HW 2.3 Report

Emma Brugman Sabrina Temesghen Dulce Torres

Introduction

In this assignment, we utilized NVIDIA GPUs to parallelize a particle simulation on Perlmutter. The overarching goal was to achieve O(N) time complexity by leveraging spatial binning and GPU-based bin assignment and sorting to limit force computations to a particle's local neighbors. This report outlines our implementation, including kernel design, synchronization methods, and memory management strategies, optimized for execution on an A100 GPU.

Methods

Bin Counting *(bin_count_kernel)* & Sorted Particle Array *(bin_sort_kernel)*:

Assigning particles to spatial bins using the atomic *bin_counts* to reduce the computational complexity of O(N^2) to O(N) from reducing the number of force calculations. This is done by only applying force computations to the local neighbors using their bin assignments based on spatial coordinates. Each bin's indices *(bin_x, bin_y)* are computed but dividing the particles (x, y) positions by the bin size *(bin_size)* and then stored as the bin id *(bin_id)*. To avoid race conditions during bin assignments, we used atomic operations *(atomic_add)* so that the particle counts are incremented into the bins accurately.

Once we have the resulting 1D array of the number of particles in each bin, we implement a prefix_sum indexing (*bin_prefix*) using the thrust libraries, *thrust::exclusive_scan.* This allows for the prefix sum array to set a start index for each bin to efficiently retrieve particles from bins in the subsequent kernels.

Using bin_sort_kernel, the particles are organized into their respective bins based on their (x,y) coordinates. An atomic array *(bin_offsets)* is used to track the particle's position within each bin so that each particle's position *(target_idx)* is different. The position is determined by the sum of the starting index *(bin_prefix[bin_id])* and the offset within that bin. The sorted particles *(sorted_particles)* and their sorted ids *(sorted_ids)* were used to efficiently and accurately carry out particle reordering after GPU computations complete.

Force:

The *compute_forces_gpu* function was revised to calculate particle interactions strictly within neighboring spatial bins. Each GPU thread is mapped to a single spatial bin and processes all particles assigned to that bin *(bin_start, bin_end)*. Neighboring bins are identified based on their spatial coordinates (*nx, ny*), and particle data from these neighboring bins is directly accessed from shared memory (sorted_particles). This localized approach reduces computational complexity from O(N^2) to approximately O(N), significantly improving efficiency. Force interactions between particles are computed using the GPU-specific function *apply_force_gpu*, optimized for memory access patterns and computational performance within the CUDA environment.

Particle Reordering:

At the end of the *simulate_one_step*, once the force and move computation have been implemented, the particles are stored in a sorted bin order that is not suitable for maintaining the particles identity across time steps as it will lead to an inaccurate tracking of each particle over time. To prevent this, we reordered the particles back to their original ordering by copying the *sorted_particle_gpu* and *sorted_ids_gpu* arrays back to the host. Each particle is reordered by their *sorted_ids* to its corresponding original index in the global memory.

GPU Synchronization & Memory Access:

AtomicAdd

As mentioned above, we used atomic operations *(atomicAdd)* in *bin_count_kernel* and the *(bin_sort_kernel)*. Using the atomic operations we can safely manage the written memory in the GPU kernels. This synchronization method was used to serialize access to the shared memory location in gpu. By preventing multiple threads from simultaneously working in the same bins we guarantee that the particle positions within the sorted bins are correct while also preserving the integrity of the data.

CudaMemcpy

Synchronization between the kernels and the CPU-GPU interactions were also managed using memory transfer functions . The CPU and GPU global memory transferred data using CUDA API function *(cudaMemcpy)* to synchronize the GPU computations with CPU operations. As seen after using the GPU kernels *compute_force_gpu* and *move_gpu,* the particles were copied from GPU memory *(sorted_particles_gpu* and *sorted_ids_gpu)* and back to the CPU memory *(sorted_particles_host* and *sorted_ids_host)* for the particle reordering. Host and device transfers under cudaMemcpy were explicitly using *cudaMemcpyHostToDevice* and *cudaMemcpyDeviceToHost* **(Figure 1)**.

```
// Reordering step
std::vector<particle_t> sorted_particles_host(num_parts);
std::vector<int> sorted_ids_host(num_parts);

cudaMemcpy(sorted_particles_host.data(), sorted_particles_gpu, num_parts * sizeof(particle_t), cudaMemcpyDeviceToHost);
cudaMemcpy(sorted_ids_host.data(), sorted_ids_gpu, num_parts * sizeof(int), cudaMemcpyDeviceToHost);

std::vector<particle_t> reordered_particles(num_parts);
for (int i = 0; i < num_parts; i++)
    reordered_particles[sorted_ids_host[i]] = sorted_particles_host[i];

cudaMemcpy(parts_gpu, reordered_particles.data(), num_parts * sizeof(particle_t), cudaMemcpyHostToDevice);
```

Figure 1. Reordering step in GPU parallelization. After GPU computation, particle data is transferred back to host memory using cudaMemcpy to restore the original particle order

Memory on the device itself was also carefully managed. The GPU arrays *(bin_counts, bin_offsets, and bin_prefix)* were allocated using the thrust vectors to manage memory on the device and integrated with the CUDA pointers *(thrust::raw_pointer_cast).* In combination, *thrust::exclusive_scan* illustrates

synchronization between the host and device as the prefix sum is computed on the device but immediately used in the device kernels.

Lastly, all of the memory allocated on the device from *sorted_particles_gpu, sorted_ids_gpu, and particle_ids_gpu,* were all freed using *cudaFree. CudaFree* deallocates the memory that was previously allocated using the cudaMalloc().

Exploratory Versions from Team Members

In this section, team members developed an independent version of the GPU kernel. These implementations explored different strategies and overlapping techniques in comparison to the final code.

Optimizations to the given starter code took into consideration the memory hierarchy of the A100 GPU. Emphasis was made utilizing shared memory in order to limit global memory access. Several helper functions were added to facilitate the use of shared memory in the compute forces function.

The first helper function, bin_particles, allocated particles to a specific bin depending on their assigned bin identification numbers. Bin identification numbers were assigned based on a particle's x and y coordinate. The number of particles assigned to each bin was recorded using a global array, bin_counts. In order to safely increment the count associated with a bin, atomicAdd was utilized.

Once each particle was assigned to a bin, particles were allocated to a position in a sorted array by bin identification number. This consisted of applying thrust's inclusive scan to calculate an array of prefix sums, bin_offsets, from the bin_counts array. Since bin_offsets is a globally shared variable, atomicAdd was used to access a bin's corresponding prefix sum. On the other hand, since each position in the sorted array corresponded to a single particle in the system, no atomic operations were necessary when writing to the sorted array.

Bin assignments were utilized in the compute_forces_gpu function in order to limit the access to global memory. Every thread in the grid was assigned a unique particle based on thread id. Neighboring bin cells were then determined based on the bin identification number associated with the particle. Particles from these neighboring bins were then written into a shared array. This calculation was followed by a synchronization step to ensure all particles were written into the shared array prior to computing forces. A for loop was then used to apply the force between a thread's particles and all the particles in the shared array.

After applying the force, the move function was applied across particles in the system. In order to distribute the move calculation across threads, each thread was responsible for moving particles in their respective bins. Although the focus of such implementation was to improve performance through limited access to global memory, the resulting time complexity was not linear.

In another attempt, optimizations to the starter code were aimed at improving the performance through spatial locality and GPU memory efficiency.

Binning

Binning was implemented as a strategy to reduce the time complexity to O(n) by limiting the particle interactions to local neighborhoods. The grid was divided into a 2D grid of uniform bins. Each particle was then assigned to a bin based on its position. This allowed the simulation to compute interactions only within a particle's own bin and its immediate neighbors, which reduced the number of calculations.

Thrust Functions

In order to facilitate the binning, Thrust functions were utilized to implement the prefix sums and bin sorting. Once each particle was assigned to a bin ID and the number of particles per bin was tracked using an array, a prefix sum was performed using Thrust's *inclusive_scan* function to produce bin start indices, where each bin's particles begin in an array. Additionally, Thrust functions *transform*, *fill*, and *raw_pointer_cast* were utilized to assign bin IDs and manage memory across device vectors.

Atomic Instructions

Atomic instructions were used to update the bin counts array to ensure correctness within the parallel execution. The array that contained the particle ids also utilized *atomicAdd* based on the prefix sums to avoid write collisions when placing particles into their respective bins.

Shared Memory

To optimize the performance of this implementation, shared memory was utilized into the force computations. Rather than access particles from global memory, particles from neighbor bins were loaded into shared memory blocks. The threads computed the forces using the cached values. Synchronization with *__syncthreads()* ensured that the particles were in shared memory before performing the force calculations.

Results

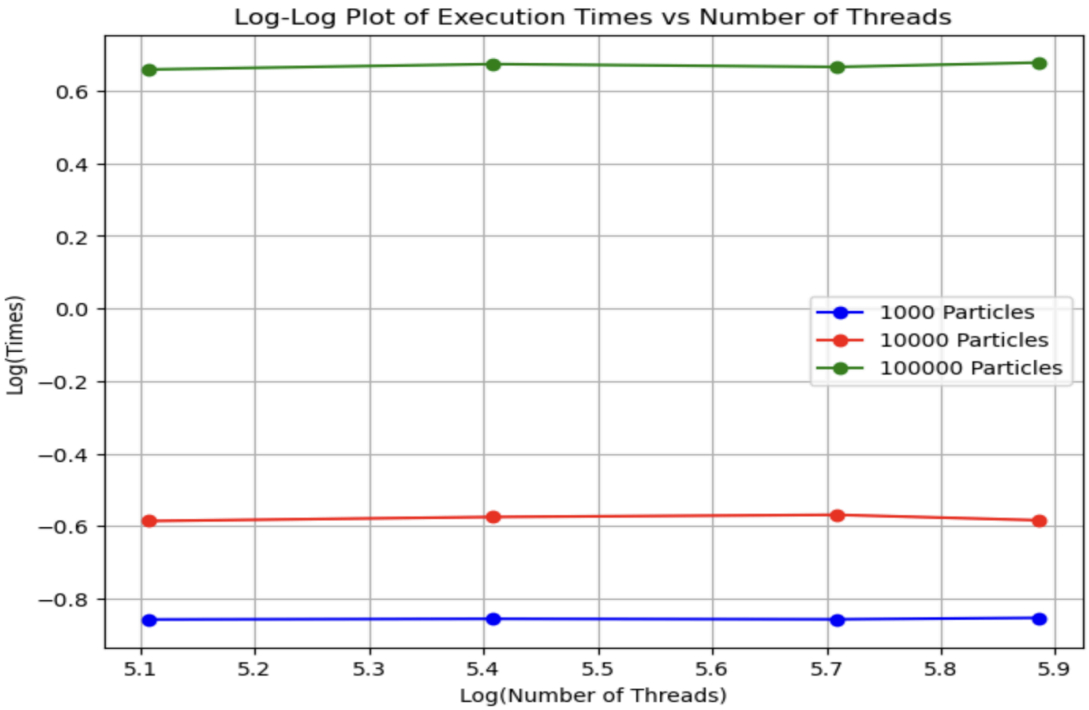

Figure 2. Strong scaling plot for 1000 particle, 10,000 particle, and 100,000 particle systems.

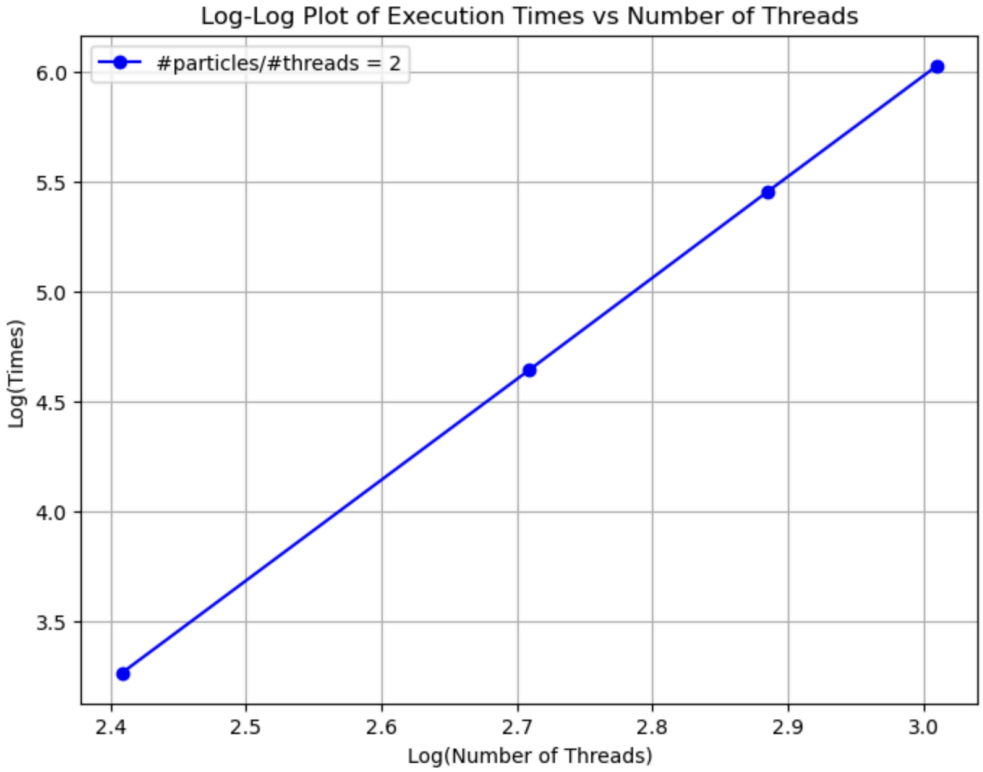

Figure 3. Weak scaling plot for a particle to thread ratio of 2.

```
Time (%)  Total Time (ns)  Num Calls  Avg (ns)   Med (ns)  Min (ns)   Max (ns)    StdDev (ns)          Name
--------  ---------------  ---------  ---------  --------  --------  -----------  -----------  ----------------------
    46.5      830,202,573      5,000  166,040.5   6,377.0     2,365   12,242,837    288,744.2  cudaStreamSynchronize
    23.9      426,647,056      7,001   60,940.9   4,849.0     2,264  389,146,887  4,650,801.1  cudaMalloc
    13.0      232,899,722      4,001   58,210.4  14,998.0     4,057    1,097,499    142,896.2  cudaMemcpy
    11.7      209,571,286      7,001   29,934.5   4,609.0     2,064   10,046,448    157,829.4  cudaFree
     4.1       72,432,665      9,000    8,048.1   7,204.0     3,797      203,910      5,641.7  cudaLaunchKernel
     0.5        8,486,589      1,000    8,486.6   7,318.5     3,998      196,507      9,270.5  cudaMemcpyAsync
     0.3        4,525,014      1,001    4,520.5   2,285.0     1,263    2,059,884     65,060.8  cudaDeviceSynchronize
     0.0            7,514          1    7,514.0   7,514.0     7,514        7,514          0.0  cuModuleGetLoadingMode

[4/6] Executing 'cuda_gpu_kern_sum' stats report

Time (%)  Total Time (ns)  Instances  Avg (ns)  Med (ns)  Min (ns)  Max (ns)  StdDev (ns)                                                                                                  Name
--------  ---------------  ---------  --------  --------  --------  --------  -----------  ----------------------------------------------------------------------------------------------------
    26.1        7,342,053      1,000   7,342.1   7,360.0     3,520     9,440        709.2  compute_forces_gpu(particle_t *, int, int *, int, double)
    18.8        5,280,325      3,000   1,760.1   1,696.0     1,663     2,144        111.1  void thrust::THRUST_200302_800_NS::cuda_cub::core::_kernel_agent<thrust::THRUST_200302_800_NS::cuda…
    17.5        4,923,010      1,000   4,923.0   4,608.0     4,512     6,208        512.2  bin_sort_kernel(particle_t *, particle_t *, int *, int *, int, int *, int, double)
    12.8        3,590,938      1,000   3,590.9   3,456.0     3,167     4,609        341.7  void cub::CUB_200302_800_NS::DeviceScanKernel<cub::CUB_200302_800_NS::DeviceScanPolicy<int, thrust:…
    10.7        2,995,144      1,000   2,995.1   3,040.0     2,783     3,936        154.4  move_gpu(particle_t *, int, double)
     8.8        2,461,434      1,000   2,461.4   2,304.0     2,239     3,105        263.4  bin_count_kernel(particle_t *, int, int *, int, double)
     5.3        1,488,870      1,000   1,488.9   1,408.0     1,344     2,016        183.4  void cub::CUB_200302_800_NS::DeviceScanInitKernel<cub::CUB_200302_800_NS::ScanTileState<int, (bool)…

[5/6] Executing 'cuda_gpu_mem_time_sum' stats report

Time (%)  Total Time (ns)  Count  Avg (ns)  Med (ns)  Min (ns)  Max (ns)  StdDev (ns)          Operation
--------  ---------------  -----  --------  --------  --------  --------  -----------  ----------------------------
    60.4        7,460,620  3,001   2,486.0   1,344.0     1,024    17,600      2,151.4  [CUDA memcpy Host-to-Device]
    39.6        4,897,881  2,000   2,448.9   2,735.0     1,279   174,817      3,935.5  [CUDA memcpy Device-to-Host]

[6/6] Executing 'cuda_gpu_mem_size_sum' stats report

Total (MB)  Count  Avg (MB)  Med (MB)  Min (MB)  Max (MB)  StdDev (MB)          Operation
----------  -----  --------  --------  --------  --------  -----------  ----------------------------
    52.052  3,001     0.017     0.004     0.000     0.048        0.022  [CUDA memcpy Host-to-Device]
    52.000  2,000     0.026     0.026     0.004     0.048        0.022  [CUDA memcpy Device-to-Host]
```

Figure 4. GPU Runtime Breakdown by Operation. Profiler-based breakdown of GPU runtime across computation, memory operations, synchronization, and data transfer, highlighting the relative cost of each component.

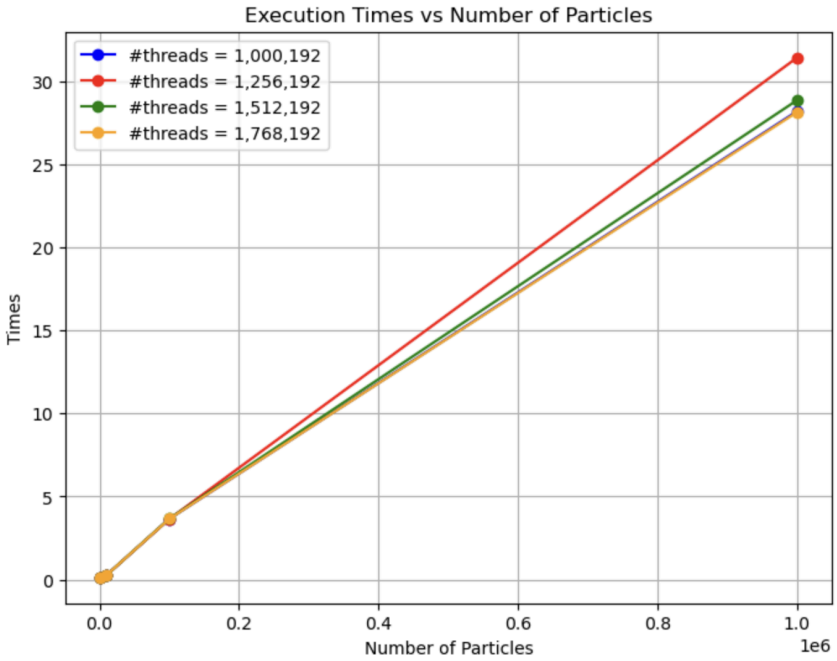

Figure 5. Time complexity plot for 1,000 particles, 10,000 particle, 100,000 particle, and 1,000,000 particle systems.

Discussion

Analysis of Figure 4 Runtime:

Computation Time:

The computation time is spent in the gpu kernel for *compute_force_gpu, bin_sort_kernel, move_gpu,* and *bin_count_kernel*. Approximately 26.1% of the total kernel time is spent in *compute_force_gpu.* This is not a surprise since the majority of the time is spent computing the pairwise interactions across each particle's neighboring bins. Binning the particles in the *bin_sort_kernel* and *bin_count_kernel* take up ~26.3% **(Figure 4)**. Force computation scales linearly with O(N) as the number of particles increases due to the spatial binning.

Synchronization Time:

An analysis of the overall runtime shows that GPU computation time is less than the combined synchronization and memory transfer overhead. Approximately 70% of the total GPU execution time is attributed to synchronization. Although we do not explicitly call *cudaStreamSynchronize()* in our code, the CUDA runtime performs implicit synchronization before memory transfers using *cudaMemcpy* and between kernel launches. These waits ensure that GPU operations complete before proceeding with host-side operations or data transfers. As the number of particles increases, this synchronization overhead is expected to grow especially without the use of asynchronous memory copies or overlapping computation and communication.

Communication:

Memory transferred between CPU and GPU using *cudaMemcpy* takes up ~13% as majority of this cost stems from transferring the sorted particle array and ID's to and from the device in the reordering step. The *Host-to-Device* transfers accounts for 60.4% while the Device-to-Host transfer accounts for 39.6% of the total communication time **(Figure 4)**. As the system grows, we can continue to expect this overhead to scale linearly with the number of particles.

Analysis of benchmarking plots

Performance of the optimized code was further assessed through strong scaling, weak scaling, and time complexity plots. As shown in **Figure 2**, the number of particles was held constant for three different systems while increasing the number of threads. An ideal performance would have been presented by an inverse linear relationship between the number of processes and the corresponding times– the time should have decreased with increasing thread count. However, instead, the time remains relatively constant as the number of processes increases. Taking into account the profiling shown in **Figure 4**, it is believed the lack of performance is due to the global memory access that occurs in the compute_forces_gpu functions. Similar trends were observed during weak scaling and time complexity analysis. Maintaining a constant ratio between the particle and process count should have resulted in a constant time across the different data points. However, the weak scaling plot shown in **Figure 2** shows a steady increase as the size of the system increases. Both strong and weak scaling results are consistent with what is observed in the time complexity plot of **Figure 5**. Although four different data points were plotted for several different threads, only three are shown in the plot. This is due to the rapid increase of time that is observed when increasing the size of the system. Thus, the time complexity does not scale to a linear time when increasing the size of the system. Overall, it is believed the trends observed across the different benchmarks are a result of memory access time.

Conclusion

Furthermore, future implementations would focus on cutting down memory access time. It was evident in both the time profiling and benchmark plots that a limitation to the current optimized code was access to global memory. Thus, future implementations would focus on utilizing a shared memory array within grid blocks rather than accessing a global array.

As previously discussed, synchronization time remains a dominant factor, both implicitly and explicitly. To address this, future optimizations may involve reducing synchronization frequency and minimizing the number of memory transfer calls. This can be achieved by overlapping memory transfers, for example, through the use of *cudaMemcpyAsync* instead of the synchronous *cudaMemcpy*. Additionally, further performance gains could be realized by performing the reordering step entirely on the GPU, thereby eliminating unnecessary host-device memory transfers.

Contributions:

Sabrina Temesghen: Implemented several parallelization methods, Contributed to write up
Dulce Torres: Implemented several parallelization methods, Performance Plots/Analysis and Results
Emma Bruggman: Implemented several parallelization methods, Contributed to write up