ECE459: Programming for Performance Winter 2016

Lecture 22 — GPU Programming (OpenCL)

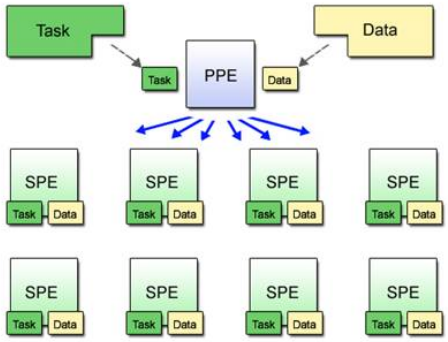
Patrick Lam

GPUs: Heterogeneous Programming

The next two lectures will be about programming for heterogeneous architectures. In particular, we'll talk about GPU programming, as seen in OpenCL (i.e. Open Computing Language). The general idea is to leverage vector programming; vendors use the term SIMT (Single Instruction Multiple Thread) to describe this kind of programming. We've talked about the existence of SIMD instructions previously, but now we'll talk about leveraging SIMT more consciously. We are again in the domain of embarassingly parallel problems.

Resources. I've used the NVIDIA *OpenCL Programming Guide for the CUDA Architecture*, version 3.1[1] as well as the *AMD Accelerated Parallel Processing OpenCL Programming Guide*, January 2011[2].

Cell, CUDA, and OpenCL. Other examples of heterogeneous programming include programming for the PlayStation 3 Cell[3] architecture and CUDA. (Note that the PS4 returns to a regular CPU/GPU configuration; however, it uses AMD hardware which combines the CPU and GPU on one chip.) The Cell includes a PowerPC core as well as 8 SIMD coprocessors:



(from the Linux Cell documentation)

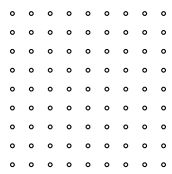
CUDA (Compute Unified Device Architecture) is NVIDIA's architecture for processing on GPUs. "C for CUDA" predates OpenCL; NVIDIA still makes CUDA tools available, and they may be faster than OpenCL on NVIDIA hardware. On recent devices, you can use (most) C++ features in CUDA code, which you can't do in OpenCL code.

OpenCL is a cross-vendor standard for GPU programming, and you can run OpenCL programs on NVIDIA and AMD chips, as well as on CPUs. We will talk about OpenCL for the next 2 classes.

Programming Model. The programming model for all of these architectures is similar: write the code for the massively parallel computation (kernel) separately from the main code, transfer the data to the GPU coprocessor (or execute it on the CPU), wait, then transfer the results back.

[1] `http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_OpenCL_ProgrammingGuide.pdf`
[2] `http://developer.amd.com/gpu/amdappsdk/assets/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf`
[3] `http://www.kernel.org/pub/linux/kernel/people/geoff/cell/ps3-linux-docs/ps3-linux-docs-latest/CellProgrammingPrimer.html`

OpenCL includes both task parallelism and data parallelism, as we've discussed earlier in this course. *Data parallelism* is central to OpenCL; in OpenCL's view, you are evaluating a function, or *kernel*, at a set of points, like so:



Another name for the set of points is the *index space*. Each of the points corresponds to a *work-item*.

OpenCL also supports *task parallelism*: it can run different kernels in parallel. Such kernels may have a one-point index space. The documentation doesn't say much about task parallelism.

More on work-items. The work-item is the fundamental unit of work in OpenCL. These work-items live on an n-dimensional grid (ND-Range); we've seen a 2-dimensional grid above. You may choose to divide the ND-Range into smaller work-groups, or the system can divide the range for you. OpenCL spawns a thread for each work item, with a unique thread ID. The system runs each work-group on a set of cores; NVIDIA calls that set a *warp*, while ATI calls it a *wavefront*. The scheduler assigns work-items to the warps/wavefronts until there are no more work items left.

Shared memory. OpenCL makes lots of different types of memory available to you:

- private memory: available to a single work-item;
- local memory (aka "shared memory"): shared between work-items belonging to the same work-group; like a user-managed cache;
- global memory: shared between all work-items as well as the host;
- constant memory: resides on the GPU, and cached. Does not change.

There is also host memory, which generally contains the application's data.

An example kernel. Let's continue by looking at a sample kernel, first written traditionally and then written as an OpenCL kernel[4].

```
void traditional_mul(int n, const float *a, const float *b, float *c) {
  int i;
  for (i = 0; i < n; i++) c[i] = a[i] * b[i];
}
```

The same code looks like this as a kernel:

```
kernel void opencl_mul(global const float *a, global const float *b, global float *c) {
  int id = get_global_id(0);  // dimension 0
  c[id] = a[id] * b[id];
}
```

You can write kernels in a variant of C. OpenCL takes away some features, like function pointers, recursion, variable-length arrays, bit fields, and standard headers; and adds work-items, workgroups, vectors, synchronization, and declarations of memory type. OpenCL also provides a library for kernels to use.

[4] www.khronos.org/developers/library/overview/opencl_overview.pdf

Branches. OpenCL implements a SIMT architecture. What this means is that the computation for each work-item can branch arbitrarily. The hardware will execute all branches that any thread in a warp executed (which can be slow).

```
kernel void contains_branch(global float *a, global float *b) {
    int id = get_global_id(0);
    if (cond) {
        x[id] += 5.0;
    } else {
        y[id] += 5.0;
    }
}
```

In the above example, the `if` statement will cause each thread to execute both branches of the `if`, keeping only the result of the appropriate branch.

Similarly, executing a loop will cause the workgroup to wait for the maximum number of iterations of the loop in any work-item.

```
kernel void contains_loop(global float *a,
                          global float *b) {
    int id = get_global_id(0);

    for (i = 0; i < id; i++) {
        b[i] += a[i];
    }
}
```

If you're setting up workgroups, though, you can arrange for all of the work-items in a workgroup to execute the same branches.

Synchronization. You might define workgroups because you can only put barriers and memory fences between work items in the same workgroup. Different workgroups execute independently.

OpenCL also supports all of the notions that we've talked about before: memory fences (read and write), barriers, and the volatile keyword. The barrier (`barrier()`) ensures that all of the threads in the workgroup all reach the barrier before they continue. Recall that the fence ensures that no load or store instructions (depending on the type of fence) migrate to the other side of the fence.

Complete OpenCL Example

```
// Note by PL: don't use this example as a template; it uses the C bindings!
// Instead, use the C++ bindings as in the other example.
// source: pages 1-9 through 1-11, http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf

//
// 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. Use it! As an example, you'll find the `vector_add` example in the slides and in the code repo.

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, also from the same source.

```
#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.