CS267 HW2-3 Write Up

Contributions:

- Write-Up: All
- Brandon: CUDA design and implementation & timing bottleneck breakdown.
- Kevin: Benchmarking performance.

Introduction:

In Homework 2.3, we focused on optimizing a two-dimensional particle simulation that models repulsive particle forces using CUDA on an NVIDIA GPU. The provided naive serial solution computed forces between every pair of particles, resulting in an O(n^2) complexity. Previous assignments introduced spatial binning to optimize the approach to O(n) complexity, first using shared-memory parallelism (OpenMP) and later extending it to distributed-memory parallelism (MPI). GPUs, however, present a distinct parallel computing model that leverages fast context switching and a massive number of lightweight threads executing SIMD operations, making them ideal for highly parallelized simulations. Our objective was to extend the binning approach to the GPU, ensuring efficient memory access and workload distribution while minimizing unnecessary computations.

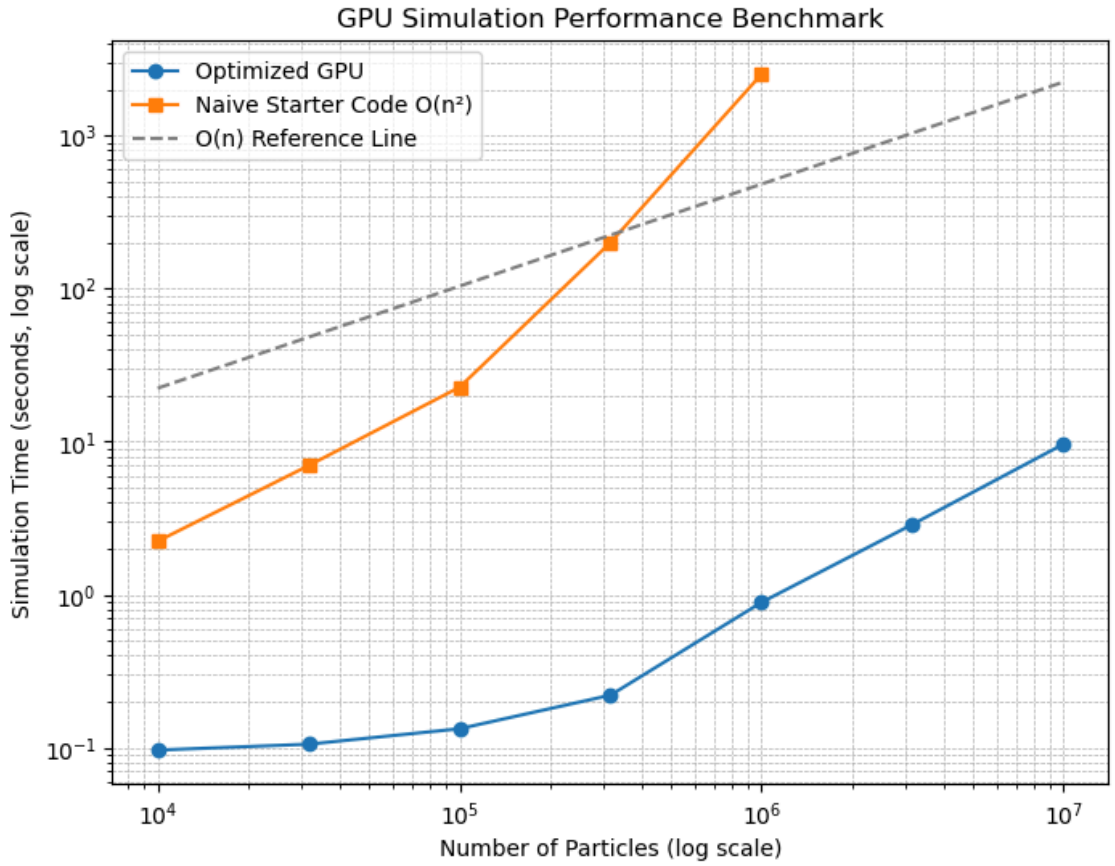
Data Structures and Optimization Strategies:

The core strategy behind our GPU optimization was bin-based spatial partitioning to avoid computing repulsive forces for all possible particle pairs. Instead of a brute-force approach, particles were assigned to spatial bins that ensured each particle only interacts with nearby particles. This reduced redundant force computations while improving memory locality, enabling coalesced memory accesses and efficient parallelism. Our implementation leveraged several key data structures to accomplish this:

1. d_bin_ids: An array storing the bin id for each particle index.
2. d_bin_counts: An array tracking the number of particles per bin.
3. d_particle_ids: An array storing particle indices in bin order, enabling efficient neighbor lookups.
4. d_bin_offsets: A prefix sum array indicating the starting index of each bin in the ordered list.

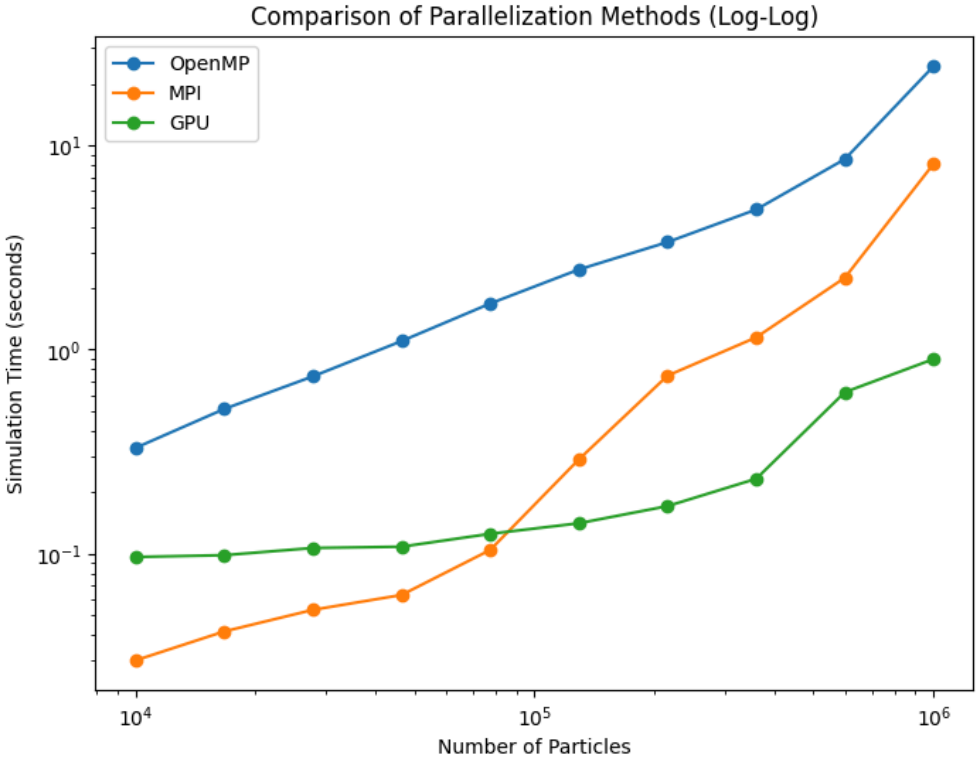
The bin assignment was performed in the assign_bins kernel, where each GPU thread processes a single particle, computing its bin index based on spatial coordinates. An atomic increment (atomicAdd) was then used to update d_bin_counts, ensuring thread-safe updates to shared memory. Once the bin counts were determined, we leveraged Thrust's exclusive_scan function to compute a prefix sum, which efficiently computed d_bin_offsets, the starting indices of bins in coalesced memory order. This allowed the scatter_particles kernel to rearrange particle_ids into contiguous memory using another atomic operation, ensuring fast and efficient bin traversal during force computation.

Our GPU-optimized implementation demonstrated a significant performance improvement over the baseline starter code, as highlighted in the log-log performance plot below. The naive approach, which computed all pairwise interactions, exhibited quadratic scaling (O(n^2)), leading to quadratic increases in runtime as the number of particles increased. In contrast, our optimized GPU implementation achieved near-linear scaling due to efficient binning and parallelization. The comparison plot of simulation times across different particle counts showed that while the starter code became intractable at large scales, the optimized CUDA implementation maintained stable growth patterns.



Key design choices played a crucial role in our GPU optimization. The binning strategy was essential for reducing redundant computations, and using atomic operations (atomicAdd) in force computations allowed safe parallel updates without requiring global memory locks. Additionally, prefix sum calculations using Thrust allowed efficient memory coalescing, ensuring minimized memory access latency. We initially explored alternative memory layouts, such as using linked lists for binning, but found that contiguous memory allocation provided superior performance due to reduced random accesses and improved cache utilization. Furthermore, the introduction of cudaDeviceSynchronize() at key computation stages ensures data consistency between kernel executions, preventing race conditions while maintaining high parallel throughput.

Comparison with Previous Implementations:



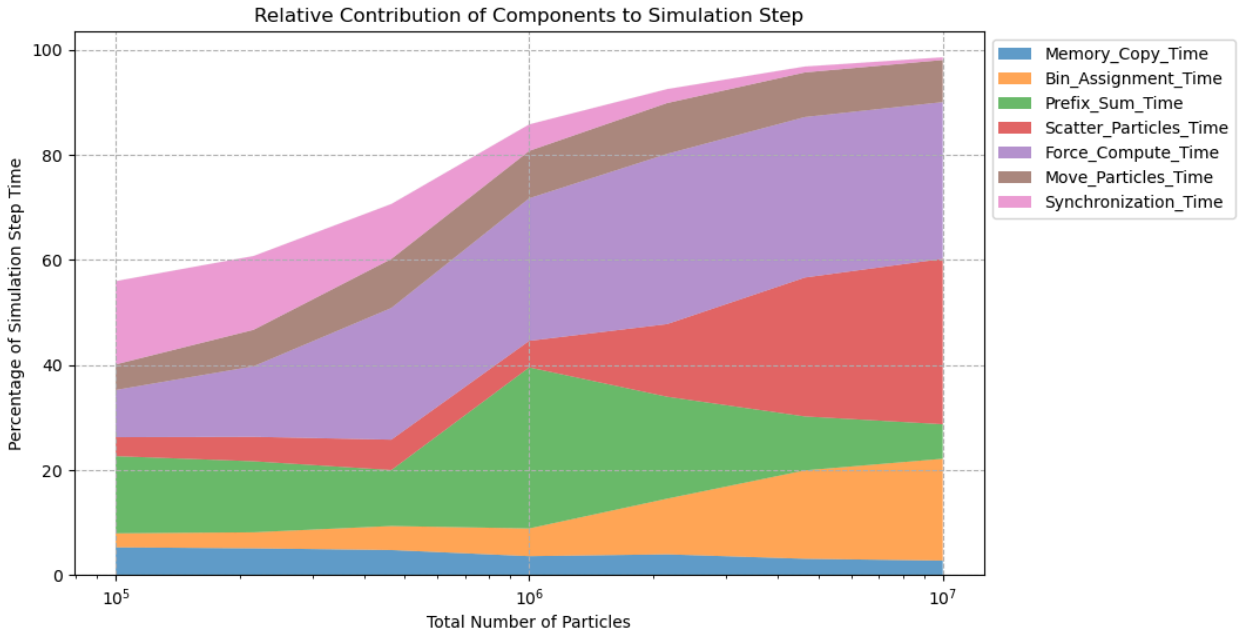
We compare the runtime performance of our GPU implementation with our previous parallelization methods using OpenMP from HW2-1 and MPI from HW2-2. For the plotted OpenMP times, we fix the number of threads to be 64. For MPI, we use two nodes and 64 tasks per node. For all three methods, we vary the number of particles from 10k to 1 million. For all problem sizes in this range, the GPU implementation outperforms the OpenMP implementation. Interestingly, for smaller problem sizes, the MPI implementation outperforms the GPU implementation up to just under 100k particles. For larger problem sizes after this threshold, the GPU implementation significantly outperforms the MPI implementation. This result aligns with our expectations as the GPU is not being fully utilized when the number of particles is low. When the workload is high, the GPU is able to fully leverage its parallel architecture.

Synchronization in the GPU Implementation and Runtime Breakdown:

Synchronization plays a crucial role in ensuring both the correctness and efficiency of our GPU implementation. Since CUDA kernels execute asynchronously, without proper synchronization, data dependencies between different computational phases can result in race conditions, incorrect computations, or inefficient memory accesses. In our implementation, cudaDeviceSynchronize() is strategically placed to ensure that each GPU kernel completes execution before proceeding to the next step. This is particularly critical for key operations such as memory management, computation phases, and binning operations.

One of the primary areas where synchronization is necessary is memory communication between the CPU (host) and GPU (device). cudaMemcpy() operations are used to transfer bin

offsets and coordinate particle indices between different kernels, ensuring that each phase of the computation has accurate, updated data before proceeding. Additionally, synchronization is crucial in computation phases, particularly in force calculations and movement updates, where atomic operations (atomicAdd) are used to prevent race conditions when multiple threads attempt to update acceleration values simultaneously. Without synchronization, the summation of forces acting on a particle could be corrupted by conflicting writes, leading to unpredictable results. Furthermore, parallel scan operations, such as the prefix sum computation for bin offsets, rely on synchronization to ensure that the results are completely updated before subsequent kernels use the bin indices.



To better understand the runtime breakdown of our simulation, we record execution time across every step of the simulation for each of the components that fall into three categories:

1. Computation Time:
    a. Force_computation_Time (compute_forces_gpu())
    b. Bin_assignment_Time (assign_bins())
    c. Move_particles_Time (move_gpu())
2. Communication Time:
    a. Memory_copy_time
        i. Resetting bin_counts at each step (cudaMemset)
        ii. Copying bin_offsets at each step (cudaMemcpy)
    b. Prefix_sum_time (compute_prefix_sum() using Thrust::exclusive_scan)
    c. Scatter_particles (scatter_particles() - Coalescing ids in bin-order)
3. Synchronization Time:
    a. cudaDeviceSynchronize - Time spent waiting for kernels to complete due to dependencies between computational steps.

From the stacked area plot above, showing the relative contributions of each component, we see that at lower particle counts, synchronization and communication operations are more prominent. However, as particle numbers increase, computation components become the dominant bottleneck, consuming a large fraction of the simulation time. This is expected because the force calculation still depends on iterating over neighboring bins, and even with optimized binning, particle interactions scale with increasing system size. Interestingly, synchronization overhead shrinks proportionally with problem size, reflecting the decreasing cost of ensuring proper execution order across multiple GPU threads. While communication grows, particularly the particle_scatter component, which should be expected with the increasing order of particle_ids that need to be sorted.

Conclusion:

In this project, we successfully optimized a two-dimensional repulsive force simulation using CUDA, leveraging GPU acceleration to achieve significant speedup over the naive O(n^2) implementation. By implementing a binning-based spatial partitioning strategy, we reduced redundant force computations and improved memory access patterns. The use of atomic operations for thread-safe updates and Thrust's exclusive scan for efficient prefix sum computation further contributed to the scalability of our approach. Benchmarking results demonstrated that while the starter code became intractable at large scales, our optimized implementation achieved near-linear performance scaling, making it feasible to simulate millions of particles efficiently.

A key takeaway from our analysis is the evolving nature of performance bottlenecks as the problem size increases. While synchronization and memory operations dominate at lower particle counts, force computation emerges as the primary bottleneck at large scales. The stacked area plot highlights how synchronization overhead, though necessary for maintaining data consistency, becomes proportionally smaller as computation time grows with particle interactions. Additionally, the increased cost of particle scattering at larger scales suggests an opportunity for further optimization, such as reducing memory accesses via shared memory or exploring alternative sorting techniques for particle organization.