/warp is `100t`

Simple Example (1)

```
#include <CL/cl.h>
#include <stdio.h>

#define NWITEMS 512

// A simple memset kernel
const char *source =
"__kernel void memset( __global uint *dst )                   \n"
"{                                                            \n"
"    dst[get_global_id(0)] = get_global_id(0);                \n"
"}                                                            \n";

int main(int argc, char ** argv)
{
   // 1. Get a platform.
   cl_platform_id platform;
   clGetPlatformIDs(1, &platform, NULL);
```

Explanation (1)

- Include the OpenCL header
- Request a platform (also known as hosts)
- A platform contains *compute devices*
    - May be GPU or CPU devices, etc.

Simple Example (2)

```
// 2. Find a gpu device.
cl_device_id device;
clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU,
               1,
               &device,
               NULL);

// 3. Create a context and command queue on that device.
cl_context context = clCreateContext(NULL,
                                     1,
                                     &device,
                                     NULL, NULL, NULL);
cl_command_queue queue = clCreateCommandQueue(context,
                                              device,
                                              0, NULL);
```

Explanation (2)

- We look for our GPU device that we wish to use
- We request a OpenCL context (which reprsents all of OpenCL's state)
- Create a command-queue, we get OpenCL to do work by telling it to run a kernel in the queue

Simple Example (3)

```
// 4. Perform runtime source compilation, and obtain
//    kernel entry point.
cl_program program = clCreateProgramWithSource(context,
                                               1,
                                               &source,
                                               NULL,
                                               NULL);
clBuildProgram(program, 1, &device, NULL, NULL, NULL);
cl_kernel kernel = clCreateKernel(program, "memset",
                                  NULL);

// 5. Create a data buffer.
cl_mem buffer = clCreateBuffer(context,
                               CL_MEM_WRITE_ONLY,
                               NWITEMS * sizeof(cl_uint),
                               NULL, NULL);
```

Explanation (3)

- We create an OpenCL program (what runs on the compute unit)
  - kernels
  - functions
  - declarations
- In this case, we create a kernel called `memset` from `source`
- OpenCL may also create programs from binaries (may be intermediate representation)

- Next, we need a *data buffer* (enables comunication between devices)
- This program does not have any input, so we don't put anything into the buffer, just declare its size

Simple Example (4)

```
// 6. Launch the kernel. Let OpenCL pick the local work
//    size.
size_t global_work_size = NWITEMS;
clSetKernelArg(kernel,0, sizeof(buffer), (void*)&buffer);
clEnqueueNDRangeKernel(queue,
                       kernel,
                       1, // dimensions
                       NULL, // initial offsets
                       &global_work_size, // number of
                                          // work-items
                       NULL, // work-items per work-group
                       0, NULL, NULL);  // events
clFinish(queue);

// 7. Look at the results via synchronous buffer map.
cl_uint *ptr;
ptr = (cl_uint *)clEnqueueMapBuffer(queue, buffer,
                                    CL_TRUE, CL_MAP_READ,
                                    0, NWITEMS *
                                    sizeof(cl_uint),
                                    0, NULL, NULL, NULL);
```

Explanation (4)

- Set the kernel arguments to `buffer`
- We launch the kernel, enqueue the 1-dimensional index space starting at 0
- We specify the index space has `NWITEMS` elements and not to subdivide the program into work-groups
- There is an event interface, which will do not use

- We copy the results back, the call is blocking `CL_TRUE`
- This means we don't need an explicit `clFinish()` call
- We specify we went to read the results back into our buffer

Simple Example (5)

```
    int i;
    for(i=0; i < NWITEMS; i++)
        printf("%d %d\n", i, ptr[i]);
    return 0;
}
```

- The program simply prints 0 0, 1 1, ..., 511 511
- Note, there is no clean up, or any error handling for any of the OpenCL functions

C++ Bindings

- If we use the C++ bindings, we'll get automatic resource release and exceptions
    - C++ likes to use the RAII style
(resource allocation is initialization)
- Change the header to `CL/cl.hpp` and define `__CL_ENABLE_EXCEPTIONS`
- We would also like to store our kernel in a file instead of a string
- The C API is not so nice to work with

Vector Addition Kernel

- Let's write a kernel that adds two vectors and stores the result
- This kernel will go in the file `vector_add_kernel.cl`

```
__kernel void vector_add(__global const int *A,
                         __global const int *B,
                         __global int *C) {

    // Get the index of the current element to be processed
    int i = get_global_id(0);

    // Do the operation
    C[i] = A[i] + B[i];
}
```

- Other qualifiers: `local`, `constant` and `private`

Vector Addition (1)

```
#define __CL_ENABLE_EXCEPTIONS

#include <CL/cl.hpp>

#include <iostream>
#include <fstream>
#include <string>
#include <utility>
#include <vector>

int main() {
    // Create the two input vectors
    const int LIST_SIZE = 1000;
    int *A = new int[LIST_SIZE];
    int *B = new int[LIST_SIZE];
    for(int i = 0; i < LIST_SIZE; i++) {
        A[i] = i;
        B[i] = LIST_SIZE - i;
    }
```

Vector Addition (2)

```
    try {
        // Get available platforms
        std::vector<cl::Platform> platforms;
        cl::Platform::get(&platforms);

        // Select the default platform and create a context
        // using this platform and the GPU
        cl_context_properties cps[3] = {
            CL_CONTEXT_PLATFORM,
            (cl_context_properties)(platforms[0])(),
            0
        };
        cl::Context context(CL_DEVICE_TYPE_GPU, cps);

        // Get a list of devices on this platform
        std::vector<cl::Device> devices =
            context.getInfo<CL_CONTEXT_DEVICES>();

        // Create a command queue and use the first device
        cl::CommandQueue queue = cl::CommandQueue(context,
            devices[0]);
```

Explanation (2)

- You can define `__NO_STD_VECTOR` and use `cl::vector` (same with strings)

- You can enable profiling by adding `CL_QUEUE_PROFILING_ENABLE` as the third argument to `queue`

Vector Addition (3)

```
    // Read source file
    std::ifstream sourceFile("vector_add_kernel.cl");
    std::string sourceCode(
        std::istreambuf_iterator<char>(sourceFile),
        (std::istreambuf_iterator<char>())
    );
    cl::Program::Sources source(
        1,
        std::make_pair(sourceCode.c_str(),
                       sourceCode.length()+1)
    );

    // Make program of the source code in the context
    cl::Program program = cl::Program(context, source);

    // Build program for these specific devices
    program.build(devices);

    // Make kernel
    cl::Kernel kernel(program, "vector_add");
```

Vector Addition (4)

```
    // Create memory buffers
    cl::Buffer bufferA = cl::Buffer(
        context,
        CL_MEM_READ_ONLY,
        LIST_SIZE * sizeof(int)
    );
    cl::Buffer bufferB = cl::Buffer(
        context,
        CL_MEM_READ_ONLY,
        LIST_SIZE * sizeof(int)
    );
    cl::Buffer bufferC = cl::Buffer(
        context,
        CL_MEM_WRITE_ONLY,
        LIST_SIZE * sizeof(int)
    );
```

Vector Addition (5)

```
// Copy lists A and B to the memory buffers
queue.enqueueWriteBuffer(
    bufferA,
    CL_TRUE,
    0,
    LIST_SIZE * sizeof(int),
    A
);
queue.enqueueWriteBuffer(
    bufferB,
    CL_TRUE,
    0,
    LIST_SIZE * sizeof(int),
    B
);

// Set arguments to kernel
kernel.setArg(0, bufferA);
kernel.setArg(1, bufferB);
kernel.setArg(2, bufferC);
```

Explanation (5)

`enqueue*Buffer` arguments:

- *buffer*
- `cl_ bool` *blocking_write*
- `::size_t` *offset*
- `::size_t` *size*
- `const void *` *ptr*

Vector Addition (6)

```
// Run the kernel on specific ND range
cl::NDRange global(LIST_SIZE);
cl::NDRange local(1);
queue.enqueueNDRangeKernel(
    kernel,
    cl::NullRange,
    global,
    local
);

// Read buffer C into a local list
int* C = new int[LIST_SIZE];
queue.enqueueReadBuffer(
     bufferC,
     CL_TRUE,
     0,
     LIST_SIZE * sizeof(int),
     C
);
```

Vector Addition (7)

```
        for(int i = 0; i < LIST_SIZE; i++) {
            std::cout << A[i] << " + " << B[i] << " = "
                     << C[i] << std::endl;
        }
    } catch(cl::Error error) {
        std::cout << error.what() << "(" << error.err()
                 << ")" << std::endl;
    }

    return 0;
}
```

- This program just prints all the additions (equaling 1000)

Other Improvements

- The host memory is still unreleased
  - In the same number of lines, we could use the C++11 `unique_ptr`, which would free the memory for us
- You can use a vector instead of an array, and use `&v[0]` instead of `<type>*`
  - Valid as long as the vector is not resized

Summary

- Went through real OpenCL examples
- Have the reference card for the AP
- C++ template for setting up OpenCL
- Aside: if you're serious about programming in C++, check out **Effective C++** by Scott Meyers (slightly dated with C++11, but it still has some good stuff)

Other Notes

- Assignment 2 grading is delayed again
- Assignment 4 will be posted on the 21st and due on the 28th (will be similar to last years, but shorter)