ECE 459: Programming for Performance
Assignment 3

Patrick Lam

April 9, 2017 (Due: March 24, 2017)

In this assignment, you will write OpenCL code to implement parts of an N-body simulation. The first part consists of converting some sequential C code into OpenCL, while the second part requires you to improve the algorithm by making approximations for points that are far away.

We'll be starting with the code from GPU Gems 3[1]. I also found an OpenCL N-body simulation code on the Internet[2]. Also, you can read Andrew Cooke's notes; they are similar to my OpenCL lecture, but they may make more sense to you[3].

Using OpenCL on ece459-1 should be fairly straightforward as long as you have the right compiler options. The Makefile I've provided should work for you. You are also free to use OpenCL on your own computer, if you choose. There are C++ as well as C bindings for OpenCL but you should use **only the C++ bindings**. The C binding has a nasty habit of crashing the machine (and/or rendering OpenCL unusable). If you use the C bindings, you will get 0 marks for that part of the assignment.

Getting started. Fork the provided git repository:

```
ssh git@ecgit.uwaterloo.ca fork ece459/1171/a3 ece459/1171/USERNAME/a3
```

and then clone the provided files.

To submit, push C++ files containing OpenCL versions (naive and optimized) of the N-body code, along with a brief report (about half a page).

Part 1: Brute-force approach (50 points)

The easiest way to do an N-body simulation is to compute the effects of all points on each other. I've posted some sequential code (based on the GPU Gems code) to calculate the forces for one time-step. Your first task is to convert this code to OpenCL. We will evaluate the correctness and efficiency of your conversion. OpenCL can assign workgroups automatically.

Part 2: Far-field approximations (50 points)

Algorithmic improvements are important. In this part, you will speed up the force calculation by crudely estimating the forces exerted by faraway points. The idea is to divide the points into a number of bins. (Real codes would

[1] http://http.developer.nvidia.com/GPUGems3/gpugems3_ch31.html
[2] http://www.browndeertechnology.com/docs/BDT_OpenCL_Tutorial_NBody-rev3.html
[3] http://www.acooke.org/cute/APractical0.html

use quadtrees or octrees to store, and thus find, the closest points[4]). You will then compute the center of mass for each bin and add the force exerted by the center of mass for faraway bins to the force exerted by individual particles for nearby particles.

To be explicitly clear, each of the three parts below should be done in an OpenCL kernel and not in CPU code. Please note also that you may not modify the sample results or the comparison python script; if the order of points your program outputs is different it means the algorithm has changed in some way and that is an error.

Computing centers-of-masses for bins (20 points). Instead of finding the k nearest points, we will divide space into a fixed number of bins and compute the center of mass of each of these bins. For nearby bins, we do the $O(n^2)$ calculation; for further bins, we compute forces for each bin. Conveniently, for this assignment I've divided space into $[0, 1000]^3$, so we can take bins which are cubes of length 100. This gives 1000 bins.

I recommend that you create a 3-dimensional array `cm` of `float4`s[5] to store centers-of-mass. The `x`, `y` and `z` components contain the average position of the center of mass of a bin, while the `w` component stores the total mass. Compute all of the masses in parallel: create one thread per bin, and add a point's position if it belongs to the bin, e.g.

```
int xbin, ybin, zbin; // initialize with bin coordinates
int b;
for (i = 0; i < POINTS; i++) {
    if (pts[i] in bin coordinates) {
        cm[b].x += pts[i].x; // y, z too
        cm[b].w += 1.0f;
    }
}
cm[b].x /= cm[b].w; // etc
```

Note that this parallelizes with the number of bins.

Bin Contents (20 points). For the next step, you'll also need to keep track of the points in each bin. Fortunately, you've collected the number of points in each bin, so you can allocate the appropriate amount of memory to store the points in a two-dimensional array `binPts`. In a second phase, iterate over all bins again, this time putting coordinates into the proper element of `binPts`.

Computing Forces (10 points). The payoff from all these calculations is to save time while calculating forces. Let's arbitrarily say that we'll compute exact forces for the points in the same bin and the directly-adjacent bins in each direction (think of a Rubik's Cube; that makes 27 bins in all, with 6 bins sharing a square, 12 bins sharing an edge, and 8 bins sharing a point with the center bin). If there is no adjacent bin, then there are no points in that bin.

Using the data that you've computed so far, write OpenCL code to estimate forces for each point. This has two parts. In the first part, compute forces directly for the points in the 27 adjacent bins. In the second part, sum the forces from the centers of mass.

There is a caveat: it's easier to add forces from all centers of mass, whether nearby or far away. I recommend that you add forces from centers of mass, and then subtract away the forces that you're double-counting:

```
// add negative forces to not double-count adjacent bins
negBin.x = 2*myPosition.x-globalCM[bin].x;
negBin.y = 2*myPosition.y-globalCM[bin].y;
negBin.z = 2*myPosition.z-globalCM[bin].z;
negBin.w = globalCM[bin].w;
bodyBodyInteraction(myPosition, negBin, pacc);
```

[4]For a readable summary of the Barnes-Hut algorithm, see `http://arborjs.org/docs/barnes-hut`.
[5]When I did the assignment, a `float` versus `float4` mismatch took me a long time to debug.

Finally, compare the performance of part 1 and part 2.

What to hand in. Commit and push your code. For part 2, write about your design choices and results (about half a page), and hand that in along with your OpenCL code. Please use LaTeX for this; a template is provided for you in the base code that you fork.

Timings

Here's some timings from the past.

With 500*64 points.

- OpenCL, no approximations (1 kernel): 0.182s
- OpenCL, with approximations (3 kernels): 0.168s

With 5000*64 points.

- OpenCL, no approximations (1 kernel): 6.131s
- OpenCL, with approximations (3 kernels): 3.506s