HW 2-3 Report

Group 15: Yinuo Zhang & Adrisha Sarkar

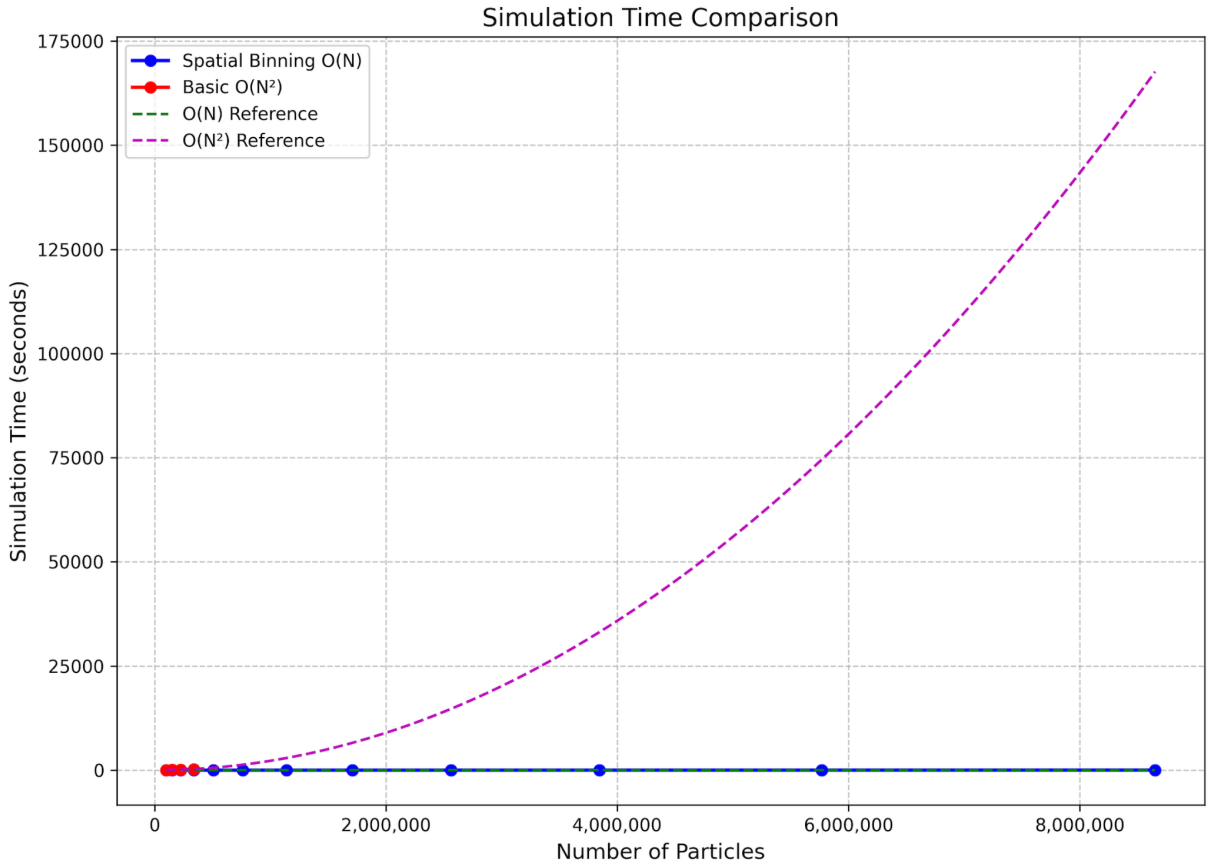
Contribution:

Yinuo: Final implementation and optimizations submitted were written by Yinuo.

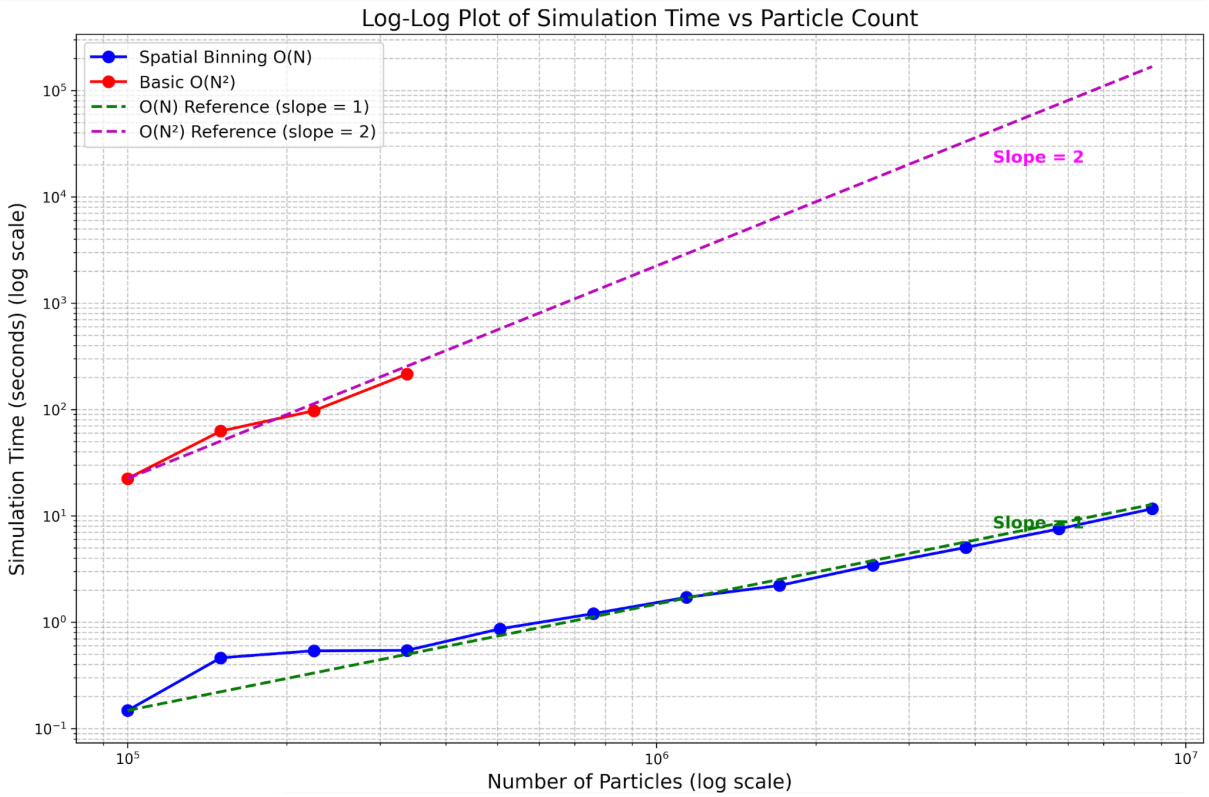
Adrisha: Memory shared memory optimization tried, parallelization approach with more threads with slightly different memory access that ended up with worse time for low num particles and 1-2% worse for higher number particles compared to the submission. Generated plots, time analysis code and collected all data for scaling and computation times and wrote the report.

GPU parallelization of particle interaction simulation

Results:
Here we plot the absolute simulation time comparison for our parallel code and the starter code. The linear scale plot illustrates the performance gap between the two implementations as particle count increases. Our GPU parallelization with spatial binning implementation is bounded by O(N) and the serial/ basic implementation is bounded by O(N^2). The dashed lines are extrapolated reference lines.



The same data, visualised as a log-log plot below demonstrates that we achieve O(N) scaling compared to the O(N²) performance.



Approach for our O(N) GPU implementation:

Key Ideas:

We need to describe our data structure here.

1) We use spatial binning to ensure particles interact with adjacent bins.
2) We scale the number of blocks with the particle counts. This is to ensure efficient GPU utilization across the block on different problem sizes.
3) Memory access was something that we played around with. In the final submission we have a design that separates particle data from bin metadata to improve memory coalescing. In summary, particle positions remain in a simple array and are accessed during movement. We tried to optimize bin neighbor searching. There is still room for different/ better implementations here but we achieve linear scaling.
4) Our rebinning process is computationally dominant. It involves two main steps in which to check for particles per bin and then place them.
5) **Fail implementation**: We can see in our performance that there is some memory access latency. We tried a shared memory optimization for the force calculation, which can significantly reduce global memory access and improve performance. This is particularly effective for our simulation since each thread needs to access neighboring particle data multiple times. This involved trying to load particles into shared memory, which is much faster than global memory. The GPU can then cooperatively load particles in a pattern that promotes coalesced memory access. We tried to use local registers to accumulate forces, then perform a single atomic update at the end. This also keeps the current particle in a register for

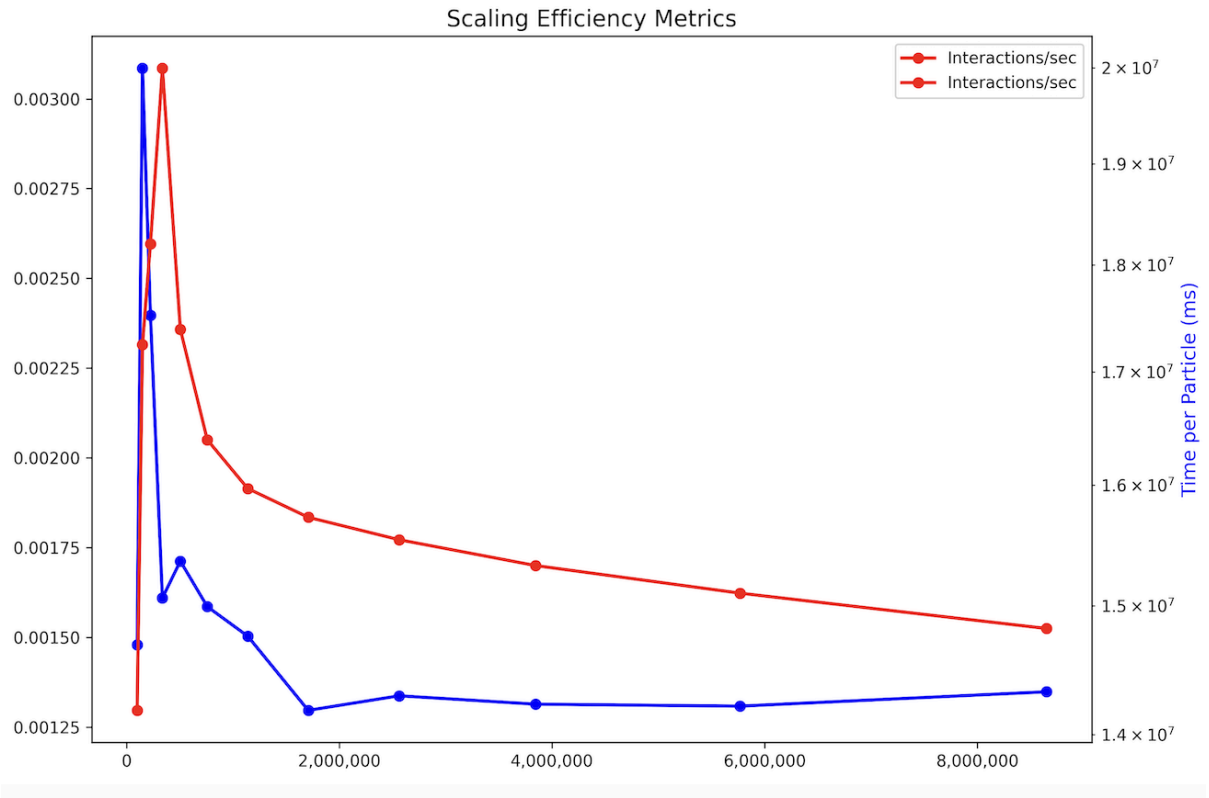
faster access. However, in the correctness check we have some subtle bugs. The particle correctness starts off correct and then by the very end is slightly different. With some more time this implementation could be finished and would possibly change the overall percentage of computational time distribution that we observe in our current submission.

Data Structure and key functions:

For the binning strategy the primary array stores all particle data in a 1D array. The position velocity and acceleration are stored. The binning grid takes the full space of the simulation area and splits its into a 2D sections area. A second 1D array is used to store the particle IDs and is the same size as the primary array. Here we store the particle indices reorganised so that the particles in the same bin are adjacent. The prefix sum used allows particles to be efficiently sorted by bin without actually moving the particles themselves. It is essentially creating a mapping between bins and their location in the sorted particle ID array, enabling efficient traversal of particles by spatial location.We use a prefix sum to to transform the array by replacing each element with the sum of all previous elements. It is key for the binning and each index, i, in the array stores the coin of particles in bin i. This makes the counts a cumulative starting position. This is used for looking up bins. When computing forces, the code can quickly find the start and end indices for particles in any bin.

Further Analysis of performance and optimization strategy:

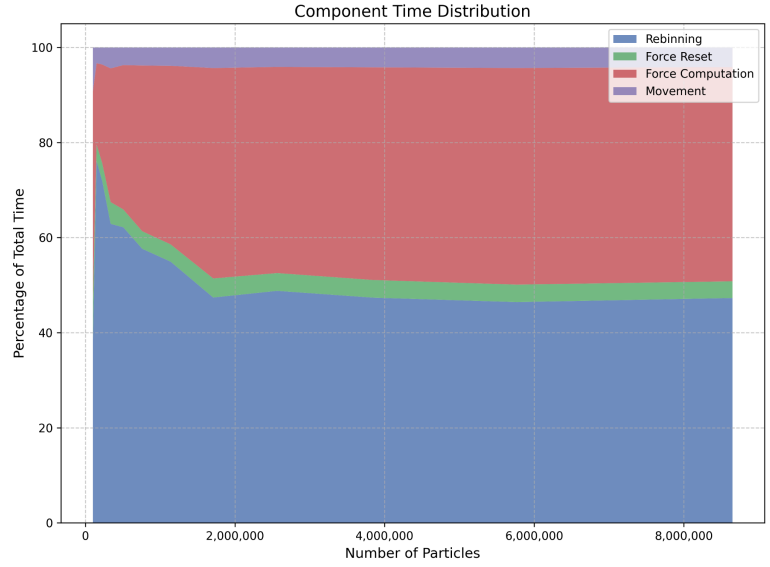
The plot below shows two efficiency metrics as a function of particle counts. The red line and left y axis show the Interactions/sec. The blue line and right y axis show how the processing time per particle changes with increasing particle numbers i.e. Interactions/sec per Particle. Both metrics show initial volatility at low particle counts. As particle count increases, both metrics stabilize - time per particle settles at approximately 0.00135ms, while the interaction processing rate stabilizes around 1.6×10^7 interactions per second. The slight downward trend in interactions/second at higher particle counts might indicate increasing cache pressure or memory bandwidth limitations.



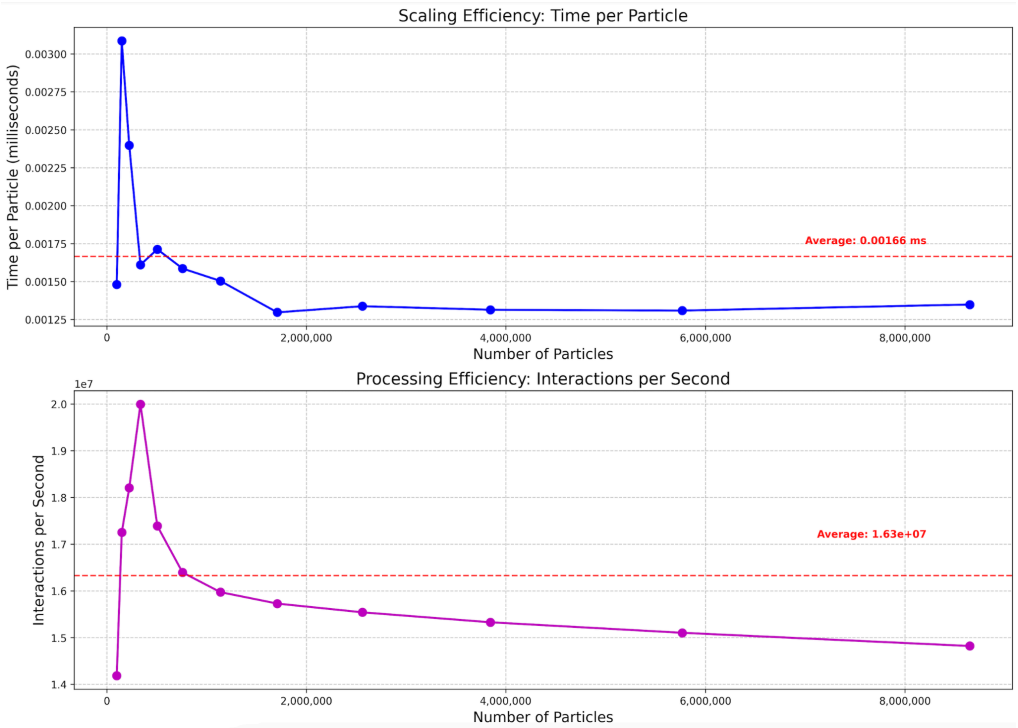
The plot below shows the relative contributions of each simulation component in the total runtime extrapolated from the time needed for each function to run. We use this to understand what is computationally time intensive, where our optimizations work and could be improved. Force computation consistently takes ~45% of runtime, rebinning ~50%, with force reset and movement operations taking minimal time. We tried another rebinning strategy to improve this.

Breakdown of computation time:

- **Rebinning** (blue): assigning particles to spatial bins
- **Force Reset** (green): zeroing forces before computation
- **Force Computation** (red): calculating inter-particle forces
- **Movement** (purple): updating particle positions in bins



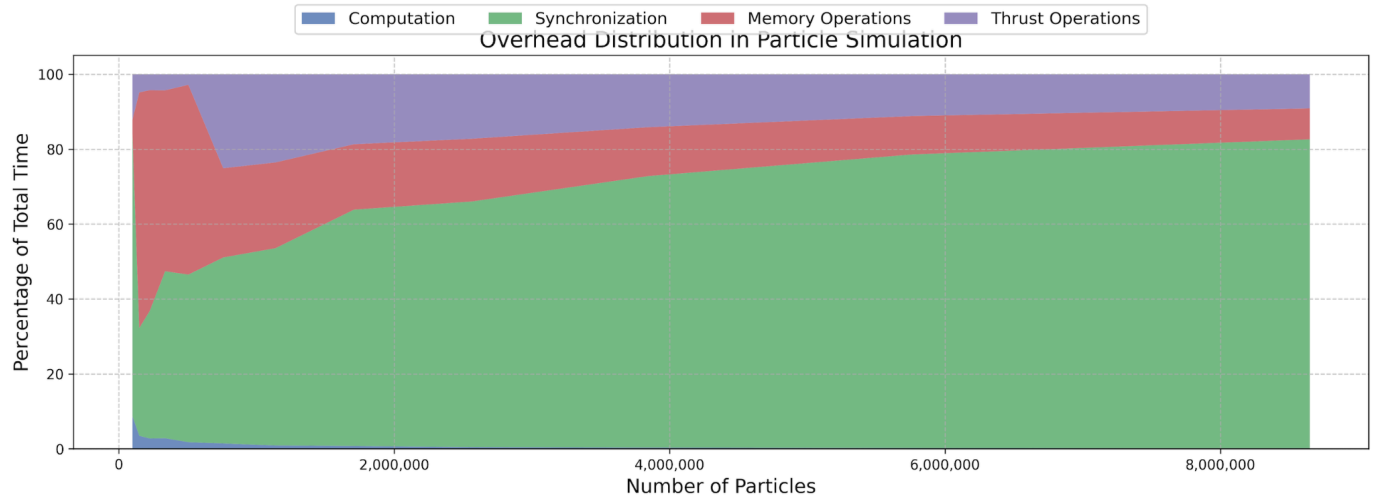
The plots below: 1st plot, tracks the time required to process each particle as system size increases and 2nd plot, below shows number of particle interactions processed per second. The stabilization confirms that our implementation achieves proper O(N) scaling. The gradual decrease at higher particle counts suggests some loss of efficiency, possibly due to increased memory pressure or cache effects.

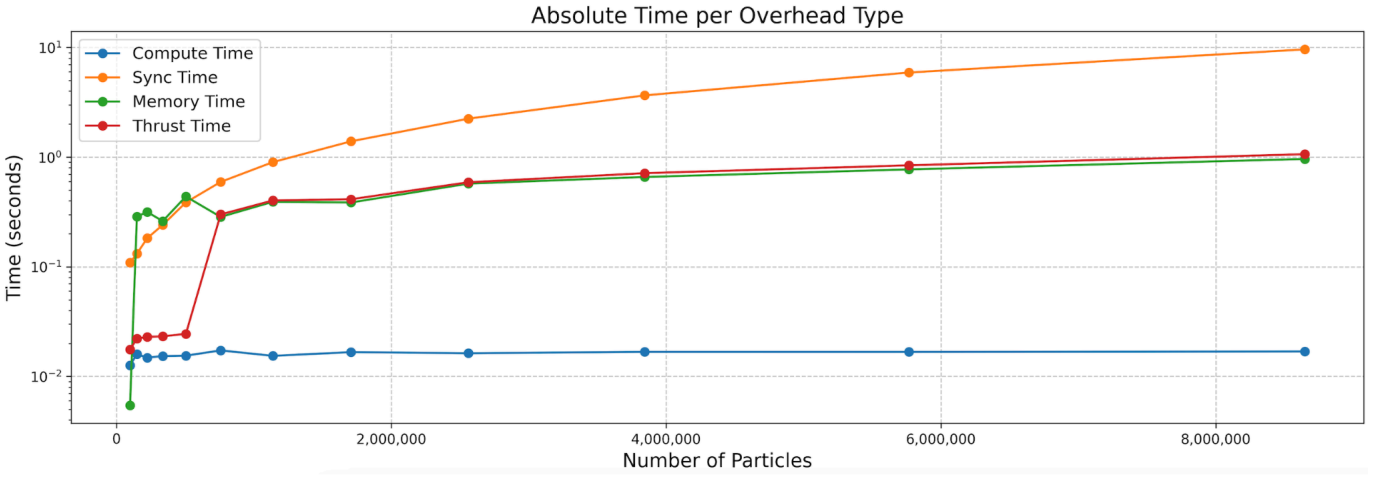


This plot below shows that synchronization becomes increasingly dominant as particle count grows, eventually accounting for ~70-80% of total runtime. This suggests that optimizations reducing synchronization requirements could be effective in further performance improvements on our implementation. The absolute times are plotted also below.

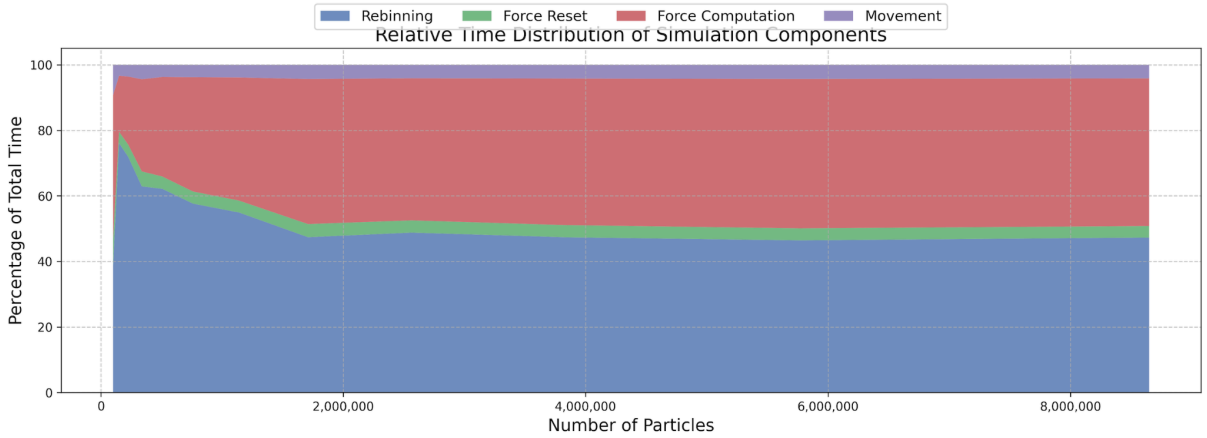
Break down of simulation time into different types of overhead:

- **Computation** (blue): pure computation time (roughly constant cost)
- **Synchronization** (green): time spent in CUDA synchronization (increase with particle nums)
- **Memory Operations** (red): memory allocation, copy, and free operations (not costly)
- **Thrust Operations** (purple): time spent in Thrust library calls (not costly)

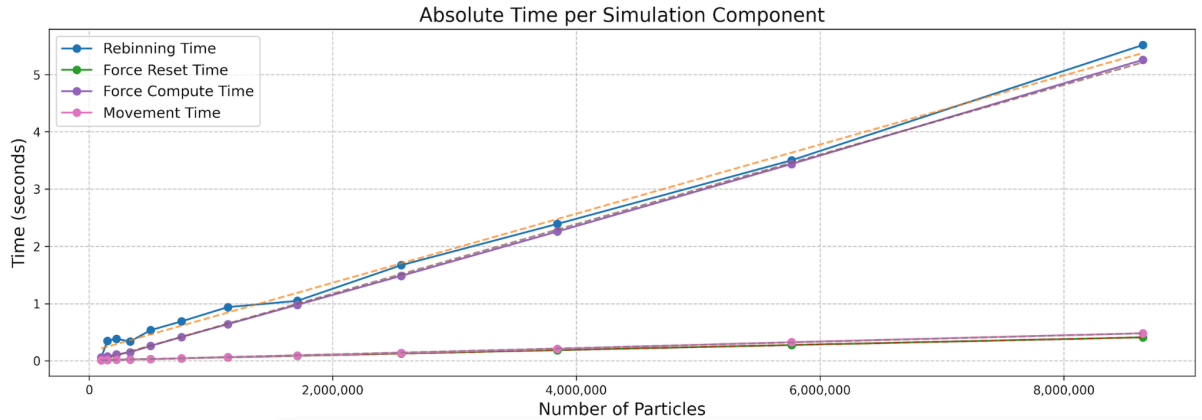




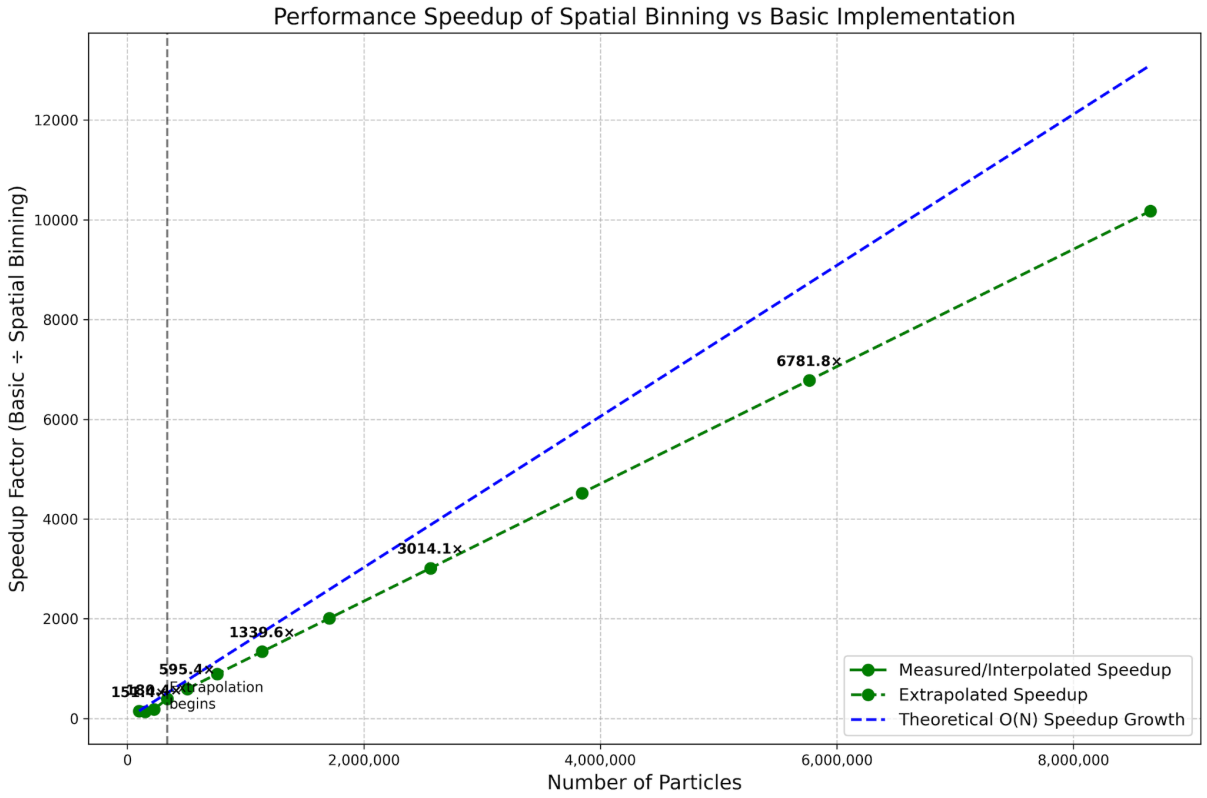
In this plot below we can see that rebinning and force computation dominate computational time; together they account for ~95% of total time. The other components remain relatively insignificant across all particle counts. This aligns with our implementation as rebinning is a rate limiting step and suggests further optimization efforts should focus on the two dominant components.



In the below plot we visualise that each component scale linearly with particle count, confirming O(N) behavior. Rebinning and force computation have nearly identical growth rates and absolute times, while force reset and movement operations consume much less time even at large particle counts. This aligns with everything we have in our implementation optimization.



The measured/interpolated speedup grows linearly with particle count, reaching factors of thousands for large simulations. The blue dashed line represents the theoretical O(N) speedup growth. This plot below illustrates the scaling advantage of spatial binning, with specific speedup factors labeled at different particle counts using a GPU.



Comparison to previous implementations:

The performance hierarchy: Serial < OpenMP < MPI < GPU

Comparing the GPU and MPI implementations of the particle simulation reveals that the GPU version significantly outperforms the best MPI configuration (32 processors) by 35-80× across different particle counts. Both implementations achieve O(N) scaling through spatial binning, but leverage different architectural advantages: the GPU excels through massive parallelism, higher memory bandwidth, and a unified memory space without explicit communication overhead. My MPI version demonstrates interesting super-linear scaling at certain configurations due to cache effects at certain particle numbers. But it is overall worse. The primary bottlenecks also differ fundamentally – the

GPU is limited by synchronization overhead (70-80% of runtime at high particle counts), while the MPI implementation is constrained by communication costs that scale as $O(\sqrt{N})$. This performance gap clearly indicates that for particle simulations that fit within GPU memory, the GPU-based approach represents a substantially more efficient solution. The OpenMP implementation represents parallelization using shared-memory parallelism within a single node. The OpenMP version outperforms the serial implementation but falls short of the GPU implementation's performance also. Performance of the openMP scales more linearly than MPI at lower thread counts but hits memory bandwidth limitations.

Conclusion:

1) Time per particle stabilizes at approximately 0.00166 milliseconds
2) Interaction processing rate settles around 1.63×10^7 interactions per second
3) Both metrics indicate good strong scaling with our GPU implementation

The data structures and algorithm chosen provide very good performance and this implementation is highly effective for large-scale particle simulations on GPU hardware achieving O(N) scaling. The implementation is much better than both the openMP and MPI versions. The synchronization overhead remains the primary limiting factor, suggesting that future optimization efforts could focus on reducing synchronization requirements or overlapping computation with synchronization where possible.