CS5220: HW3 Optimizing Particle Simulation GPU

Name: Andrew Cheng, Cornell ID: 5128767,

Cornell netid: akc55, Perlmutter username: andyhero

CS 5220 Spring 2024

1 Introduction

The core of this assignment is optimizing particle simulations. For this assignment, we have to optimize the total simulation time of particles. In the naive version of the execution, each step would scale quadratically $O(n^2)$ because you would need to calculate the impact of each particle on any of its possible neighbors. The simulation moves over 1000 steps, and we need to minimize the time. Due to the quadratic nature of the problem, there are many ways to optimize the algorithm's execution. These optimizations include binning and many cache optimizations for optimal execution. The change of this assignment compared to HW2 is the introduction of GPU computations rather than CPU computations. Even within the baseline design of the GPU, we can already see a magnitude of difference between the original serial CPU implementation and a naive GPU implementation. The original serial implementation had a 40-second runtime on 10000 particles, but the GPU can achieve a runtime of just 4 seconds.

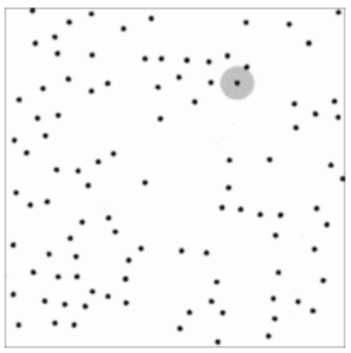

Figure 1: Particle Simulation

2 Personal Performance

For this assignment, the main source of computation is the A100 GPU. For 1 million particles, I achieved a time of around 0.5 - 0.6 seconds. For 2 million particles, I achieved a time of around 0.95 - 1.0 seconds. For computations less than 1 million, most of the particle simulations took less than 0.2 seconds. The large majority of my performance came from the discussion slides. Within the slides, I utilized the existing data structures to calculate the particle interactions optimally. Beyond the slides, I also incorporated Newton's third law back into the algorithm even though it would increase the number of atomic instructions on the over algorithm. However, this ended up being beneficial because the total number of computations that needed to be done by the algorithm was reduced by half.

3 Time Spent Within Algorithm

It is hard to quantify the runtime breakdown into computation time and synchronization time because the atomic increments are scattered throughout the algorithm. There is no set amount of time dedicated to synchronization and calculation. Furthermore, the GPU already has built-in synchronization when launching kernels. Each kernel launch will occur one step at a time, resulting in the synchronization required. Instead, I will be discussing the computation time between moving and calculating the particles within each step. For each step, the binning of the particles took the most time. This is because of the introduction of atomic increments and inserts into the particle idx array. These atomic components are needed to make sure each thread is synchronized on the index it needs to access for each thread. This step does result in some serialized computation because the counters have to be serially updated. Within the computation step, I initially had each thread operate and only update its own thread. This would result in no data sharing or synchronization required between threads. This resulted in a computation time of around 1.9 seconds. However, by doing repulsive forces going in both directions, it still ended up being faster, even though I now had to use atomic variables rather than a temporary accumulator. This showed that reducing the total number of computations by two and adding atomic instructions wasn't as expensive as doubling the total number of computations.

4 Data Structure Design Choices

Given the structure of GPUs, we need to use statically compiled sizes as often as possible. This resulted in me changing the way I iterate through the particles. Because bins can have a dynamic amount of particles, I needed to use a prefix sum to atomically increment where each particle exists in a static particle array. This maintains a static size array for all future simulation steps. While this does maintain a static size, we can't iterate through our bins or particles the same way on the GPU. For fast parallelization on the GPU, we now need to use particle-level parallelization. Given that bins are variable size, we can no longer parallelize our code based on bins. Thus, it is more beneficial to loop across particles. Now that we are doing parallel computations across particles, we want to reduce the amount of accesses on particles. This normally would be reduced if we did bin component parallelization, but all particles need to be contained within global memory. My first choice was to create a temporary acceleration accumulator within each thread. This reduced the total computation time required to 2 million particles down to 1.9 seconds. However, using Newton's third law using atomic addition still provided a better time of 1.8 seconds because the total number of computations has been reduced by half. Given the random distribution of particles, there is a very small chance of collisions resulting in serial accumulation between particles. Therefore, reducing the computation by half was still the most beneficial for GPU acceleration.

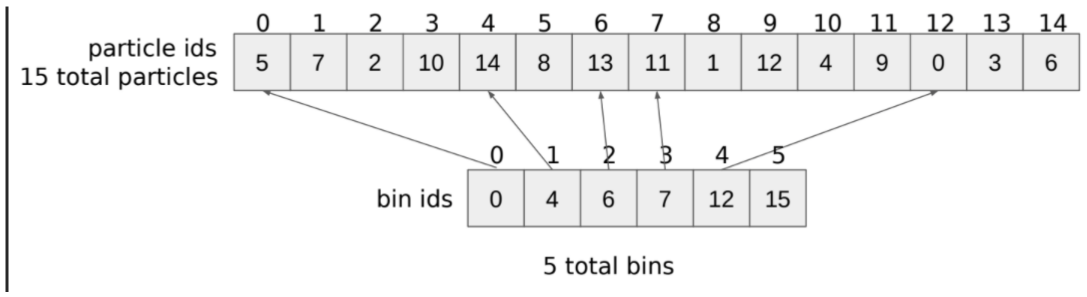

Figure 2: Particle Structure within GPU

5 Parallel Implementation

5.1 GPU Memory Hierarchy Utilization

One of the hardest challenges towards parallelization of binning is the memory structure of the algorithm. Unlike the stencil example seen in class, in which there is a compile-time constant that shows how to retrieve shared memory from global memory, I'm not sure of an efficient way of incorporating shared memory within the binning algorithm. This is mainly due to the fact that the number of particles per bin is dynamic between iterations, and the boundaries of the bins change between iterations. In the original OpenMP version, I computed threads based on the bins. However, this is no longer possible due to the hardware architecture of the GPU. As a result, I found it very difficult to utilize any component of shared memory between the thread blocks. This resulted in me only using global memory for the entire computation of the algorithm. Due to the memory bottleneck of the algorithm, I used many tactics to reduce the total number of trips to memory. I used a temporary accumulator within each individual thread to reduce the total number of atomic additions in half. This reduced the total serial computations that occurred on the GPU, leading to better results. Overall, the algorithm allowed for much better thread-level parallelism when it came to computation and struggled on the update component because the binning is dynamic.

5.2 Shared Memory Synchronization Bottlenecks

The only synchronization that was utilized was atomic accumulation. Given that each kernel is executed sequentially, there are implicit barriers between each step of the execution. For manual synchronization, I need to count the number of particles in each bin atomically as well as compute the force between each particle atomically. Given the random distribution of particles, the atomic updates were not that expensive compared to nonatomic additions. With this limited component of synchronization, I drastically reduced the number of memory operations required to compute the particle positions properly.

5.3 Parallel Specific Performance Optimizations

Since I have a single array of particles sorted by their bin IDs, we want to mark the beginning and end of every single bin using a prefix array. I used the thrust exclusive scan algorithm to optimize the prefix sum calculations for the beginning and end of each bin efficiently. After calculating the bins for each of the particles, we can now use the binning approach to compute forces. Since the bin sizes are dynamic, we use GPU resources to parallelize particles. Each thread executes the particle computation on each individual particle. For each of these particles, we locate the surrounding bins using the prefix array to calculate all surrounding particles against which we need to compute a force. After a single step, we will then execute the parallel all along particle-wide parallelism. The main difference between OpenMP and GPU programming is that I needed to be much more deliberate in memory sharing and the cost of memory sharing. Programming on a GPU requires much more use of static size arrays and not vectors. Thus, changing the iteration pattern to optimize only along particles was required, unlike CPU optimizations. The cache benefits from the CPU can't be realized from the GPU, so it was more beneficial to always iterate from particles.

6 Odd Behavior

One odd behavior of the calculations is the inconsistency of the computed time. Due to the fast nature of completing 2 million particles, any minor difference in computation time drastically decreases the LSC computed. I believe this variation in computation time is due to the randomness of global data synchronization as well as data communication. Throughout the computation process, there is randomness in the L2 cache on the GPUs, and it is important to take this randomness into account. Besides the variance of the computation time, there are not too many other odd behavior components that take place within the GPU algorithm.

7 Failed Optimizations

One of the assignment's key restraints was that we needed to maintain the original array of particles. This is mainly to guarantee correctness. However, one of the largest issues is that we now need two levels of indirection to find where the particle is actually sitting to update the particle. One optimization I tried to use was just creating a new struct that held the particle and the index, so we copied around the struct instead of just the index. This turned out to slow the program down because the particle structure was around 12 times larger than just holding the indexes of the particles. I believe this is the case because the copying does result in a lot of extra memory overhead that the double indirection doesn't cause. Furthermore, there is a good chance that the original particles are still in the GPU cache, leading to a much higher cache hit rather than just copying to a new segment each time. This failure of an optimization shows how memory is a very strong bottleneck in modern systems, and understanding the underlying memory is important to optimizing the algorithm.

The second failed optimization is the padding of the bins within the GPU. Currently, I use an if condition to check if the offset within the bin is a legal neighboring bin. In GPU computation, an if statement can be very punishing towards the total throughput of the algorithm. However, the introduction of these very rare if statements don't reduce the throughput drastically and actually increases the total time of execution due to the padding bins and the extra memory it uses when placing particles into bins. Thus, the work ends up being split between more threads, leading to slower results. The if conditions remained within the algorithm to reduce the total number of bins required to compute within the GPU.

8 Overall Scaling

It is hard to properly define strong and weak scaling within the context of GPUs because the blocks and number of threads within a kernel function don't properly represent how much faster the GPU is supposed to operate. Thus, I will be analyzing the overall scaling of the GPU computations. GPU computations seem very interesting before 100,000 particles, and the total time for computation stayed very stagnant. However, after 100,000, the LSC factors begin to reach 1.0 rather than smaller numbers of around 0.4 for less than 1 million particles. I believe this is the case for smaller particle numbers because the full GPU isn't utilized on these smaller numbers. Once the full grid is utilized, we see a 1.0 LSC factor of up to 50 million particles. This is mainly due to the fact that the kernel launches are limiting me to the GPU. I need to constantly wait for the GPU to finish one set of particles before moving to the next set of particles. Thus, we see a consistent LSC from 100,000 and onwards. The scaling factor can be seen in the figure below.

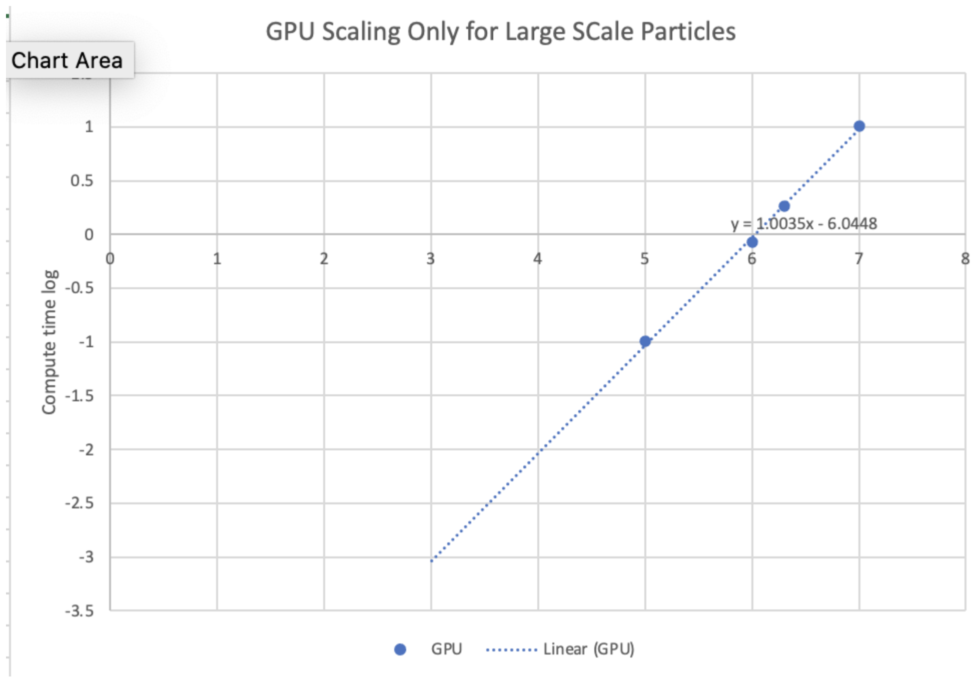

Figure 3: Large Particle GPU scaling

9 Comparison with OpenMP, Baseline GPU, and Serial

The first comparison is to compare the O(n) GPU version with the $O(n^2)$ GPU version. The main difference between the performance of these computations is the scaling factor. For the baseline version, we can see the scaling factor of the time complexity outshines any benefits the GPU might possess. While the GPU still maintains a magnitude of difference compared to the serial implementation and the $O(n^2)$ algorithm, we can see the scaling factor outpaces the computation.

The second comparison is between the OpenMP 64 thread binning algorithm and the GPU binning algorithm. While my OpenMP binning algorithm split the iteration with bins, the GPU algorithm split between the particles. This is mainly to utilize either the cache efficiency of the CPU with OpenMP or the thread-level parallelism in the GPU. The differences between the computations of these two can be seen in the slopes of the two algorithms. While the OpenMP version does create a $O(N)$ computation, there is a certain epsilon attached because there could be multiple particles in a certain bin leading to a higher exponent than linear. However, with the GPU version, the slope becomes much less apparent, and the slope of the GPU version is near Linear $O(n)$. This is mainly due to the parallel nature of the GPU and how it is able to mask the throughput and computation latency of the algorithm. As the number of particles scales, each particle is computed on a single thread on the GPU. Given the large scale of the GPU, each additional particle doesn't add nearly enough time to the overall computation compared to the binning iteration contained with the OpenMP computation. These differences in slope can be seen in the figure below.

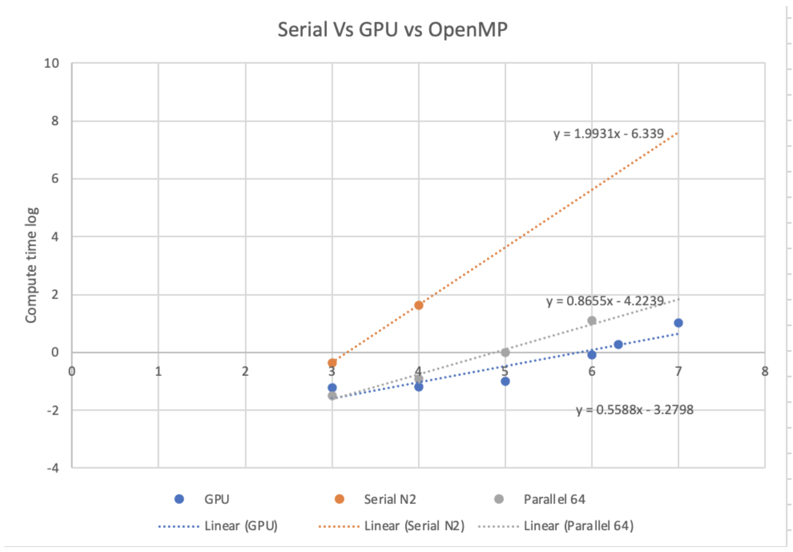

Figure 4: Computation Performance GPU vs OpenMP vs Serial

10 Conclusion

In the second part of the particle simulation optimizations on a GPU, I have adapted the way to calculate the particles using the GPU. For GPU calculation, static sizes are important, and parallelism is important to fully utilize each component of the GPU. Unlike openMP, the hardware on the Nvidia GPU is set in stone and can't use basic CPU instructions to execute the algorithm. As a result, one of the largest challenges of GPU programming is readjusting the algorithm to fit on the GPU. By programming specifically for the GPU, I reduced copies and increased the speed at which the algorithm was able to compute the particle simulation.