CS 267 HW 2.3

Group 28 - Alan Liang, Ashank Verma, Arnaud Fickinger
All parts of our optimization implementation and write-up were completed jointly between Alan, Arnaud, and Ashank

Data Structures Utilized

(Ashank, Arnaud, Alan)

We construct and utilize four integer arrays at each step of the simulation for this project:

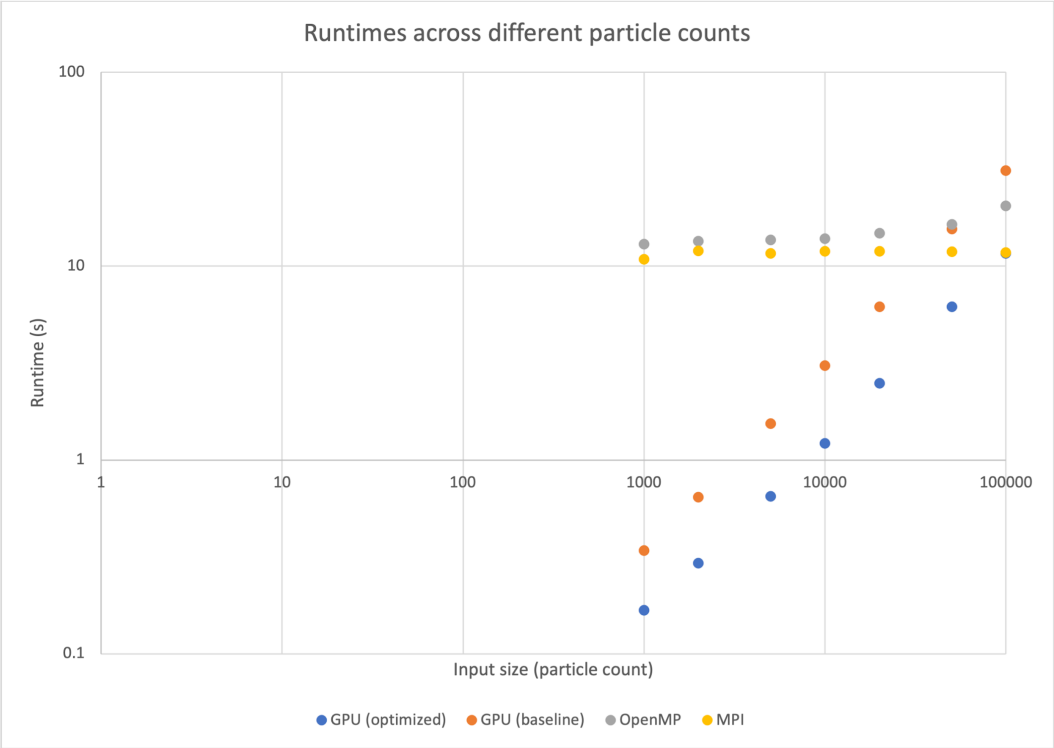
1. **Bin Counts Array**:
    - This array holds the number of particles that each bin contains. It is initially initialized with all 0's, with a total length of *num_bins*. After we are done populating this array, each slot in the array holds the number of particles in each bin at its corresponding index. This will allow us to compute a prefix sum which will help us order the array.

2. **Ordered Particles Array**
    - This array is the final data structure that we want to create using the other three arrays. It orders the particles by bin number, with bin numbers from [0 … *num_bins*] and each particle from the corresponding array grouped in the same region. The particles within each bin have no specific ordering. Each slot in this array will contain the index of the particle in the *original particles array*.

3. **Prefix Sum Array**
    - This array is generated using the **Bin Counts Array** described above. We run a prefix sum algorithm (by using thrust::exclusive_scan) to generate the starting indices of each bin in the **Ordered Particles Array** described above. The length of this array is *num_bins + 1*, where the last element is the number of particles, or *num_parts*. This is because the ending index in the **Ordered Particles Array** will be *num_parts*.

4. **Incremental Prefix Sum Array**
    - This array is initialized as a copy of the **Prefix Sum Array** described above. The purpose of this array is to guide in placing the particles in the **Ordered Particles Array**. Initially, each slot, or bin, is the starting index of that bin in the **Ordered Particles Array**. When we place a particle in the **Ordered Particles Array**, the value in the **Incremental Prefix Sum Array** corresponding to the bin increases by 1 atomically. That way, the next thread which is attempting to place the

particle will know which index in the **Ordered Particles Array** to put the particle in.



The plot above compares our GPU implementation's performance to that of the baseline GPU, as well as our OpenMP and MPI implementations (note that our MPI implementations may not be completely valid). We find that our optimized GPU implementation is generally the fastest across different particle sizes. In addition, we observe that both GPU implementations scale linearly as the particle count increases, without considerable initialization costs.

Synchronization

(Ashank, Arnaud, Alan)

Through our design choices, we were able to avoid using any type of synchronization. Every thread utilized the *atomicAdd()* operation in CUDA, avoiding the need to sync threads.

Design Choices

(Ashank, Arnaud, Alan)

As described above, we had one data structure for each of the major steps (as outlined in the lab) to get the simulation to work.

In our *initSimulation()* function, we simply utilized *cudaMalloc()* to initialize device level memory. At first, we were using a host array and a device array, which were supposed to be copies for each other. We were using the host array to initialize all of our initial values and then copy that to the device level array. However, we soon found that this was redundant and that we could do the initialization on the device array directly. In the *initSimulation()* function, we also initialized the number of bins we were going to use based on the cutoff as well.

Our *simulate_one_step()* consisted of 7 function calls in this order:

1. *update_bin_counts()* [PARALLELIZED]
    a. This parallelized function populates the **Bin Counts Array** described above. Each thread receives a particle, and using the parameters of that particle, we calculate the corresponding bin it belongs in. Then, we utilize *atomicAdd()* to increment the frequency of that bin counter in the **Bin Counts Array**, preventing any race conditions.
    b. Doing this computation is not that performance intensive since every thread receives its own particle and is able to do the computation in parallel.

2. *thrust::exclusive_scan()* [PARALLELIZED]
    a. We utilized the *thrust* library to generate the **Prefix Sum Array** from the **Bin Counts Array**. This function is parallelized by the *thrust* library.

3. *cudaMemcpy()*
    a. This function call allows us to create a copy of the **Prefix Sum Array**, which becomes our **Incremental Prefix Sum Array**.

4. *order_particles()* [PARALLELIZED]
    a. This function utilizes the **Incremental Prefix Sum Array** to create the **Ordered Particles Array** as described in the Data Structures section.
    b. Initially, we were not utilizing the return value from the *atomicAdd()* function when writing it. We were using *atomicAdd()* to increment the index in the **Incremental Prefix Sum Array**, and then subtracting one from that index to get the position of the particle. However, this resulted in a race condition. We were quickly able to find out that we should use the return value from the *atomicAdd()* function to place the particle in the correct position.

5. *compute_forces_gpu()* [PARALLELIZED]
    a. We modified this function from the original so that we could use our **Ordered Particles Array** for a performance boost.
        i. First, we wanted to find all the neighboring bins based on the particle that the thread had received. At first, we decided to use an array of length 9

(up to 9 potential neighbors), where we would populate each position in the array with a neighboring bin number. However, we quickly realized that this was inefficient, since we would have to *cudaMalloc()* a 9-length array for every particle that was being processed. Instead, we decided to hardcode the neighboring bin numbers in 9 variables within the function itself. Then, for every existing neighboring bin, we called a device helper function that we made called *compute_forces_bin()*. This function takes in the current particle and the neighboring bin number, calculating the forces of the particle with all particles in the neighboring bin. Using the **Prefix Sum Array**, we are able to calculate the start and end offset in the **Ordered Particles Array**, which allows us to look at every particle in the neighboring bin. Then, we simply call the given function *apply_force_gpu()* for every particle in the neighboring bin. These functions were completely parallelized, so this greatly decreased the amount of time it took to run the simulation.

6. *move_gpu()* [PARALLELIZED]
    a. This is the given function that takes in all the particles and moves them based on their updated forces.

7. *cudaMemset()*
    a. Finally, we reset the **Bin Counts Array** to an array of length *num_bins*, each with value 0. This prepares us for the next step.

Runtime Breakdown

(Ashank, Arnaud, Alan)
Since our implementation minimizes the communication between the CPU and GPU, most of the run time costs are likely from computation costs. In addition, since we did not leverage any explicit synchronization, synchronization costs were also kept at a minimum. This claim is supported by our runtime diagram above: across all particle sizes, the computation time scales linearly. This suggests low fixed costs and instead mainly costs from computation.