ECE 459: Programming for Performance
Assignment 3

Jeff Zarnett & Stephen Li

February 12, 2020 (Due: March 10, 2020)

In this assignment, you will convert CPU code to use the GPU (via OpenCL). The two parts are your jwtcracker code that finds the secrets from assignment 2, and a Coulomb's Law simulation that can solve the sort of problem you'd see on a first year electromagnetism exam (but scaled up). Both problems are great candidates for OpenCL. The jwtcracker can try different combinations in parallel; the Coulomb's law simulation works because computation of the force on each point p at a given time is independent of the computation of any other point at that same time.

OpenCL is covered in the lecture notes. Or, you can read Andrew Cooke's notes. Although they are similar to the OpenCL lecture, they may make more sense to you[1].

Using OpenCL on any of the `ecetesla` systems should be fairly straightforward as long as you have the right compiler options. The Makefile I've provided should work for you. 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 and we will be slightly displeased**.

Getting started. Check `git.uwaterloo.ca` for your repository which should be automatically created for you and populated with the starter files. The starter files are the Makefile and some OpenCL boilerplate that will be necessary, alongside a sequential implementation of the Coulomb's Law problem and some test cases.

Once you clone your repository, you can copy in your source files from the serial version of the jwtcracker. You'll need to modify them extensively, but it's probably much better to start with those than to start from scratch. But it's up to you! At the end of this document is a small amount of guidance on how to convert OpenMP code to OpenCL.

What to submit. Push C++ files as well as your kernel file(s), containing the OpenCL version of your code, along with a brief report (max 1 page). Use `hyperfine` to run your code to produce your numbers in your report.

Part 1: Crack it to me (30 marks)

The first part of this assignment is to convert your sequential jwtcracker code so that it uses OpenCL because it is the ideal way to do it. The starter code files give you some OpenCL boilerplate as well as some OpenCL implementations of the helper functions you need.

For a refresher on the JWT cracker and how it is supposed to work, see the assignment 2 description; repeating it here only introduces the possibility of error.

Your code should produce output that's exactly the same as the sequential (and OpenMP) versions of the jwtcracker. The goal of the assignment is to correctly convert the CPU code into GPU (OpenCL) code. Correctness counts the most, but efficiency is also a good thing (you don't want it to be massively slower).

[1] `http://www.acooke.org/cute/APractical0.html`

Part 2: Coulomb's Law Problem (60 marks)

The provided program does a simulation of Coulomb's Law: there are proton and electron particles (that have the standard masses and charges). The protons are kept fixed in place via mechanical forces, but the electrons will move. Electrons move according to classical physics: they are attracted to protons and repelled by other electrons. The program will perform just one step of the simulation (in all cases), which makes it possible to verify you did it correctly by hand if you need. The code is an implementation of Heun's method from ECE 204A (which you may remember if you took that course or an equivalent).

The program takes parameters:

1. h – initial size of simulation step (a measure of time)
2. e – epsilon, the amount of error allowed
3. An input file of initial positions (comma separated value file).

The input file is in csv format, first column is whether it is a proton (indicated by `p`) or electron (indicated by `e`), second is x coordinate, third is y coordinate, fourth is z coordinate. The input values are floating point numbers (of `float` type) and they have a precision of 6 digits and are written using scientific notation such as `3.14159e-05`.

The program produces as output the new positions of the electrons and protons (the protons do not move). The output file format should be the same as the input file format, and the order of each of the particles must be the same as the order of the input file.

Some physics facts you may need to know to get this done:

- Coulomb's law calculates the force of attraction as $\frac{k(q_1 q_2)}{r^2}$, where k is Coulomb's constant, q the charge, and r the distance between the two points.
- The value for k is Coulomb's constant $8.99 \times 10^9 N m^2 C^{-2}$.
- The charge of an electron is the same as the charge of a proton which is $1.60217662 \times 10^{-19}$ Coulombs. Electrons have negative charge and protons have positive charge.
- The mass of an electron is $9.10938356 \times 10^{-31}$ kg.

The simulation algorithm is:

1. The vector y_0 contains the initial positions of the electrons and protons.
2. Use Coulomb's Law to calculate a vector k_0 which contains the net force on each particle (protons have net force of zero).
3. After you calculate the force vector k_0, approximate the new positions vector $y_1 = y_0 + \Delta d$.

   To calculate new positions, remember $F = m \times a$; acceleration will be considered constant in the range we're talking about (h represents a very small unit of time, or it will after the value of h has been divided multiple times after we find the error has been too large).

   With the acceleration then we compute velocity v for any point as $h \times a$; with the velocity in hand, the position is updated as simply $h \times v$ (again worth noting that the times are very small here so we don't have wild swings in v).

   Or to break it down in math notation: $F = ma \rightarrow \frac{F}{m} = a \rightarrow \frac{hF}{m} = v \rightarrow \frac{h^2 F}{m} = \Delta d$

4. Calculate a second force vector k_1 which contains the sum of the forces on each point at their new position y_1, again having zero forces on the protons.
5. Using the two force vectors, compute a new positions vector $z_1 = y_0 + \Delta d$ using $F = \frac{(k_0 + k_1)}{2}$ (the average of the two forces). This produces a second, more accurate position vector.
6. If $||z_1 - y_1|| > e$ for any particle (i.e., the error at any one position is larger than the tolerance), then the simulation is too coarse, and we need to go back to step 3, this time with h divided by 2.
7. Once you have found h such that the displacement for all particles is smaller than the tolerance, you are ready to print the output to `stdout`. The output should be in the same format as the input file, and the positions z_1 are the final results that should be put in the output.

You should also try to achieve the maximum speedup you can while preserving behaviour. The usage of the compiled output is: `./protons h e inputfile`. So a sample call might be: `./protons 1 1.000e-05 example.in` ; this behaviour needs to be preserved for your solution to be tested.

Note that the order of operations on IEEE floating point numbers can affect the final values. Since not everyone will write their code the same way as our sample solution, we will allow differences up to ± 0.00005 (`5e-5`). If your output values differ more than this, it is very likely you made a mistake or typo in your calculations.

We have provided some sample input and output files. The sample output files are created with parameters $h = 0.001$ and $e = 0.00001$. You should, however, do some tests with other parameters as well. In addition, it is highly recommended, but not required, for you to generate larger test cases for yourself. The test cases provided are not a complete test suite.

The goal of the assignment is to correctly convert the CPU code into GPU (OpenCL) code. Correctness counts the most, but efficiency is also a good thing (you don't want it to be massively slower). You need to do at least the following operations in GPU (OpenCL kernel). You may **not** do these things in CPU code because it defeats the purpose of the assignment:

1. the y_1 computation
2. the z_1 computation
3. checking for error larger than the maximum allowed.

Hint for item 3: You might think of this as either a reduction, or as finding the max value...

Part 3: Report (10 marks)

In the report, write about your design choices and results (max one page). Collect your results using `hyperfine` with a warmup run. Make sure you tell us what server you used. Please use LaTeX for this; a template is provided for you in the base code that you cloned.

Grading

Part 1: 15 marks for correctness of conversion; 15 marks for efficiency;

Part 2: 30 marks for correctness of conversion; 30 for efficiency.

Part 3: 10 marks for report.

Efficiency in this case means (1) clean code conversion, (2) leverages the GPU well, and (3) doesn't do unnecessary work. Your code will be evaluated on the ECE GPU servers (the `ecetesla` machines) so make sure it works as expected there.

Guidance: Converting Your Code to OpenCL

Step one would be to remove all OpenMP annotations. Your code should not contain OpenMP directives (the goal is, after all, to do work in OpenCL).

The OpenCL kernel is written in a language that is very C-like. This language does have some significant limitations compared to C. This makes it difficult to use structures like the `Vec3` that appear in the starter code. But rest assured that we didn't provide something that would force a massive rewrite later. OpenCL gives you the types `float3` and `float4` (and other variants like 2, 8, 16...), which can take the place of `Vec3`.

A `float4`, for example, is a grouping of 4 single-precision floating point numbers. It is equivalent to a `struct` composed of 4 `float` variables. It has properties (like `.x`) to access the components, and there are already-defined arithmetic and comparison operations on them. So you should be able to drop these types into your program, replacing `Vec3`, without great difficulty. The links below should be helpful in getting familiar with the vector types and how to use them. They are both from the same article; the whole article might be worth a read but you should definitely read at least these two pages to understand the types and how the operations on them work:

`http://www.informit.com/articles/article.aspx?p=1732873&seqNum=3`

`http://www.informit.com/articles/article.aspx?p=1732873&seqNum=10`

Beware, of course, of mismatches such as writing `float` when you mean `float4` or similar. Beware also that if you use the `w` property of `float4` for things like determining if something is a proton or electron then you might not get the desired behaviour for the default math operations unless you use them carefully. Note that the types have slightly different names depending on where you use them: in the OpenCL kernel it's called `float4` but in your C++ code it is `cl_float4`, for example.

Once you have converted to using the OpenCL vector types, it's probably a good idea to test with some test cases and make sure that nothing was broken by this. Then make a commit to save your work and you'll then go on to the next part, which is handing work over to the GPU. You do set-up and configuration in the host (CPU, C++) code, before giving the work to the GPU and collecting the results back to the host code.

Your solution should probably consist of three kernels (but could be slightly more or less, depending).