Lecture 4

OVERVIEW OF OPENCL APIS

Host programs can be “ugly”

- OpenCL’s goal is extreme portability, so it exposes ***everything***
  - (i.e. it is quite verbose!).
- But most of the host code is the same from one application to the next – the re-use makes the verbosity a non-issue.
- You can package common API combinations into functions or even C++ or Python classes to make the reuse more convenient.

The C++ Interface

- Khronos has defined a common C++ header file containing a high level interface to OpenCL, cl.hpp
- This interface is dramatically easier to work with[1]
- Key features:
  - Uses common defaults for the platform and command-queue, saving the programmer from extra coding for the most common use cases
  - Simplifies the basic API by bundling key parameters with the objects rather than requiring verbose and repetitive argument lists
  - Ability to "call" a kernel from the host, like a regular function
  - Error checking can be performed with C++ exceptions

[1] especially for C++ programmers…

C++ Interface: setting up the host program

- Enable OpenCL API Exceptions. Do this before including the header file

```
#define __CL_ENABLE_EXCEPTIONS
```

- Include key header files … both standard and custom

```
#include <CL/cl.hpp>     // Khronos C++ Wrapper API
#include <cstdio>        // For C style
#include <iostream>      // For C++ style IO
#include <vector>        // For C++ vector types
```

For information about C++, see the appendix:
"C++ for C programmers".

C++ interface: The vadd host program

```
std::vector<float>
  h_a(N), h_b(N), h_c(N);
// initialize host vectors…

cl::Buffer d_a, d_b, d_c;

cl::Context  context(
   CL_DEVICE_TYPE_DEFAULT);

cl::CommandQueue
   queue(context);

cl::Program  program(
  context,
  loadprogram("vadd.cl"),
  true);

// Create the kernel functor
cl::make_kernel<cl::Buffer,
 cl::Buffer, cl::Buffer, int>
 vadd(program, "vadd");
```

```
// Create buffers
// True indicates CL_MEM_READ_ONLY
// False indicates CL_MEM_READ_WRITE

d_a = cl::Buffer(context,
        h_a.begin(), h_a.end(), true);

d_b = cl::Buffer(context,
        h_b.begin(), h_b.end(), true);

d_c = cl::Buffer(context,
        CL_MEM_READ_WRITE,
        sizeof(float) * LENGTH);

// Enqueue the kernel
vadd(cl::EnqueueArgs(
                queue,
                cl::NDRange(count)),
        d_a, d_b, d_c, count);

cl::copy(queue,
      d_c, h_c.begin(), h_c.end());
```

The C++ Buffer Constructor

- This is the API definition:
  - Buffer(startIterator, endIterator, bool readOnly, bool useHostPtr)
- The readOnly boolean specifies whether the memory is CL_MEM_READ_ONLY (true) or CL_MEM_READ_WRITE (false)
  - You must specify a true or false here
- The useHostPtr boolean is default false
  - Therefore the array defined by the iterators is implicitly copied into device memory
  - If you specify true:
    - The memory specified by the iterators must be contiguous
    - The context uses the pointer to the host memory, which becomes device accessible - this is the same as CL_MEM_USE_HOST_PTR
    - The array is not copied to device memory
- We can also specify a context to use as the first argument in this API call

The C++ Buffer Constructor

- When using the buffer constructor which uses C++ vector iterators, remember:
  - This is a blocking call
  - The constructor will enqueue a copy to the first Device in the context (when useHostPtr == false)
  - The OpenCL runtime will automatically ensure the buffer is copied across to the actual device you enqueue a kernel on later if you enqueue the kernel on a different device within this context

The Python Interface

- A python library by Andreas Klockner from University of Illinois at Urbana-Champaign
- This interface is dramatically easier to work with[1]
- Key features:
  - Helper functions to choose platform/device at runtime
  - getInfo() methods are class attributes – no need to call the method itself
  - Call a kernel as a method
  - Multi-line strings – no need to escape new lines!

[1] not just for python programmers…

Setting up the host program

- Import the pyopencl library

```
import pyopencl as cl
```

- Import numpy to use arrays etc.

```
import numpy
```

- Some of the examples use a helper library to print out some information

```
import deviceinfo
```

```python
N = 1024
# create context, queue and program
context = cl.create_some_context()
queue = cl.CommandQueue(context)
kernelsource = open('vadd.cl').read()
program = cl.Program(context, kernelsource).build()

# create host arrays
h_a = numpy.random.rand(N).astype(float32)
h_b = numpy.random.rand(N).astype(float32)
h_c = numpy.empty(N).astype(float32)

# create device buffers
mf = cl.mem_flags
d_a = cl.Buffer(context, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=h_a)
d_b = cl.Buffer(context, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=h_b)
d_c = cl.Buffer(context, mf.WRITE_ONLY, h_c.nbytes)

# run kernel
vadd = program.vadd
vadd.set_scalar_arg_dtypes([None, None, None, numpy.uint32])
vadd(queue, h_a.shape, None, d_a, d_b, d_c, N)

# return results
cl.enqueue_copy(queue, h_c, d_c)
```

Exercise 3: Running the Vadd kernel (C++ / Python)

- Goal:
  - To learn the C++and/or Python interface to OpenCL's API
- Procedure:
  - Examine the provided program. They will run a simple kernel to add two vectors together
  - Look at the host code and identify the API calls in the host code. Note how some of the API calls in OpenCL map onto C++/Python constructs
  - Compare the original C with the C++/Python versions
  - Look at the simplicity of the common API calls
- Expected output:
  - A message verifying that the vector addition completed successfully

Exercise 4: Chaining vector add kernels (C++ / Python)

- Goal:
  - To verify that you understand manipulating kernel invocations and buffers in OpenCL
- Procedure:
  - Start with a VADD program in C++ or Python
  - Add additional buffer objects and assign them to vectors defined on the host (see the provided vadd programs for examples of how to do this)
  - Chain vadds … e.g. C=A+B; D=C+E; F=D+G.
  - Read back the final result and verify that it is correct
  - Compare the complexity of your host code to C
- Expected output:
  - A message to standard output verifying that the chain of vector additions produced the correct result

(Sample solution is for C = A + B; D = C + E; F = D + G; return F)