Parallelizing a Particle Simulation - CUDA
CS 267 HW2-3 – Spring 2020

Group10: Xiyu Zhai, Ziqian Qin

Contents

1 Introduction

In this project, we will simulate a simple particle system that particles are subject to Newton's law of motion. The system has a fixed square boundary and a fixed number of equal-weighted particles. The particles repel one another, but only when closer than a cutoff distance highlighted around one particle, as shown in figure 1. When calculating the force between particles, there is a minimum distance limitation between the particles to prevent the extremely large acceleration.

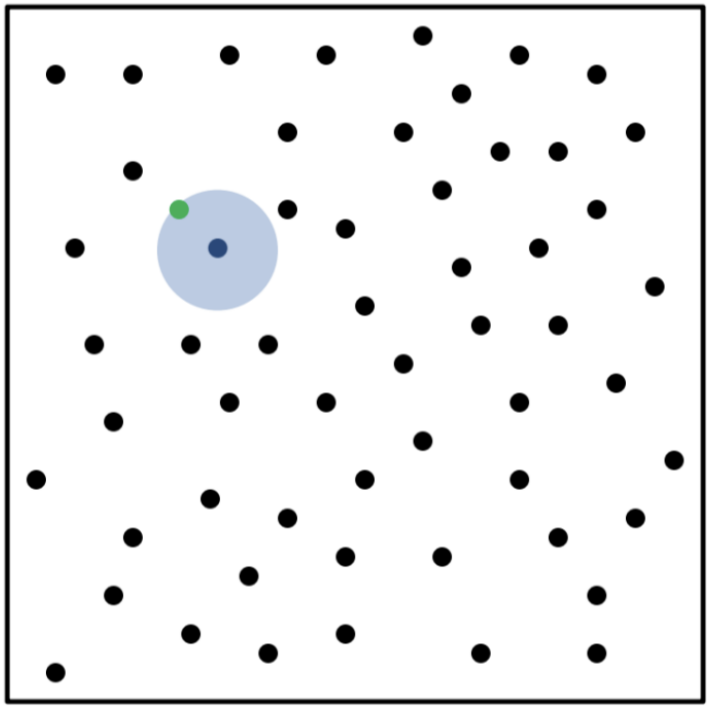

Figure 1: The particle system

The time complexity of naively computing the forces by iterating through every pair of particles is $O(N^2)$, where N is the number of particles.

We will still use the binning algorithm. The main idea is to ignore all far-field interactions by splitting the whole simulation area into several bins. As shown in figure 2, for each bin, we only calculate the force between the particles in this bin and particles in its eight neighboring bins. In this way, assuming there are d particles in each bin on average, the time complexity will be $O(9d \times n) = O(n)$.

In this report, we inspired by the two methods of Simon Green (1) to implement the parallel speedup using CUDA and improve the efficiency of the parallel algorithm step by step. Ideally, we would like to make the time complexity be $O(\frac{N}{P})$ when using P processors.

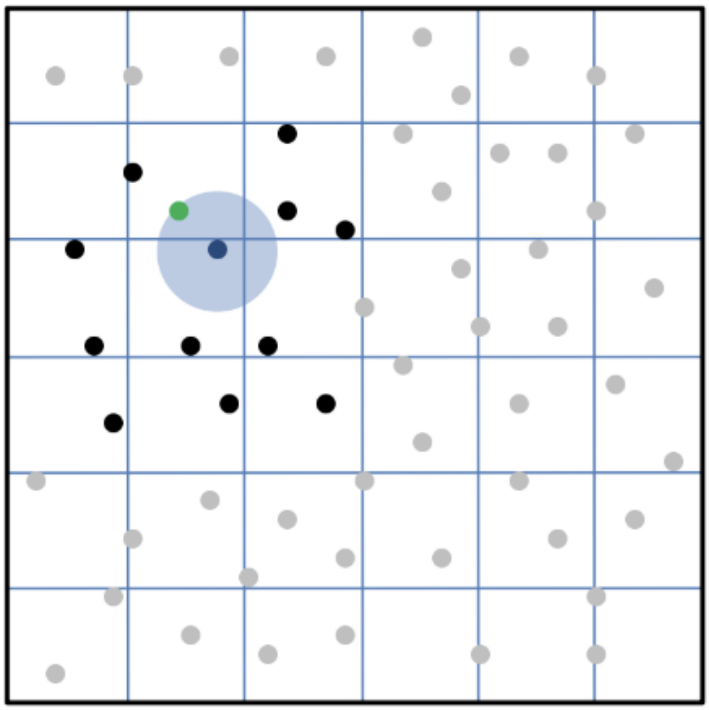

Figure 2: Binning the particle system

2 Contribution

Xiyu Zhai: implementation, test, analysis, documentation

Ziqian Qin: debug, optimization, test, documentation

3 Linked List Solution

Our goal of the first implementation is to have a simple correct version of the CUDA parallel solution, and we cared less about efficiency at this time. We chose to use the same idea we used in the OpenMP solution which is to recalculate all particles' bin number.

3.1 Motivation

As CUDA has unified memory, the easier way is still to reassign particles to bins every simulation step like using OpenMP. Following the given code, we assign particles to different threads. Each threads will only calculate the assigned particle's movement. After simulating one step, we will update all particle information and reassign particles to bins. Intuitively, we decided to use linked list to organize particles.

Once the communication part is completed, each processor will clear all of its bins and then put all received particles to the corresponding bins to prepare for the next simulation step.

3.2 Implementation Details

3.2.1 Assign particles to bins

As it said in the introduction, we still build the box-shaped bins. We then need to discuss how to implement linked list structure. We decided to simply use two arrays 'heads' and 'Llist' to store all particles' id in the same bin.

Each element in array 'heads' is used to store the first particle id in the each bin. The element in array 'Llist' is the id of the next particle in the same bin. The element will be '-1' if it is the last particle in the bin.

```
int* heads;
int* Llist;
cudaMalloc((void**)&heads, binCount * sizeof(int));
cudaMalloc((void**)&Llist, num_parts * sizeof(int));
```

To implement synchronization in CUDA, we use atomic operations to ensure the read and write protection of memory shared among multiple parallel threads. This way only one thread can read and write to the 'Llist'. We use 'atomicExch' to put the new current particle id in 'heads' array and then put the old particle id back to Llist.

The code of the reassigning process is shown below.

```
__global__ void rebin(particle_t* particles, int num_parts,
    int binPerRow, int binCount, double binSize, int* heads, int* Llist) {
// Get thread (particle) ID
int tid = threadIdx.x + blockIdx.x * blockDim.x;
if (tid >= num_parts)
    return;
if (tid < binCount) {
    heads[tid] = -1; //reset heads to -1
}

int thisBin = calcBin(particles[tid], binPerRow, binSize);
Llist[tid] = atomicExch(&heads[thisBin], tid);
}
```

3.2.2 Compute forces and move particles

The implementation of computing forces and moving particles is the same as the OpenMP solution in the last homework. Only after calculating all forces, we will then move particles, and one step cannot start before the previous step finishes. For each particle, we will only compute it with particles in its own bin and neighbor bins.

3.3 Benchmark and Analysis

In this section, we analyzed the performance of the CUDA parallel code method with MPI and OpenMP parallel methods.

3.3.1 Scalability in the Number of Particles

To measure the scalability in the number of particles, we had experimented using 256 threads on Bridge. The log-log scaled figure of the relationship between particle number and simulation time is shown in figure 3. The range of particle number is $[\log_2 10,000, \log_2 2,560,000]$.

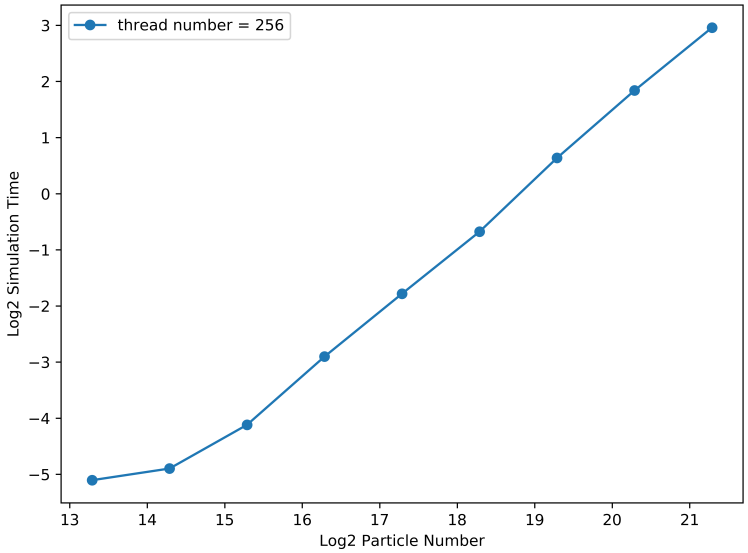

Figure 3: Simulation Time for Different Numbers of Particles

From figure 3, we can tell the simulation time and the particle number are directly proportional when the number of threads remains the same.

3.4 Comparing with other methods

We compared the performance of our code with the naive CUDA code. As shown in figure 4, our code is about 600X faster than the naive code. For example, our code runs 0.29 seconds simulation time for 160,000 particles, while naive code runs 185.04 seconds

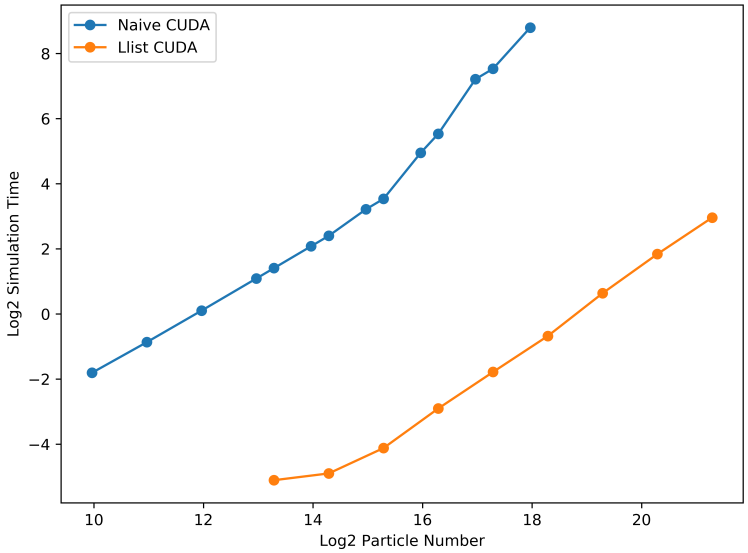

Figure 4: Simulation Time Comparing with Naive CUDA

We compared the performance of our code with the previous OpenMP and MPI solutions. As shown in figure 5, the CUDA solution is the fastest. The OpenMP and MPI solution both run with 68 core on Cori.

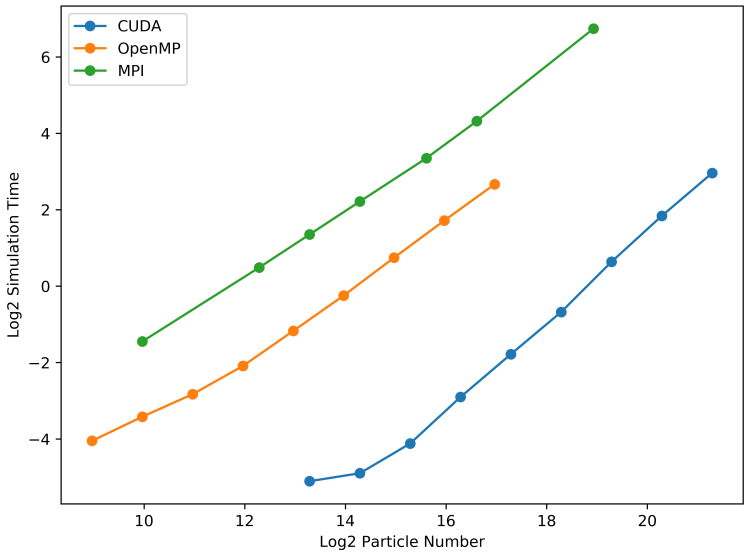

Figure 5: Simulation Time Comparing with OpenMP and MPI

3.5 Time Analysis

If we break the entire time consumption into computation time and synchronization time, we can have a better understanding of our performance.

The computation time and N should be inversely related. This is because particles are evenly distributed among bins, and every thread can get balanced amount of work.

The synchronization time will merely change as N increase, as the we define the bin size as

the $1.3 \times cutoff$, each bin will most likely only have one particles. Thus, the synchronization overhead will not change too much.

References

[1] S. Green, "Particle simulation using cuda," *NVIDIA whitepaper*, vol. 6, pp. 121–128, 2010.