ECE459: Programming for Performance Winter 2013

Lecture 21 — March 28, 2013

Patrick Lam *version 1*

Complete OpenCL Example

```
//
// Copyright (c) 2010 Advanced Micro Devices, Inc. All rights reserved.
//
// A minimalist OpenCL program.
#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 );

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

Walk-through. Let's look at all of the code in the example and explain the terms. 1) First, we request an OpenCL *platform*. Platforms, also known as hosts, contain 2) OpenCL *compute devices*, which may in turn contain multiple compute units. Note that we could also request a CPU device in step 2, without changing the rest of the code.

Next, in step 3, we request an OpenCL *context* (representing all OpenCL state) and create a *command-queue*. We will request that OpenCL do work by telling it to run a kernel in the queue.

In step 4, we create an OpenCL *program*. This is a confusing term; an OpenCL program is what runs on the compute unit, and includes kernels, functions, and declarations. Your application can contain more than one OpenCL program. In this case, we create a program from the C string `source`, which contains the kernel `memset`. OpenCL can also create programs from binaries, which may be in an intermediate representation, or already compiled for a particular device. We get a pointer to the kernel in this step, as the return value from `clCreateKernel`.

There's one more step before launching the kernel; in step 5, we create a *data buffer*, which enables communication between devices. Recall that OpenCL requires explicit communication, which we'll see later. Since this example doesn't have input, we don't need to put anything into the buffer initially.

Finally, we can launch the kernel in step 6. In this case, we don't specify anything about workgroups, but enqueue the entire 1-dimensional index space, starting at (0). We also state that the index space has `NWITEMS` elements, and not to subdivide the problem into work-items. The last three parameters are about events. We call `clFinish()` to wait for the command-queue to empty.

Finally, in step 7, we copy the results back from the shared buffer using `clEnqueueMapBuffer`. This copy is blocking (first `CL_TRUE` argument), so we don't need an explicit `clFinish()` call. We also indicate the details of the command we'd like to run: in particular, a read of `NWITEMS` from the buffer.

You might also want to consider cleaning up the objects you've allocated; I haven't shown that here. The code also doesn't contain any error-handling.

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'd also like to store our kernel in a file instead of a string. The C API is not so nice to work with; the C++ API is nicer.

More Complicated Kernel

I've omitted the C code. it's pretty similar to what we saw before, but it uses workgroups, customized to the number of compute units on the device. Here is a more interesting kernel.

```
#pragma OPENCL EXTENSION cl_khr_local_int32_extended_atomics : enable
#pragma OPENCL EXTENSION cl_khr_global_int32_extended_atomics : enable

// 9. The source buffer is accessed as 4-vectors.
__kernel void minp( __global uint4 *src,
                    __global uint  *gmin,
                    __local  uint  *lmin,
                    __global uint  *dbg,
                    size_t          nitems,
                    uint            dev )
{
  // 10. Set up __global memory access pattern.
  uint count  = ( nitems / 4 ) / get_global_size(0);
  uint idx    = (dev == 0) ? get_global_id(0) * count
                           : get_global_id(0);
  uint stride = (dev == 0) ? 1 : get_global_size(0);
  uint pmin   = (uint) -1;

  // 11. First, compute private min, for this work-item.
  for( int n=0; n < count; n++, idx += stride )
  {
    pmin = min( pmin, src[idx].x );
    pmin = min( pmin, src[idx].y );
    pmin = min( pmin, src[idx].z );
    pmin = min( pmin, src[idx].w );
  }

  // 12. Reduce min values inside work-group.
  if( get_local_id(0) == 0 )
    lmin[0] = (uint) -1;
  barrier( CLK_LOCAL_MEM_FENCE );

  (void) atom_min( lmin, pmin );
  barrier( CLK_LOCAL_MEM_FENCE );

  // Write out to __global.
  if( get_local_id(0) == 0 )
    gmin[ get_group_id(0) ] = lmin[0];

  // Dump some debug information.
  if( get_global_id(0) == 0 )
     { dbg[0] = get_num_groups(0); dbg[1] = get_global_size(0);
       dbg[2] = count; dbg[3] = stride; }
}

// 13. Reduce work-group min values from __global to __global.
__kernel void reduce( __global uint4 *src, __global uint *gmin )
{
   (void) atom_min( gmin, gmin[get_global_id(0)] ) ;
}
```

Let's discuss the notable features of this code, which finds the minimum value from an array of 32-bit ints. (OpenCL ints are always 32 bits). Steps 1 through 8 are in the C code, which I've omitted; see the AMD guide for the code. At 9), we can investigate the signature of the `minp` kernel. The use of `uint4`, or 4-int vectors, enables SSE instructions on CPUs and helps out GPUs as well. We'll access the constituent `int`s of `src` using the `.x`, `.y`, `.z` and `.w` fields. This kernel also writes to an array of global minima, `gmin`, and an array of local minima (inside the workgroup), `lmin`.

In step 10, we figure out where our point in the index space, as reported by `get_global_id()`, is located in the `src` index, as well as the stride, which is 1 for CPUs and $7 \times 64 \times c$, where c is the number of work units, which was rounded up using the following heuristic:

```
  cl_uint ws = 64;
  global_work_size = compute_units * 7 * ws; // 7 wavefronts per SIMD
  while ( (num_src_items / 4) % global_work_size != 0 )
    global_work_size += ws;

  local_work_size = ws;
```

The core of the kernel occurs in step 11, where the `for`-loop computes the local minimum of the array elements in the work-item. In this stage, we are reading from the `__global` array `src`, and writing to the private memory `pmin`. This takes almost all of the bandwidth.

Then, in stage 12, thread 0 of the workgroup initializes the workgroup-local `lmin` value, and each thread atomically compares (using the extended atomic requested using the pragma) its `pmin` to the local `lmin` value. We have local memory fences here to make sure that threads stay in synch. This code is not going to consume much memory bandwidth, since there aren't many threads per work-group, and there's only local communication.

Finally, thread 0 of the workgroup writes the local minimum of the workgroup to the global array `gmin`. In step 13, a second kernel traverses the `gmin` array and finds the smallest minimum.

Summary. We've now seen the basics of GPU programming. The key idea is to define a kernel and find a suitable index space. Then you execute the kernel over the index space and collect results. The main difficulty is in formulating your problem in such a way that you can parallelize it, and then in splitting it into workgroups.