ECE 459: Programming for Performance
Assignment 3

Pavel Shering

March 11, 2016

OpenCL Implementation

This assignment deals with the implementation of the Coulombs Law simulation from assignment 2 using OpenCL instead of OpenMP parallelization. The problem fits perfectly with OpenCL as computation on each particle is done independently of any other. The GPU kernels is used to perform calculations of y1, z1 and the max error. My design of the program and kernels is very similar to that of Assignment 2, having two kernels to perform all the calculations. The first propagates position and the second calculates the average and checks that the position is within the maximum tolerance of the error. The program structure is shown in the code snippet bellow.

```
// get kernels
// create buffers
// set up kernel args

while loop {
  // enqueueWriteBuffer for h
  // run kernel1 to get y0
  // run kernel1 to get y1
  // ren kernel2 to get z1 and check error
  // enqueueReadBuffer to check if error calc passed
}
```

For optimization, I let the compiler decide of the maximum local work group sizes, and thus get a major improvement as shown in the benchmarks in **Table 1**. The Table shows 10 runs on `ecetesla0` using only the **C++ OpenCL bindings**, the average of the ten runs, as well as the speed ups compared to the Sequential version, including the comparison of OpenMP and OpenCL performances. The OpenCL version proves to be much more efficient on large amount of particles, as seen in the 4000 particle test. After optimization the OpenCL version achieves a 13x speed up, which is marginally better than the OpenMP, and fails miserably for the small inputs of 50 and 500 due to large set up times for kernels.

Table 1: Speed benchmark results comparing OpenCL implementation of Coulombs Law simulation to sequential and OpenMP versions

	50 particles				500 particles				4000 particles			
Run #	seq	omp	ocl	ocl opt	seq	omp	ocl	ocl opt	seq	omp	ocl	ocl opt
1	0.027	0.196	2.288	0.664	0.344	0.33	2.24	0.99	22.887	3.565	6.206	1.733
2	0.023	0.122	2.309	0.67	0.284	0.289	2.682	0.958	22.727	3.896	6.146	1.721
3	0.024	0.111	2.284	0.668	0.296	0.267	1.777	0.935	22.547	3.501	6.155	1.72
4	0.028	0.046	2.265	0.675	0.286	0.255	2.302	0.882	22.9	3.444	6.179	1.636
5	0.025	0.104	2.259	0.677	0.283	0.262	2.337	1.131	23.4	3.427	6.16	1.636
6	0.024	0.118	2.199	0.672	0.295	0.25	2.296	0.946	23.22	4.06	6.226	1.635
7	0.023	0.111	2.195	0.683	0.289	0.257	3.032	0.895	22.833	3.684	6.172	1.591
8	0.029	0.188	2.217	0.679	0.293	0.256	1.468	0.928	25.488	4.409	6.28	1.809
9	0.024	0.107	2.168	0.676	0.289	0.257	2.263	0.904	22.553	3.555	6.693	1.748
10	0.024	0.074	2.001	0.655	0.297	0.252	2.348	1.456	22.952	3.436	6.102	1.845
Average	0.0251	0.1177	2.2185	0.6719	0.2956	0.2675	2.2745	1.0025	23.1507	3.6977	6.2319	1.7074
Speed Up		**0.213**	**0.011**	**0.037**		**1.105**	**0.130**	**0.295**		**6.261**	**3.715**	**13.559**

The difficulties that occurred during this assignment is debugging the math that already worked in Assignment 2 and realizing that for some reason the passed in `float` variables had to be casted again to achieve the correct results. Further optimizations are also possible, the main one would be to perform reduction for the error calculation and obtain the sum of `max(0, magnitude(z1 - y1) - e)` and check if the sum is not zero then run huns method again with a smaller time step.