ECE459: Programming for Performance Winter 2020

Lecture 17 — Early Termination, Reduced-Resource Computation

Patrick Lam & Jeff Zarnett *2019-10-31*

The N-Body Problem and Reduced-Resource Computation

A common physics problem that programmers are asked to simulate is the N-Body problem: you have some number of bodies (N!) and they interact via gravitational forces. The program needs to compute the movements of the bodies over time. This is a typical example of a program that is well suited to OpenCL: you can compute the forces on each body n from all other bodies in parallel. This was even at one time an assignment in this course, although now there are too many good solutions on the internet so it was replaced. Ah well.

Once you've made the conversion to OpenCL and let's assume you've optimized it well so far. What can you do here if you want to speed it up even more? You could look for optimizations that trade off accuracy for performance. As you might imagine, using `float` instead of `double` can save half the space which should make things quite a bit faster. But you want more...

In the specific case of the N-body simulation, domain knowledge would enable you to skimp out on some unnecessary computation: points that are far away contribute only very small forces. So you can estimate them (crudely). The idea is to divide the points into a number of "bins" which are cubes representing a locale of some sort. Then, compute the centre of mass for each bin. When calculating the forces on a given point, add the force exerted by the centre of mass for faraway bins to the force exerted by individual particles for nearby particles.

A more concrete explanation with an example: suppose the space is divided into $[0, 1000]^3$, so we can take bins which are cubes of length 100. This gives 1000 bins. If you want to increase the accuracy, increase the number of bins. If you want to increase the speed, decrease the number of bins.

The program should have a 3-dimensional array `cm` of `float4`s to store centres-of-mass. The `x`, `y` and `z` components contain the average position of the centres 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.

For the next step, the program needs 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`.

The payoff from all these calculations is to save time while calculating forces. In this example, 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 centre bin). If there is no adjacent bin (i.e., this is an edge), just act as if there are no points in the place where the nonexistent bin would be.

Necessarily, writing the program like this is going to mean more than one kernel. Each discrete step (compute centre of mass, put coordinates into bins, calculate forces) will be different kernels. This does mean there is

overhead for each kernel, meaning the total amount of overhead goes up. Is it worth it? Let's see the data from a reference solution [PL] for the previous assignment:

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

Early phase termination

In [RHMS10], Martin Rinard summarizes two of his novel ideas for automatic or semiautomatic optimizations which trade accuracy for performance: early phase termination [Rin07] and loop perforation [HMS+09]. Both of these ideas are applicable to code we've learned about in this class.

We've talked about barriers quite a bit. Recall that the idea is that no thread may proceed past a barrier until all of the threads reach the barrier. Waiting for other threads causes delays. Killing slow threads obviously speeds up the program. Well, that's easy.

"Oh no, that's going to change the meaning of the program!"

Let's consider some arguments about when it may be acceptable to just kill (discard) tasks. Since we're not completely crazy, we can develop a statistical model of the program behaviour, and make sure that the tasks we kill don't introduce unacceptable distortions. Then when we run the program, we get an output and a confidence interval.

Two Examples. When might this work? Monte Carlo simulations are a good candidate; you're already picking points randomly. Raytracers can work as well. Both of these examples could spawn a lot of threads and wait for all threads to complete. In either case, you can compensate for missing data points, assuming that they look like the ones that you did compute. If you have a function where some graph is being computed, you can probably guess that a missing point is somewhere in between the two adjacent points.

The same is true for graphics, of course: if rendering a particular pixel did not go well for some reason, you can just average the adjacent ones and probably people would not notice the difference. Not bad!

Also recall that, in scientific computations, you're entering points that were measured (with some error) and that you're computing using machine numbers (also with some error). Computers are only providing simulations, not the ground truth; the question is whether the simulation is good enough.

Loop perforation

You can also apply the same idea to sequential programs. Instead of discarding tasks, the idea here is to discard loop iterations. Here's a simple example: instead of the loop,

```
for (i = 0; i < n; i++) sum += numbers[i];
```

simply write,

```
for (i = 0; i < n; i += 2) sum += numbers[i];
```

and multiply the end result by a factor of 2. This only works if the inputs are appropriately distributed, but it does give a factor 2 speedup.

Example domains. In [RHMS10], we can read that loop perforation works for evaluating forces on water molecules (in particular, summing numbers); Monte-Carlo simulation for swaption pricing; and video encoding. In that example, changing loop increments from 4 to 8 gives a speedup of 1.67, a signal to noise ratio decrease of 0.87%, and a bitrate increase of 18.47%, producing visually indistinguishable results. The computation looks like this:

```
min = DBL_MAX;
index = 0;
for (i = 0; i < m; i++) {
  sum = 0;
  for (j = 0; j < n; j++) sum += numbers[i][j];
  if (min < sum) {
    min = sum;
    index = i;
  }
}
```

The optimization changes the loop increments and then compensates.

References

[HMS+09] Henry Hoffmann, Sasa Misailovic, Stelios Sidiroglou, Anant Agarwal, and Martin Rinard. Using code perforation to improve performance, reduce energy consumption, and respond to failures. Technical Report MIT-CSAIL-TR-2009-042, MIT CSAIL, Cambridge, MA, September 2009.

[RHMS10] Martin Rinard, Henry Hoffmann, Sasa Misailovic, and Stelios Sidiroglou. Patterns and statistical analysis for understanding reduced resource computing. In *Proceedings of Onward! 2010*, pages 806–821, Reno/Tahoe, NV, USA, October 2010. ACM. URL: `http://doi.acm.org/10.1145/1932682.1869525`.

[Rin07] Martin Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *Proceedings of OOPSLA 2007*, pages 369–386, Montreal, Quebec, Canada, October 2007.