CS 267 HW 2-3 Report: Parallelizing Particle Simulation

Kathryn Wang, Milad Shafaie (Group 65)

March 25, 2025

1 Member Contributions

- Milad
  - Made optimizations to Kathryn's initial implementation to get runtime down from 13.9 seconds to 9.2 seconds on 10M particles, including optimizing sorting of particles by bin, local force accumulation, limiting number of blocks for some kernels, and attempting shared memory optimizations.
  - Created performance graphs and wrote section 3 of report.
  - Updated section 2 of report with optimizations made.
- Kathryn
  - Implemented binning on cpu
  - Moved binning to gpu kernels (defined kernels for bin counting and placing particles in bins), implemented prefix sums on gpu using thrust
  - Implemented mutual force function and application
  - Wrote report section on binning design choices, gpu synchronizations, and optimizations

2 Method

2.1 Binning

By dividing the 2D simulation domain into bins of size `cutoff`, each particle only needs to check for neighbors in a small set of adjacent bins. This significantly reduces the number of force calculations. However, we could no longer use vectors as we did in HW 2-1. Therefore, we used multiple arrays to ultimately sort the particles by their bins into a contiguous array. Below we discuss how we did this:

Data Structures

1. `int* bin_starts_gpu`: length = number of bins + 1, stores the bin counts initially, and then offsets of all bins after prefix sum
2. `int *bins`: length = number of particles, a helper array that the bins that the particle at index j belongs to.
3. `int* binned_particles_gpu`: length = number of particles, stores the particle indices in bin order

Kernels

1. `__global__ void zero_accels`: Set accelerations of all particles to 0 at the beginning of each step simulation
2. `__global__ void init_parts`: Initialize the list part_arr such that part_arr[j] = j.

3. `__global__ void compute_bin_counts`: For each particle, computes the bin the particle is in and sets bins[j] = bin (where j is the index of the particle in the particles array) and increments bin_counts[bin] by one.

4. `__global__ void compute_forces_gpu`: Computes the forces on each particle

5. `__global__ void move_gpu`: Moves all particles

GPU Synchronization

- Atomic operations (e.g., `atomicAdd`) to increment bin counts and to accumulate forces on each particle. Since multiple threads might update the same bin or the same particle's force, atomic operations ensure correct updates without race conditions.

- Thrust's exclusive_scan to compute prefix sums. We use Thrust to do the parallel prefix sum. This call is a synchronization point, since we have to wait for it to finish before binning and force calculations.

- Thrust's sort_by_key to sort the particle indices by the bin the particle belongs to. This call is also a synchronization point, since it must complete before forces are computed.

- Kernel boundaries serve as synchronization points. When one kernel finishes, all threads in that kernel are done before the next kernel starts.

Because we're just using a single CPU core and a single CPU thread, there aren't any other CPU-based synchronizations we considered.

Kernel Launching
We launched all of the kernels with `NUM_THREADS = 1024`. For the kernels `__global__ void zero_accels` and `__global__ void init_parts`, we used `blks = numSMs*BLOCKS_PER_SM`, where `numSMs` is the number of streaming multiprocessors belonging to the device, and `BLOCKS_PER_SM=32`. For the kernels `__global__ void compute_bin_counts`, `__global__ void compute_forces_gpu`, and `__global__ void move_gpu`, we use `blks = (num_parts + NUM_THREADS - 1) / NUM_THREADS`.

Each kernel defines `tid = threadIdx.x + blockIdx.x blockDim.x`. In the `compute_bin_counts` function, each thread is responsible for adding 1 to the bin of the particle at index `threadIdx` in `particles`, and for setting the value of `bins` an the index `threadIdx` equal to the bin index of the particle at index `threadIdx` in `particles`. Similarly, `compute_forces_gpu` and `move_gpu` use an "owner computes" paradigm as well, where they compute the forces upon its particle and move the particle at `threadIdx`. `__global__ void zero_accels` and `__global__ void init_parts` follow a similar approach, but threads are also responsible for particles an indices which are distance that is a multiple of `global_threads` away from `threadIdx`, where `global_threads = numSMs*BLOCKS_PER_SM*NUM_THREADS`.

Optimizations

The size of each bin is the variable `bin_size`, which we set to be equal to the `cutoff` distance so all of the force interactions happen within a particle's bin or a neighboring bin.

Memory Allocation
We initially allocated memory for each of these arrays at the beginning of our `simulate_one_step` function and freed the memory at the end of the function. However, there was a clear optimization. We instead used `cudaMalloc` only once in `init_simulation` and then `memSet` to clear the `bin_starts_gpu` array at the start of each `simulate_one_step` call. This led to a small speedup.

Mutual Force Application
Instead of doing an owner computes paradigm, we switched to mutual force application where the `apply_force` function increments the accelerations of both input particles instead of just the first particle as in the starter code. This requires some care in making sure that forces aren't applied more than once. In order to do this, a particle at `(x, y)` only applies the mutual force with particles in

the bins `(x+1, y), (x, y+1), (x+1, y+1), (x-1, y+1)`. It also applies the mutual force upon particles with index j in its own bin where `tid < j`. This led to a 16.67 % speedup for 10 million particles.

Local Force Accumulation
Because of the "owner computes" paradigm used int the `compute_forces_gpu` kernel, forces from all particles on the owned particles can be accumulated locally before an atomic add is done to the particles stored in global memory. This reduces the number of atomic add and global memory accesses required, and led to a moderate speedup.

Reducing Number of Blocks for Some Kernels
As described earlier, the kernels `__global__ void zero_accels` and `__global__ void init_parts` use a fewer number of blocks than the others. This is because we noticed that reducing the number of blocks and thus the number of threads led to improved performance because of the reduced thread creation/destruction time. While this requires that some threads take on greater responsibility, we found empirically that this tradeoff was worth it.

Increasing Threads Per Block
For all kernels, we found that increasing the number of threads per block from 256 to 1024 improved performance slightly.

3 Results and Discussion

3.1 Linear Time Complexity

1 illustrates the performance of the provided starter code versus our optimized CUDA implementation for various problem sizes. The dotted black line represents linear scaling behavior for reference.

As shown in the figure, the starter code begins to demonstrate quadratic runtime growth as the number of particles in the simulation exceeds 100,000. On the other hand, our optimized CUDA code appears to continue to scale linearly beyond 100,000 thousand particles.

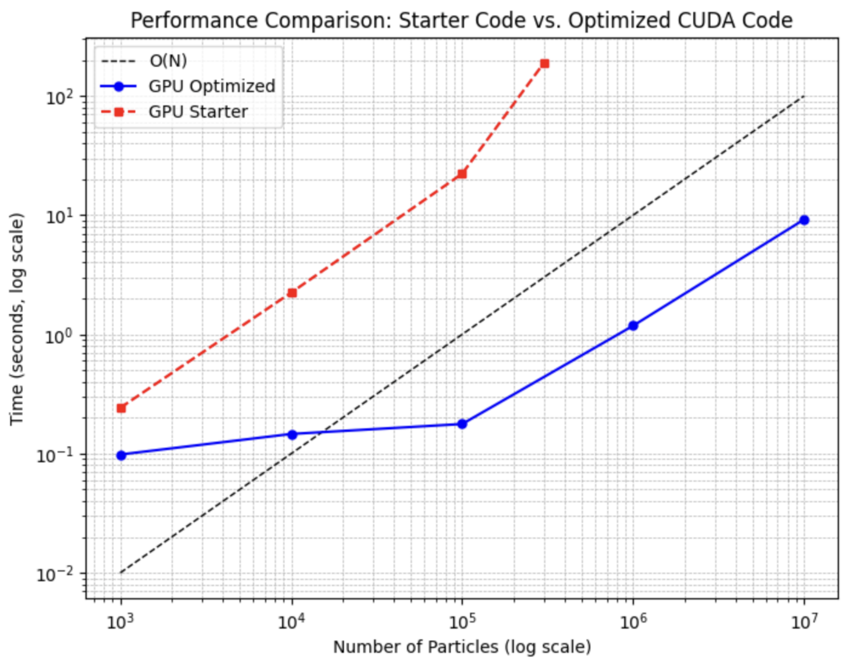

Figure 1: Performance plot of starter vs. optimized CUDA code

3.2 Comparison with MPI/OpenMP Implementation

2 compares the performance of the different parallel architectures used to parallelize the particle simulation. Our CUDA implementation code seems to to perform favorably for the largest problem sizes. However, for smaller problem sizes, the CUDA implementation is slower.

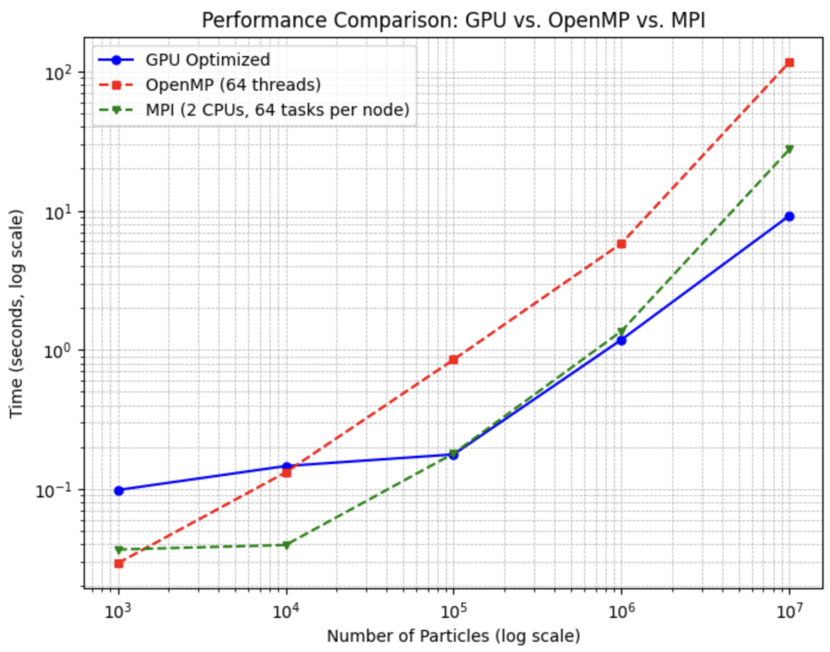

Figure 2: Performance comparison of OpenMP, MPI, and CUDA implementations of particle simulation

3.3 Where Does the Time Goes

3 breaks down the runtime across the various CUDA kernels needed to run our particle simulation for various problem sizes. Notably, the computation of forces between particles and the counting of particles per bin each time a simulation step is run seems to dominate the runtime as the problem size grows large. We hypothesize that this is due to the increasing number of global memory accesses required to execute these approaches as the problem size grows.

3.4 Future Optimizations

The key optimization that we were unable to make successfully was leveraging shared memory effectively. As mentioned above, we hypothesize that the reason for the force computation dominating runtime for large problem sizes relates to the fact that the information (xy position) of neighboring particles has to be accessed from global memory each time you want to compute the force exerted by that particle on the particle a thread is assigned to. This leads to lots of repetitive global memory accesses, wherein the same particle is accesses from global memory by many different threads. Ideally, there should be a way to efficiently load in relevant particles into a block's shared memory so that each particle whose information is needed by a group of threads in a block is only read from global memory once.

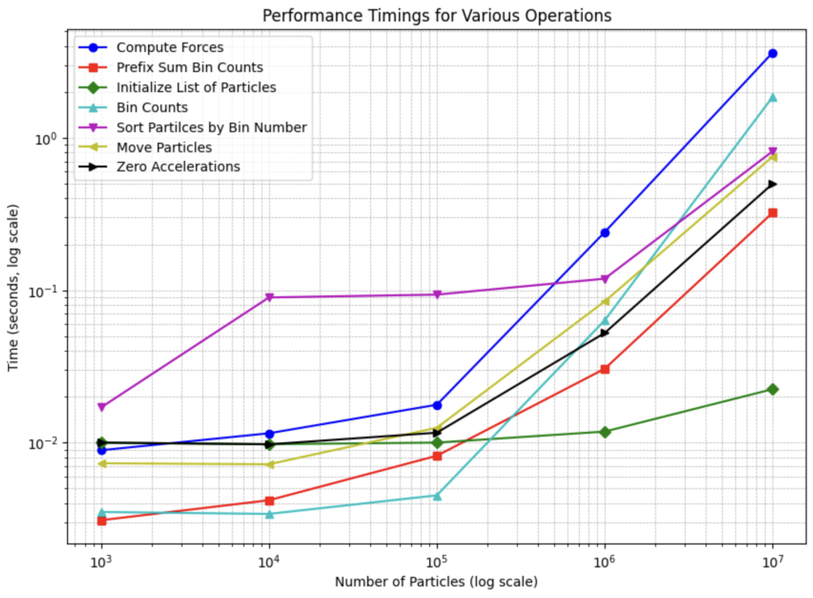

Figure 3: Breakdown of runtime amongs various CUDA kernels