CS267 HW2-3: Parallelizing a Particle Simulation

Grace Wei, Yinjun Zheng, Emin Burak Onat (equally contributed)

Spring 2023

1. Introduction

In this assignment, we implemented an algorithm to parallelize a particle simulation where particles interact through repulsive forces. The asymptotic complexity of the naive approach would be *O(n^2)*. We have improved the code to run in *O(n)* time on a single processor by the same way as in HW2-1. Then we further parallelize the code to reach *O(n/p)* time complexity when using *p* processors through a GPU.

2. Algorithm Description

2.1 Algorithm:

The algorithm we use for achieving *O(n)* time complexity is the same as in HW2-1 and HW2-2. We divide the 2D space into square bins with dimensions `***BIN_SIZE***`. Each particle is then assigned to a particular bin based on its position within the space. The spatial grid is composed of `***dim***` number of bins in one dimension, and the overall length of that dimension is defined by the value `***size***`. To compute forces on particles of one bin, we only consider particles in that bin and in the neighboring eight bins. We ignore the forces from far away particles.

To parallelize the code to achieve *O(n/p)* time complexity, we further allocate the bins to different threads, with each thread having its own memory. Each thread is responsible for one bin.

Overall our algorithm works in the following way:

1. Initialize the bins
2. Copy the particles array from host to device using cudaMemcpy(, , , cudaMemcpyHostToDevice)
3. For t = 1:T do (in GPU)
    a. Atomically assign particles to bins
    b. Compute forces within bins
    c. Move particles
4. Copy the particles array back to CPU from GPU using cudaMemcpy(, , , cudaMemcpyDeviceToHost)

Initializing the Simulation:

We initialize the simulation by allocating the memory on the GPU for the bins array and the bin count array using the cudaMalloc function. The bins_array is an integer array of length num_parts, and the bin_count is an integer array of length dim*dim+1.

Simulating One Timestep:

At each step the `simulate_one_step` function resets the bin_count array to zero using the thrust::fill function. Then, it counts the number of particles in each bin by calling the count_part_per_bin_gpu kernel. This kernel assigns each particle to a bin based on its position, and increments the bin's count.

Next, the thrust::inclusive_scan function performs a prefix sum on the bin_count array, which is used to determine the starting index for each bin's particles in the bins_array array.

Then, the add_particle_to_bin_gpu kernel assigns each particle to its corresponding bin in the bins_array array. The tar_bin_id variable is the index in bins_array where the particle should be stored, and is obtained by using atomicAdd to increment the count for the bin and get its new value. Here we used atomicAdd to ensure that multiple threads accessing the same bin_count index do not cause race conditions.

After the particles are assigned to their bins, the compute_forces_gpu kernel calculates the forces acting on each particle based on its position and the positions of nearby particles in neighboring bins.

Finally, the move_gpu kernel updates the position of each particle based on its current position and velocity.

2.2 Data Structure:

(1) bins_array: We divide the 2D space into square bins of size `***BIN_SIZE***`. We are using the data structure *array<int particle_id>* to represent the bins. Array bins[i] represents the i-th bin, and inside the array bins[i] are the particles that are located in the i-th bin. We are using arrays instead of vectors as in HW2-2 because CUDA doesn't have std::vector data type, we use arrays to store particles, which works fast on GPUs.

(2) bin_count: bin_count stores the number of particles in each bin. We are using the data structure *array<int num_parts>* to represent the bin_count.

3. Synchronization in GPU Implementation

The simulation runs for the predefined number of steps, which is 1000 in our case. Synchronization is done between each time step and once after the 1000 loop iterations. The synchronization is done using cudaDeviceSynchronize() function call. There are two calls to this function in the code:

1. After the simulate_one_step() function is called in the for loop, the cudaDeviceSynchronize() function is called. This ensures that all threads executing on the GPU complete their execution before the host code continues.
2. After the for loop, cudaDeviceSynchronize() is called again to ensure that all CUDA operations are complete before the end time is calculated.

4. Design choices

4.1 one direction force vs. bi-direction force

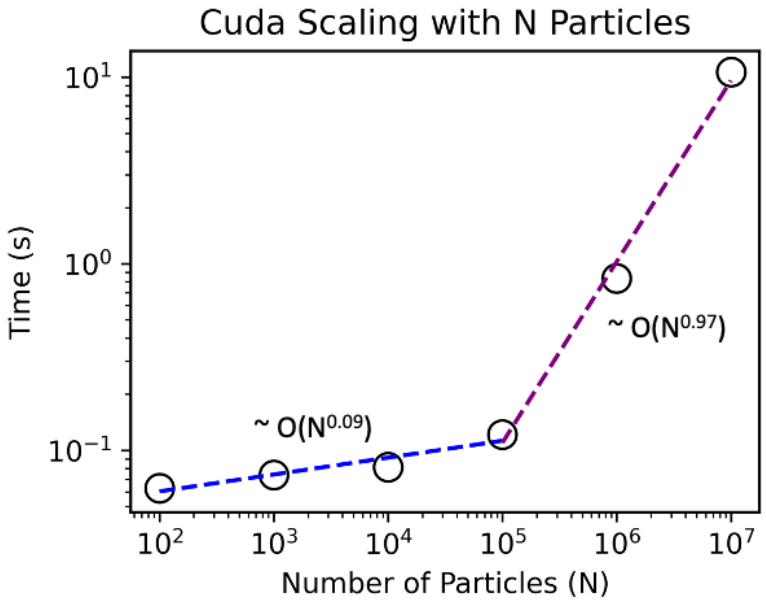
One direction force is the original apply_force function that only updates the attribute for the current particle, so we need to loop over all the eight directions to calculate the force between the current particle and its neighbor. Besides that, we have also tried the bi-directional force, which is to update the attribute for both the current particle and its neighbor. In this way, we only need to loop over half of the eight directions: the up, up right, right down directions to apply force between the current particle and the neighbor particle. We find that the bi-direction force method is similarly fast as the one-direction force method.

5. Log-Log Scale Plots

5.1 Log-Log Plot (running time vs number of particle)

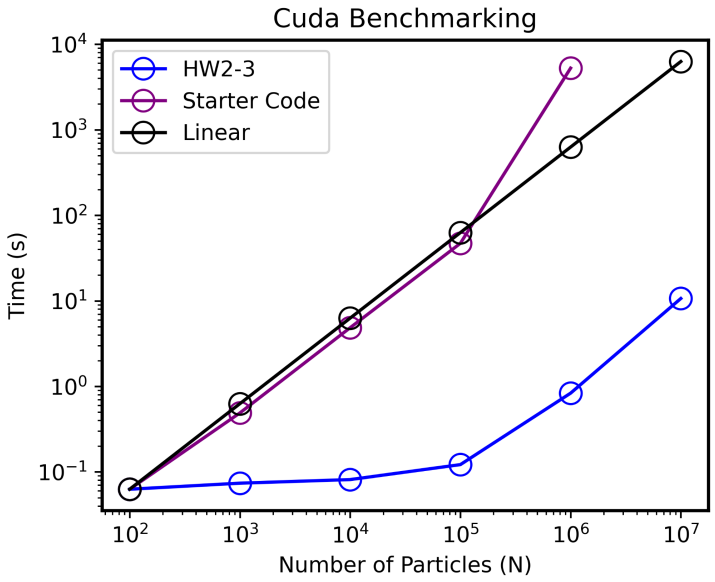
We increase the number of particles and compare the running time in the plot below. We observe that at a lower number of particles (< 10^5), the computational time scales as O(N^0.09), the best scaling of any program we've written in this class thus far. After N exceeds 10^5, the scaling becomes close to linear. This is because a GPU executes many-thread parallelism at once, so as long as the number of particles/bins does not exceed the number of threads, the performance will be roughly the same. The slight increase in computational time is due more or less to synchronization. After the number

of particles (and therefore bins) exceeds the number of threads, then we start to observe linear scaling.



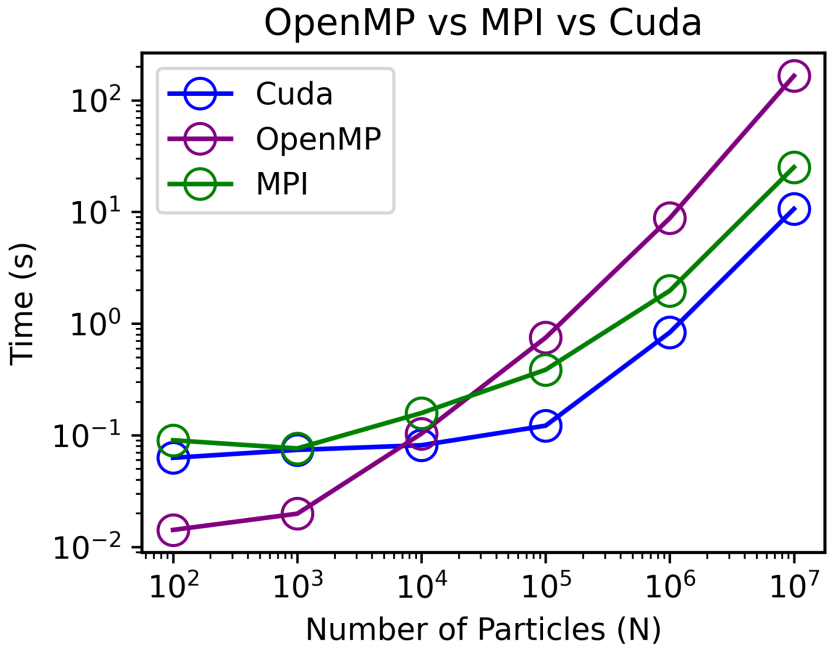
5.2 Cuda Benchmarking

Below, we benchmark our code against the starter code and compare both with the linear behavior. The starter code achieves linear behavior when N < 10^5, but the scaling derails when N > 10^5. Running N=10^6 (1 million particles) on the starter code took ~ 2500 seconds, over 40 minutes! On the other hand, our code performs significantly better than linear behavior when N < 10^5, and approaches linearity when N > 10^5. Our code computes N = 1 million particles (N = 10^6) in under a minute, at 0.83 seconds, achieving over 3000x speedup from the starter code.



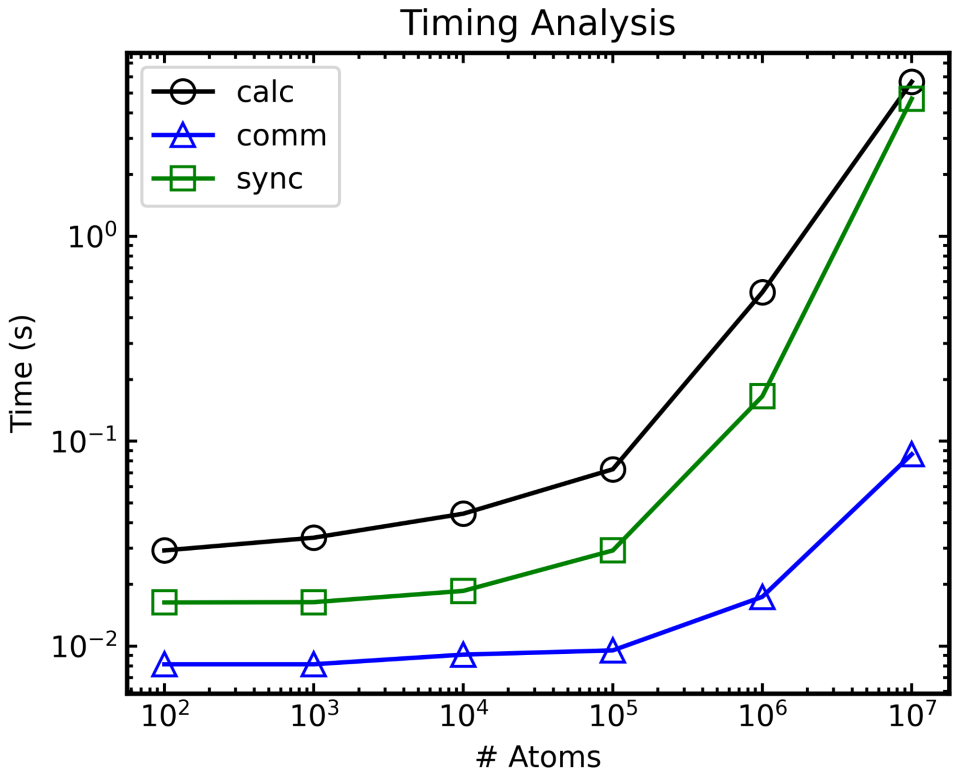
5.3 OpenMP vs MPI vs Cuda

We also compare our Cuda implementation with the OpenMP and MPI implementations from previous parts of HW2. We observe that at very small particle number (N <= 1000), the OpenMP implementation has the highest performance. When 1000>N>100,000, the Cuda implementation has the highest performance and best scaling. When N >= 1 million particles, all implementations achieve similar scaling, with MPI having the best scaling but CUDA having the best overall performance.



6. Timing Analysis

Below we plot the approximate time breakdown of the program spent on calculation, communication, and synchronization for various numbers of atoms. We do this by using std::chrono::steady_clock::now() to mark the start and end times of methods that focus on calculation, communication, and synchronization, and use std::chrono::duration<double> to mark the difference between the start and end times. We keep a global running sum of the time spent in each method, making sure to call cudaDeviceSynchronize() before calling the end time. The count_part_per_bin_gpu() and add_particle_to_bin_gpu() were “synchronize” methods, because they primarily consisted of an atomicAdd that is the bottleneck for synchronization. The cudaMemcpy call was the communication call. The compute_forces_gpu() and move_gpu() methods are the calculation methods.



From the plot, we observe that overall, calculation costs are the most time consuming portion. At a higher number of atoms (> 10^5), we begin to see the synchronization time approach the calculation time. At a higher number of atoms, we also see the communication costs begin to increase, although not as much as synchronization. The scaling of calculation and synchronization is roughly linear for a high number of atoms, whereas communication goes around log(N).