CS267 HW2 Part 3 - Parallelizing a Particle Simulation

Charis Liao
MS in MSSE
charisliao@berkeley.edu

Apratim Banerjee
MEng in EECS
apratimbanerjee@berkeley.edu

Yulin Zhang
PhD in Computational Biology
zhangyulin9806@berkeley.edu

March 2024

1 Introduction

This assignment serves as an introduction to parallel programming utilizing Graphics Processing Units (GPUs), building upon concepts explored in previous homeworks. We embark on parallelizing a near-forces particle simulation. Within our simulation, particles interact through repulsive forces, with interactions occurring only when particles approach within a specified cutoff distance, depicted as a grey region surrounding each particle. While a naive approach to computing particle forces would yield a time complexity of $O(n^2)$, our simulation leverages a low particle density to achieve an expected time complexity of $O(n)$ due to the reduced number of interactions. The goal of this assignment is to exploit parallel computing techniques, specifically targeting GPUs, to achieve a significant speedup over the serial implementation, aiming for close to linear speedup (T/p) when utilizing p processors.

2 Method

2.1 First Attempt

1. **Sorted parts array**:
   We ensured the creation of a `sorted_parts` array, where each element contains the indices of particles sorted by their corresponding bin IDs. This approach allows for more efficient access to particles compared to having one array of length particles per bin. The `update_sorted_parts` kernel is responsible for computing the `sorted_parts` array.

2. **Count number of particles and synchronization**:
   The `update_bin_start_idx` kernel function iterates over all particles and maps each particle to its corresponding bin based on its position. Atomic operations (`atomicAdd`) are used to ensure that concurrent threads can safely update shared variables without causing race conditions or data corruption.

3. **Prefix sum the bin counts**:
   The `prefix_sum_bins` kernel function performs an exclusive prefix sum operation on the bin

counts array. This operation computes the cumulative sum of the bin counts up to each bin index, resulting in an array of prefix sums that represents the starting index of each bin in the sorted array of particles.

4. **Add particles to separate array starting from bin_idx**:
   The `update_sorted_parts` kernel function is responsible for updating the `sorted_parts` array based on the bin counts and prefix sum. It assigns each particle to the correct position in the sorted array according to its bin index. Atomic operations are used here to ensure correct indexing and thread safety when updating the array.

5. **What worked and what didn't**:
   `update_sorted_parts`: Although the code followed the standard guidelines and sanity checks by performing each step on the GPU in parallel, utilizing atomic instructions to handle concurrent updates to shared memory locations, and avoiding unnecessary memory copies, we were facing issues in compiling the code as the implementation was running into segmentation fault.

2.2 Second Attempt

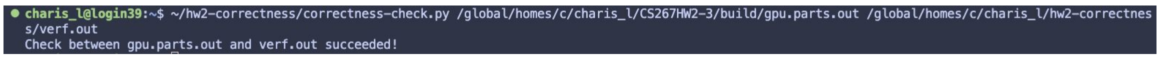
1. **Global Variables**:
   - Introduced the `bin_blks` variable to determine the number of blocks needed for bin-related operations.
   - Introduced the `bin_count` array to store the count of particles in each bin.

2. **Initialization**:
   - Modified the `gpu_init_arrays` function to allocate memory for the `bin_count` array in addition to existing arrays.
   - Calculated the `bin_blks` variable to determine the number of blocks needed for bin-related operations.

3. **Kernel Functions**:
   - Introduced a new kernel function `set_int_array` to initialize integer arrays with default values.
   - Modified the `update_bin_count` kernel to update the `bin_count` array with the count of particles in each bin.
   - Modified the `update_sorted_parts` kernel to update the `sorted_parts` array based on the `dynamic_assign_idx`.

4. **Simulation Steps**:
   - Modified the `simulate_one_step` function to initialize the `bin_count` array with default values.
   - Called the `update_bin_count` kernel to update the `bin_count` array.
   - Used Thrust library's `exclusive_scan` function to perform prefix sum operations on the `bin_count` array and store the result in the `bin_start_idx` array.
   - Called the `compute_forces_gpu_ON` kernel to compute forces based on the updated `sorted_parts` and `bin_start_idx` and called the `move_gpu` kernel to move particles.

5. **Results**:
   These modifications gave a successful code compilation. There was no segmentation fault, and the code passed the correctness check. The performance speed was also good.



```
● charis_l@login39:~$ ~/hw2-correctness/correctness-check.py /global/homes/c/charis_l/CS267HW2-3/build/gpu.parts.out /global/homes/c/charis_l/hw2-correctnes
  s/verf.out
  Check between gpu.parts.out and verf.out succeeded!
```

Figure 1: Code passing correctness check

3 Results

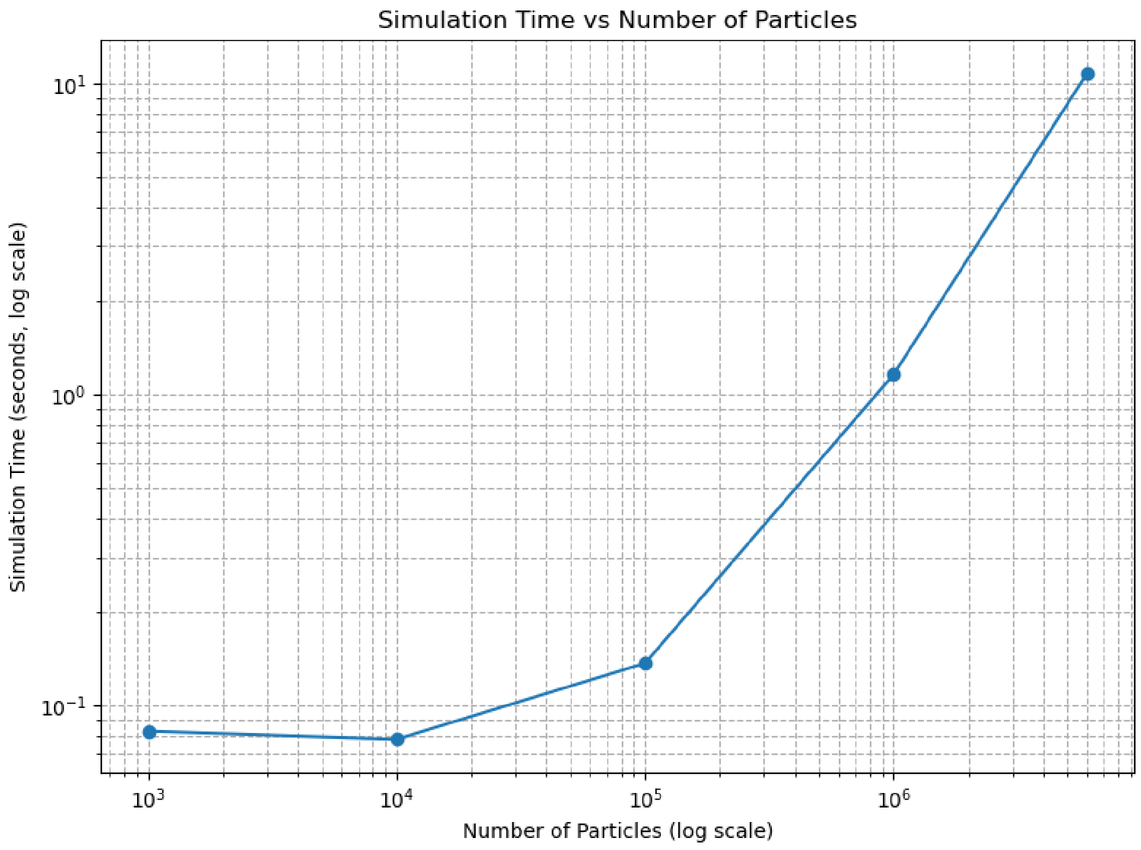
3.1 Simulation Results for varying particles

We tested the performance of our implementation by varying it for a number of particles. The performance is below. The configuration for this is 1 CPU core and CPU thread, targeting the NVIDIA A100s, with 4 GPU's per node.

Number of Particles	Simulation Time (seconds)
1000	0.0828411
10000	0.0779872
100000	0.136895
1000000	1.15956
6000000	10.7499

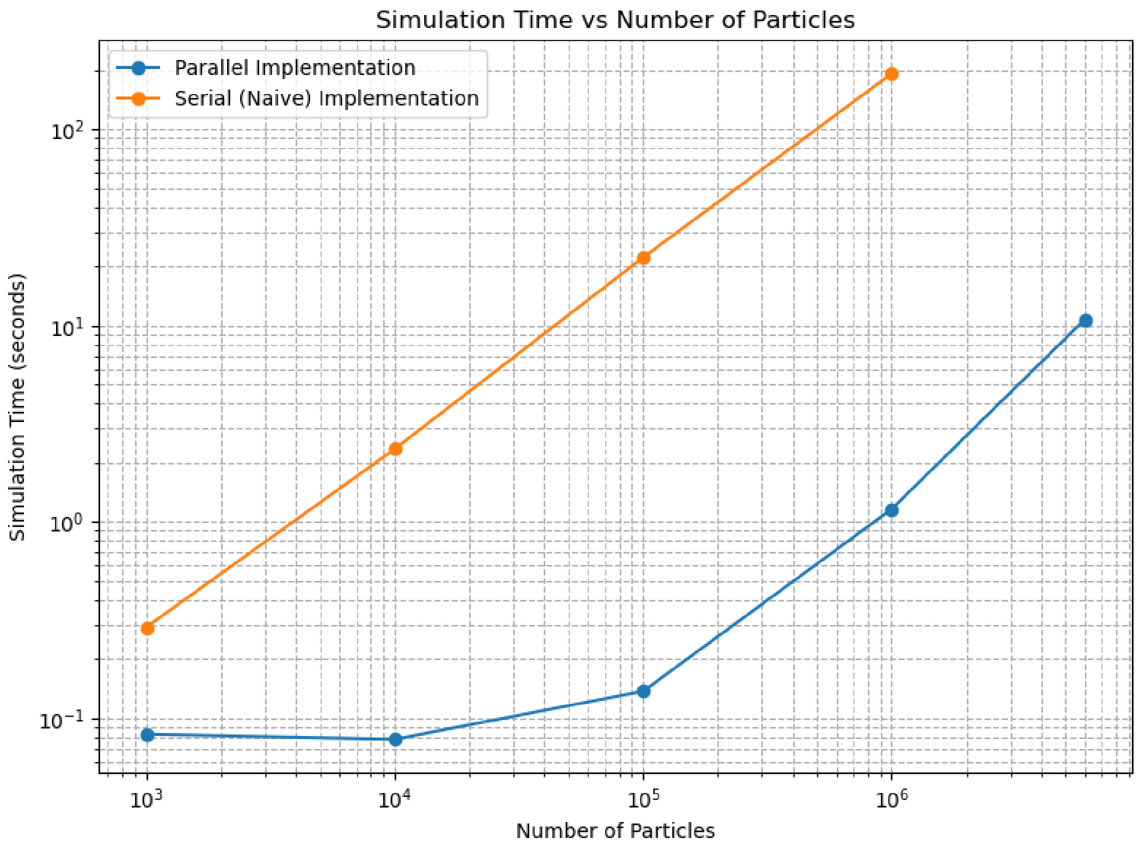
Table 1: Simulation Time for Different Numbers of Particles

3.2 Log-log scale showing parallel code's performance



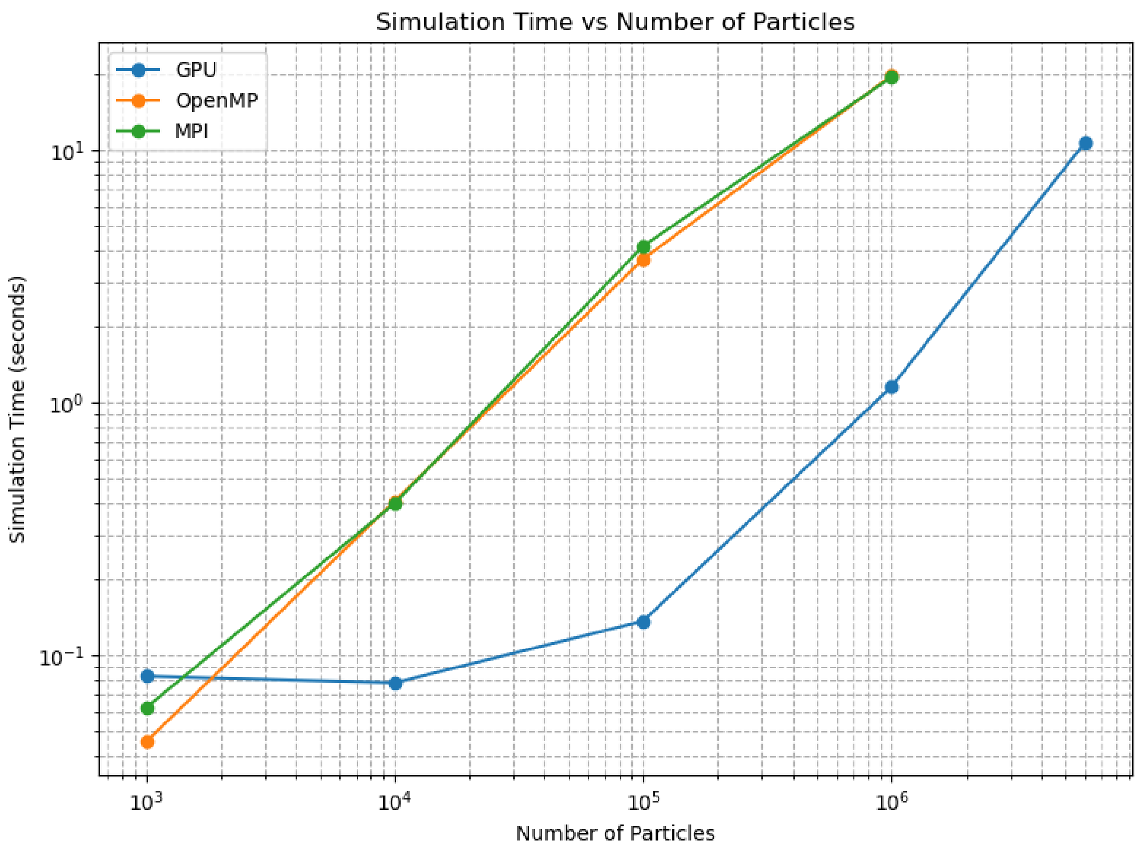
The graph exhibits a near-linear behaviour for higher number of particles, i.e. more than 10^5 particles in the log-log scale.

3.3 Benchmarking GPU implementation against the starter code



Both graph exhibits a near-linear behaviour. For parallel implementation, the linear behaviour is more prominent for higher number of particles, i.e. more than 10^5 particles in the log-log scale.

3.4 Benchmarking GPU parallelization against the OpenMP and MPI parallelization.



Based on our observations, we noticed that both OpenMP and MPI graphs exhibit similar linea rspeed-up as the number of particles increase, but GPU implementation is much faster than the former.

4 Conclusion

In this report, we have successfully parallelized a near-forces particle simulation using Graphics Processing Units (GPUs), focusing exclusively on GPU computing with one CPU core and thread. Our implementation involved careful consideration of synchronization mechanisms to ensure data consistency and efficient parallel execution on the GPU.

Regarding synchronization, we employed atomic operations, such as `atomicAdd`, to manage concurrent updates to shared variables, ensuring thread safety and preventing race conditions. Throughout the development process, we experimented with various design choices to optimize performance. These choices included the introduction of global variables for efficient memory management, kernel optimizations to minimize memory accesses and maximize parallelism, and algorithmic optimizations to reduce computation overhead.

The impact of these design choices on performance was significant. By carefully managing data access patterns and minimizing synchronization overhead, we achieved substantial speedup compared to the serial implementation. Benchmarking against the starter code revealed consistent improvements in computational efficiency, with the GPU implementation showcasing near-linear scalability with increasing numbers of particles.

Furthermore, we compared our GPU implementation with OpenMP and MPI implementations, focusing on a fixed number of nodes/threads for each. While OpenMP and MPI exhibited competitive performance under specific conditions, our GPU code outperformed them consistently as the number of particles increased, highlighting the superior scalability and efficiency of GPU-accelerated computing.

In conclusion, our successful parallelization of the particle simulation on GPUs underscores the transformative potential of GPU-accelerated computing for scientific simulations and similar computational tasks. Through careful synchronization, design optimization, and performance benchmarking, we have demonstrated the scalability, efficiency, and superiority of GPU computing over traditional CPU-based approaches. Moving forward, continued refinement and optimization of GPU algorithms and techniques promise to unlock further advancements in parallel programming and scientific computing.

5 Contribution

Apratim helped with debugging and wrote the report. Charis worked on the initial implementation of the code, helped with debugging, and helped writing a part of the report. Yulin implemented the correct version of the code, and ensure correctness and right implementation.