ECE459: Programming for Performance Winter 2013

Lecture 20 — March 26, 2013

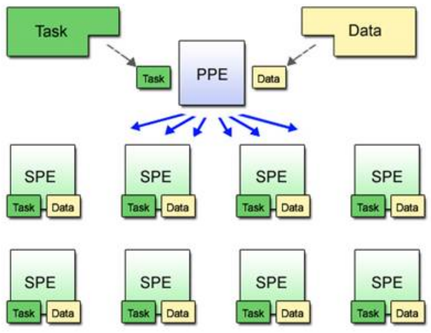
Patrick Lam *version 1*

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. 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

[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

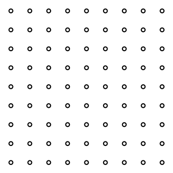
[3]http://www.kernel.org/pub/linux/kernel/people/geoff/cell/ps3-linux-docs/ps3-linux-docs-latest/CellProgrammingPrimer.html

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.

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.

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).

For instance, including an `if` statement in a kernel will cause each thread to execute both branches of the `if`, keeping only the result of the appropriate branch; executing a loop will cause the workgroup to wait for the maximum number of iterations of the loop in any work-item.

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. One reason that you might define workgroups is that 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.

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