Parallel Galaxy Simulation
Alexander DeMello, Alexander Lazar
March 4, 2016
CPE 419

1. Implementation Overview. A few paragraphs describing the parallelization details of your implementations, including the specific items mentioned above.
2. Results. Compute frames-rendered-per-second.

The parallelization process began with running vTune with the base implementation of the galaxy simulation to confirm where the bulk of the work was happening in the program. We found that the stepParticles() function was where 99.9% of the runtime work was occurring so parallelizing other portions of the code would not matter much at all. In order to test any speedup that we would have in the future we implemented a frames rendered counter that displays in the command line window. This was later changed to a FPS counter to better display real time performance. First to test base levels of speed up we implemented some basic openMP pragmas. The results of this first test are displayed below.

| # of particles | Frames rendered in 1m Base version | Frames rendered in 1m OpenMP |
|---|---|---|
| collide | 1518 | 4541 |
| chainsaw | 269 | 1318 |
| three passing | 25 | 175 |

Figure 1: Baseline speed up test

This base version showed roughly 3 times speedup on the smallest of test and up to 7 times speedup on the largest test. The next step was to offload the computation to the MIC. As it turned out this could not be done successfully without major refactoring of the code given to us by Ian. Ian's implementation was designed to work and did not have speed in mind at all. This meant the use of the Vector3d type and its native functions from C++. This type does not transfer to the MIC at all because it is not bitwise copyable due the pointer heavy structure that it relies on.

The next step was to refactor the data types heavily in order to facilitate cache locality and functionality of the MIC offload. This step proved to be incredibly difficult due to the implementation of the galaxy simulation given to us. As mentioned previously this implementation relies heavily on C++ types and functions native to these types. This made refactoring the program to improve cache locality extremely unsuccessful as it required the changing of almost the whole of the code base without the help of the original implementer. In

the end we decided it was best to revert to the last working version of the program which was the original speedup test displayed in the Figure 1 above.
For CUDA:

| Galaxy | Frames rendered in 1m<br>Base version | Frames rendered in 1m<br>CUDA |
|---|---|---|
| collide | 1518 | 14760 |
| chainsaw | 269 | 4980 |
| three passing | 25 | 816 |

The massive 10-32x speedups can be seen as the larger thread count of the GPU allows it to work with more particles. In order to get the particle data to work well with the GPU, we had to refactor all physics data of each particle (location, velocity, mass) to a newly created array of structs. With this, we were able to get compiling to work much more easily and didn't have to copy as much data back - we only need location copied back, not the entire particle data. We also realized that there was no need to copy location etc onto the GPU each time, but only at the start as all the calculations happened on the GPU and the data only needed to be taken off the GPU for rendering, no modifying took place outside the GPU. Data was put into structs consisting of x, y and z elements, with one struct for velocity, one for location and one for forces. This decreased the amount of data we had to copy off, as we only need to copy off location. Unfortunately, due to the many pairings any particle could have when calculating forces shared memory was not viable.

Our biggest failure of optimization was branch divergence. We put in a lot of time getting the program to even compile, and then even more to make it work, and not enough time into making sure threads in a block all had something to do. This isn't as big an issue for moving the particles once forces are calculated, as there will always be overflow when thread count doesn't divide evenly into particle count. However, for calculating the forces we simply tell each thread a pair of particles to consider and then don't let it run if that pairing isn't valid. We did a little optimization so that entire blocks would fall into this, but many blocks will have only a small portion of the threads running while the others sit idle. We could have gotten even better performance if we had done this optimization better.