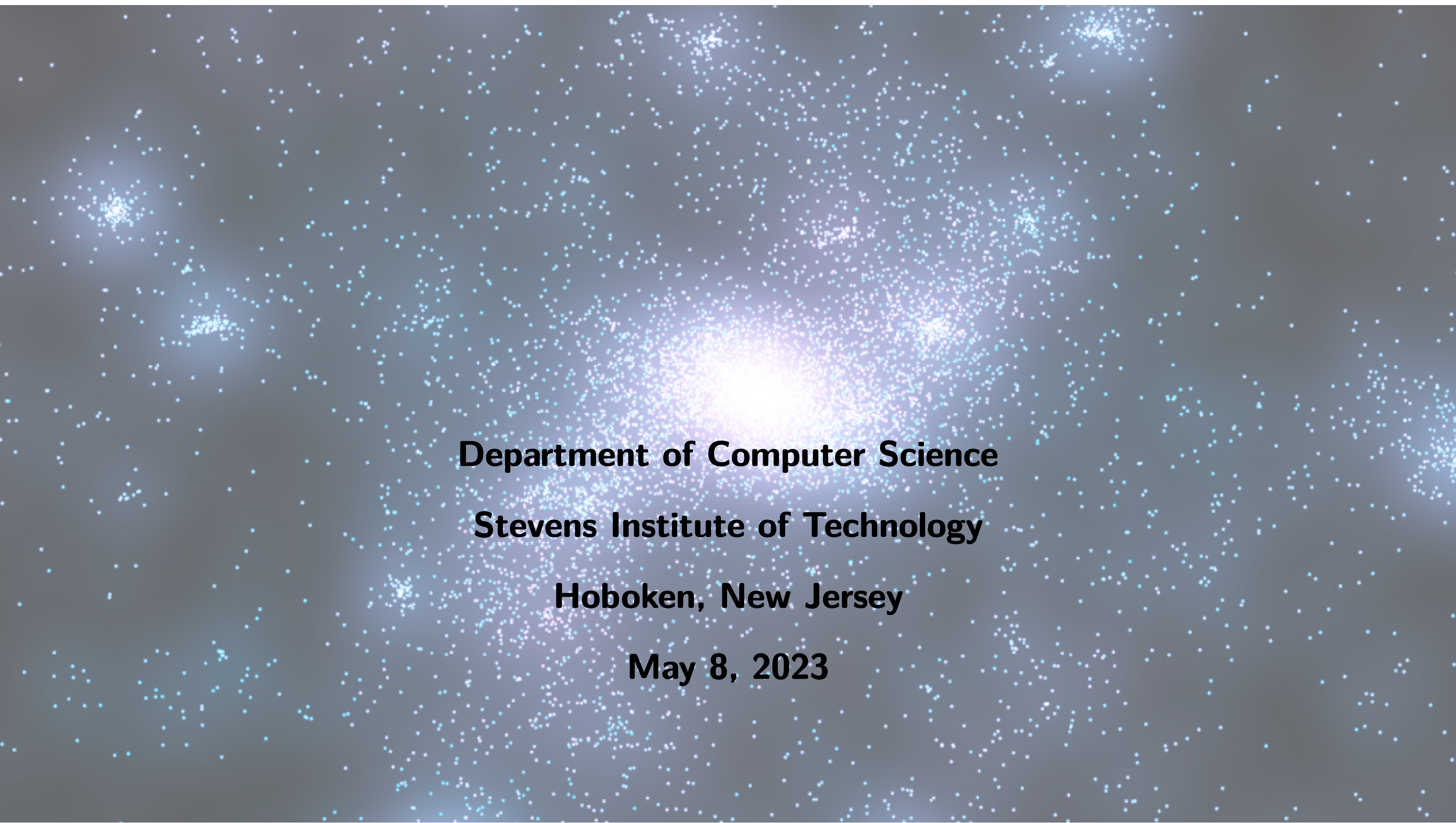
N-Body Simulation

Robert Feliciano

CS677: Final Project Report

Professor Mordohai



Department of Computer Science

Stevens Institute of Technology

Hoboken, New Jersey

May 8, 2023

Contents

1 | Introduction

For my project, I implemented a parallel brute force N-Body Simulation which runs in $O(n^2)$ time. The brute force approach is often times called the "all-pairs N-body simulation" as it simulates the interaction between all the pairs of bodies in the simulation, while other methods, such as the Barnes-Hut simulation, achieve better performance by approximating the interaction by reducing the number of interactions based on the distance between two bodies.

N-Body Simulations can be used to model the interaction between many different things, such as planets, galaxies, and particles. My simulation models the interaction between bodies exerting a gravitational force on one another, with a very large and dense mass being in the middle that also exerts a force on every body in the simulation.

The acceleration of a body due to the gravitational force exerted on it by another body can be modeled by the following equation:

$$a_i \approx G \cdot \sum_{j=1}^{N} \frac{m_j \cdot \vec{r_{ij}}}{(\|\vec{r_{ij}}\|^2 + \epsilon^2)^{\frac{3}{2}}} \tag{1.1}$$

The ϵ is known as a softening factor and is set to 1×10^{-8} in order to prevent division by zero and not have particles collide, but rather pass through each other. My simulation uses this equation to update the velocity of each particle over a small time-step $dt = 0.01$ by multiplying the result a_i by the time-step dt.

I have tested multiple different implementations of the brute-force algorithm. They include sequential CPU code, OpenMP parallelized CPU code, basic CUDA code (this is just a direct port of the sequential CPU code to a CUDA kernel), and optimized CUDA code using shared memory and loop unrolling. Both of the CUDA versions use structures of arrays, `float4`s, and a compiler flag that flushes denormal numbers to zero (more on this later) to achieve the most optimal speed.

Times were recorded in milliseconds using the `high_resolution_clock` from the `<chrono>` library from the `C++` STL. For the CPU implementation, the timer was started before the function that handled the simulation was called and was ended as soon as the function returned. For the CUDA implementation, the timer was started before the kernel was called and was ended as soon as the kernel returned (before memory was copied from the device to the host). The time reported is the amount of time each function/kernel call took averaged over 10 total calls, and the first CUDA kernel launch is ignored since it always takes significantly longer due to the overhead of setting up the device for the first call.

The simulation, timing scripts, checkers, and LaTeXcode for this report in available on GitHub.

2 | Hardware Specifications

The simulation was run locally on my Windows 10 PC using WSL2. The specifications are the following:

- CPU: AMD Ryzen 5 5600X, 6-core, 12-Thread
- GPU: NVIDIA GeForce RTX 3070 Ti

These specifications are important as different CPUs will experience different speed-ups from OpenMP depending on how many threads are available. Along with that, different GPUs will also experience different speeds based on factors such as their streaming multiprocessor count.

3 | Suitability for GPU Acceleration - Amdahl's Law

Since the interaction between one pair of bodies is completely independent from the interaction between another pair of bodies, this problem is embarrassingly parallel. The entire computation can be parallelized on the GPU. Unlike the versions showed in class that updated each body's position vector on the CPU, the implementations discussed in this report have the entire computation on the GPU or in the OpenMP parallelized for-loop. With these implementations, we achieve 100% parallelism for simulating the interaction between all N bodies.

4 | CPU Implementations

The sequential implementation of the brute-force simulation is very simple. There are two for-loops with one nested in the other:

```
for (int i = 0; i < n; i++){
    float fx = 0.0f, fy = 0.0f, fz = 0.0f;
    for (int j = 0; j < n; j++){
        float dx = b[j].x - b[i].x;
        // code continues...
```

This sequential version of the code is very slow. Thanks to OpenMP, we can easily parallelize this by putting a `#pragma omp parallel for schedule(dynamic)` directive above the outer loop. This will spawn some number of threads, depending on how many are availble on the CPU, and significantly speed up the simulation. By doing this, each thread calculates approximately `N/num_threads` interactions rather than a single thread calculating every interaction (as it did in the original code).

The code presented in class on May 3rd was very similar to this except I was updating the positions outside of the parallelized loop by using OpenMP's SIMD pragma directive (I simply added `#pragma omp simd` above a loop that iterated over every body and updated each body's position vector). This was because I was experiencing a bug when I would update the positions in parallel on the GPU, so decided to use SIMD for both implementations. After fixing that bug and moving the position update inside the parallelized loop I've noticed a slight increase in performance, but not too significant.

Something interesting I discovered prior to updating the positions in parallel was that compiling the `C++` code with the `-O3` flag to optimize the code as much as possible led to no difference when adding `#pragma omp simd`. Afer some research, I found out the loop I had originally was being automatically vectorized by the compiler due to the `-O3` flag turning on the `-ftree-vectorize` flag. I believe understanding more about how compilers optimize our code, oftentimes even better than any human could, is crucial to being a good programmer.

Please see Section 8 for visual representations on the timing experiments.

5 | CUDA Occupancy

Before discussing the GPU implementations, I feel it is important to discuss the CUDA Occupancy achieved in them. I use a total of 12,288 bytes of shared memory since each block utilizes a shared array of 768 (this is the block size) `float4`s, each having a size of 16 bytes. Using the CUDA Occupancy calculator (CITE LEI MAO HERE) I was able to determine that my implementation is at 100% occupancy.

PUT THE STUFF WITH THE GRAPHS FROM THE SHEET HERE AND STUFF HERE

I orginially was using 1024 threads per block, which increased the shared memory size ber block to be 16,384 bytes. However, using the occupancy calculator I discovered this was only utilizing 67% occupancy. After switching from 1024 threads per block to 768, I noticed a consistent 1 to

2 millisecond speed up when simulating 30,000 bodies. This speed up scaled appropriately while increasing the number of bodies, sometimes saving about a full second or two when N grew to 1 million bodies.

6 | GPU Implementations

ah ah something direct port and then optimized yeah explain flush denorms and stuff

7 | Time Comparisons

yeah a lot of graphs and tables and stuff yeah and yeah explain the times and be like woahhhhhh thats a big time moment

8 | Discussion

what was cool about this brah? what could i do different or add to this and stuff yeah any limits bro?

9 | Conclusion

gpu rules parallelism is great yeah i be balling uh-huh yeah

10 | References