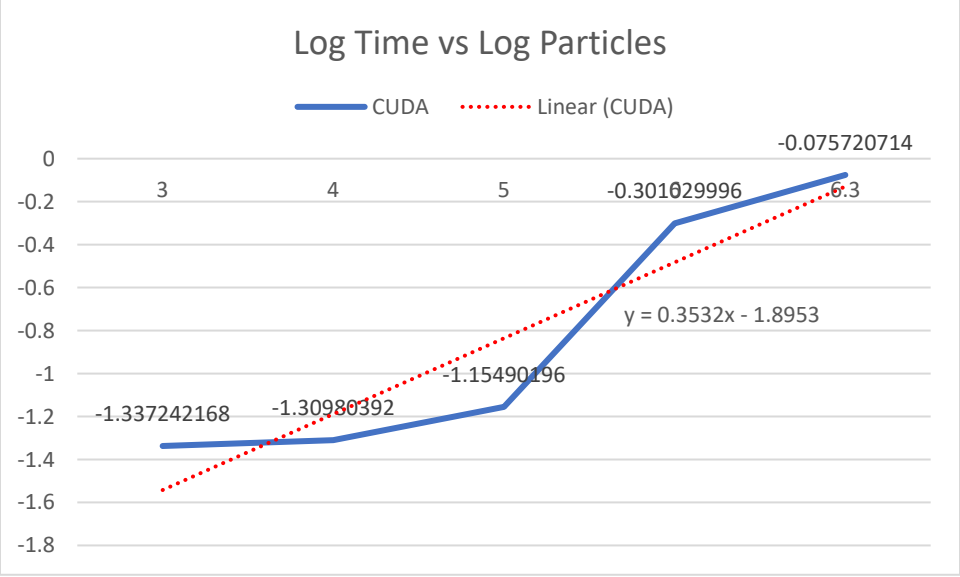
CS 5220 HW3 Report

Name: Santiago Lai, Cornell id: zl345, NERSC username: Zhengnan

In this report, I will discuss each of the optimizations I attempted in GPU implementation, from data structures and synchronization I used to some design choices in my code. I will also benchmark my GPU implementation against the starter code, comparing it to the performance with OpenMP, in addition to a breakdown of runtime into computation time/ synchronization time and communication time.

0. Plot in log-log scale for my GPU codes:

The plot for the runtime by my GPU code is shown below. In a log-log scale, it is seen that the implementation on GPU is highly efficient. However, it is also worth noticing that there is a runtime jump from 1e5 particles to 1e6 particles.



1. Data structures in GPU implementation

In my GPU implementation, I followed the instructions in the handout to perform particle binning. I used a one-dimensional array to count the number of particles. I then created another one-dimensional array to store the (exclusive) prefix sum of the previous result. The importance of the prefix sum array is that for the i-th bin, I will store the indices of particles in that bin into the entries between prefix[i-1] and prefix[i]. Finally, I assigned particles into the one-dimensional bin according to the prefix sum array I calculated previously.

2. Design choices in GPU implementation

One of the most important parts in my code implementation is the size of bins. I hope to choose a bin size so that I'll only need to visit the 8 nearby bins when applying force, thus reducing the runtime complexity from quadratic to linear. Considering edge cases where a particle is at the corner of a bin, the bin size should be at least the cut-off size. However, I later noticed that there is a tradeoff between large and small bin sizes. If the bin size is small, there will be less particles in each bin, meaning that we'll consider less interactions, but in the meanwhile, there will be more bins that require reinitialization at each step, thus increasing the runtime. By experimentally setting the bin size to different multiples of cut-off distance, I found that the runtime is the best when the size is set to 2.2 times the cut off distance.

The second significant change I made is to use the symmetric of the forces. By applying a two-way function, I was able to consider the interactions of only four nearby bins, thus noticeably reducing the runtime.

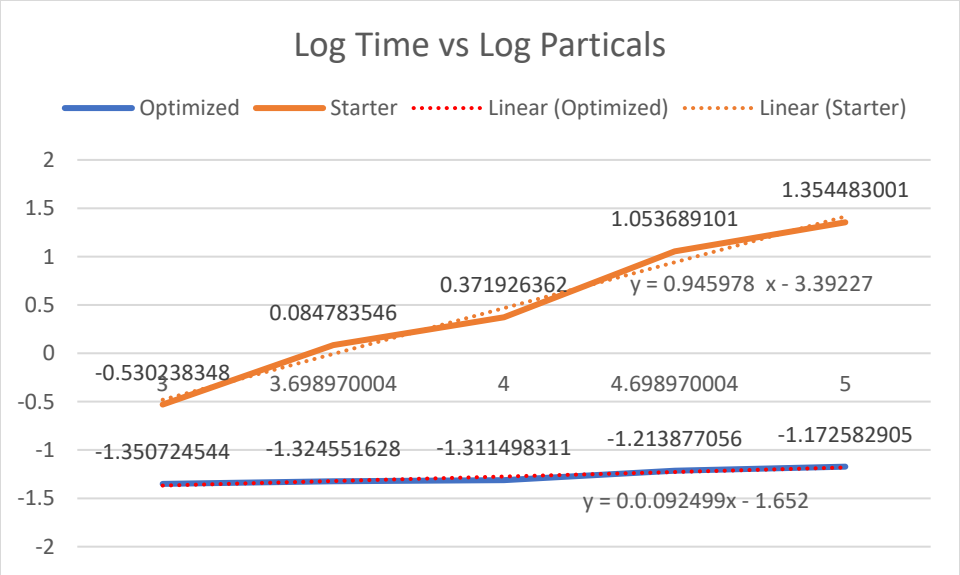
Finally, I noticed that the acceleration of a particle is actually not necessary, as I can directly change the velocity of a particle. Therefore, I removed all references to the accelerations and modified them to equivalent expressions. This process reduces a lot of memory accesses, and improve the performance greatly.

3. Synchronization in GPU implementation

The most important synchronization I used in this project are AtomicAdd and AtomicSub. They were especially useful when a few threads are accessing the same array at a time. I used these operations when counting the number of particles in each bin, inserting particles into bins, applying the two-way force, and in re-binning process.

4. Comparison with starter code

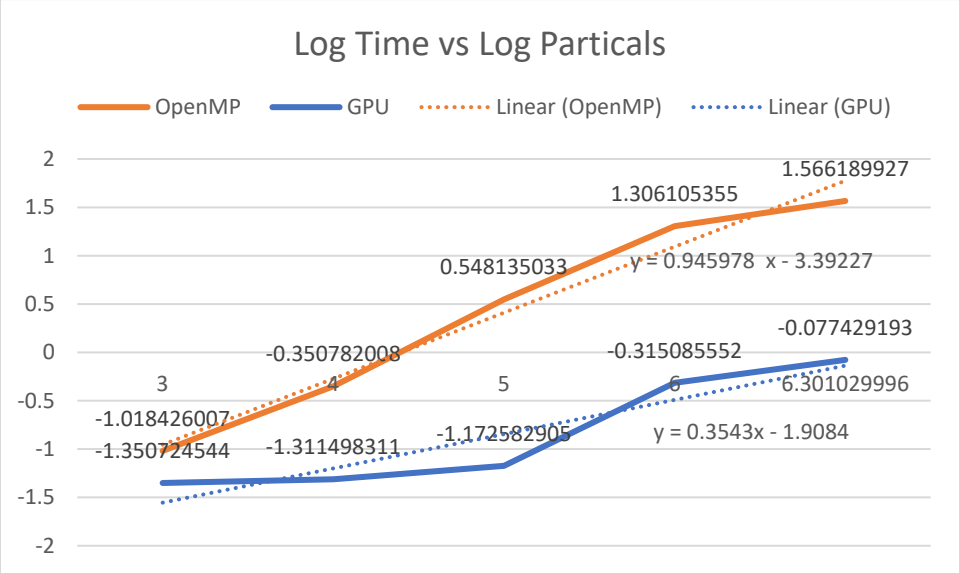
Please see the following plot for comparison between my code and the starter code. I discarded the runtime at 1e6 (and beyond) because it takes too long for the starter code to finish. From the data points collected, we see that the starter code scales linearly, though there is a huge runtime increase if we use 1e6 particles. On the other hand, my optimized code demonstrated highly parallel behavior, as the slope is roughly 0.1, which is significantly lower than 1.



It is hard to do better because there's a significant portion of my code that is non-parallelizable, i.e., putting the particles into bins. It implies that my parallel efficiency will approach zero as the number of processors increases.

5. Comparison with OpenMP code

I compared the performance using my OpenMP code with 64 threads.



We see that our OpenMP code achieves a linear behavior, but the performance on GPU outperforms the OpenMP code for every number of particles I selected. The performance difference is large especially when the number of particles increases to 1e6 or 2e6.

6. Runtime decomposition

I used the tool nvsight to profile the runtime details of my program. The computation time involves time spent on GPU kernel, the communication time involves CUDA memcpy Host-to-Device, and the synchronization time involves cudaDeviceSynchronization and cudaSteamSynchronization. We see that the major components of the runtime are computation time and synchronization time. However, as the number of particles increases, the communication time increases rapidly.