Fall 2023 ME/CS/ECE759 Final Project Report
University of Wisconsin-Madison

A Comparison of Sequential CPU, Parallel CPU, and GPU Implementations of an N-Body Simulation with Virtual-Reality Viewing

Kincannon Wilson

December 13, 2023

Abstract

This project evaluates different approaches to the N-Body problem of simulating the movement of particles in a vacuum due to gravity. The project reports on various strategies one can use to speed up the calculations for this problem, and it demonstrates the resulting performance increases. In particular, the project evaluates the differences between sequential CPU, mult-threaded CPU, and GPU computing for use in the N-Body problem. Finally, the project demonstrates a virtual reality (VR) application deployed on a VR headset that lets a user see the result of the simulation.

Link to Final Project git repo: https://git.doit.wisc.edu/KGWILSON2/759-final

Contents

1 General Information

1. Name: Kincannon Wilson
2. Email: kgwilson2@wisc.edu
3. Home department: Computer Sciences
4. Status: MS Student
5. Teammate: NA
6. I release the ME759 Final Project code as open source and under a BSD3 license for unfettered use of it by any interested party.

2 Problem Statement

2a. Overview

The project involves the simulation of a large number of particles using C++ code running on the Euler cluster at the University of Wisconsin-Madison. The particles exert force on every other particle equal to the following equation:

$$F = G\frac{m_1 m_2}{r^2} \tag{1}$$

where F is the force exerted between particle 1 and particle 2, m_1 is the mass of particle 1, m_2 is the mass of particle 2, and r is the distance between the centers of particle 1 and particle 2.

The results reported in this project begin with the simulation and visualization of individual particles. Beyond this initial method, all experiments will be based on particle densities in regions of uniform sizes. The number of particles in each region for each frame of the simulation is recorded in a large text file, which is then used by the visualization application.

2b. Technologies used

The project makes use of the C++ programming language for its simulation code. When running the simulation, SLURM scripts are used in order to let the Euler cluster schedule the job and allocate the resources necessary to complete the job. OpenMP, a standard parallel programming API, is used for all multi-threaded code running on a CPU. CUDA, a parallel computing platform that enables general-purpose GPU computing, is used to run the simulation using a GPU. Finally, the Unity game engine and scripts in the C# language are used to visualize the results of the simulation and to build an Android application that can run on a Meta Quest 2 VR headset.

2c. Simplifications and assumptions

Because of the complexity of building a fully-fledged particle simulation, the project makes numerous simplifications.

First, the particles are idealized to have a mass of 1, with no size. This simplifies the force equation above and also means that the particles do not collide with each other. Particles can collide with the walls of the cube that contains them, but this collision is simplified so that the position of the particle is 'clipped' in the direction of the wall it hit and its velocity and acceleration in that direction are zeroed out. This clipping means particles cannot leave the specified simulation area, which makes viewing the results easier.

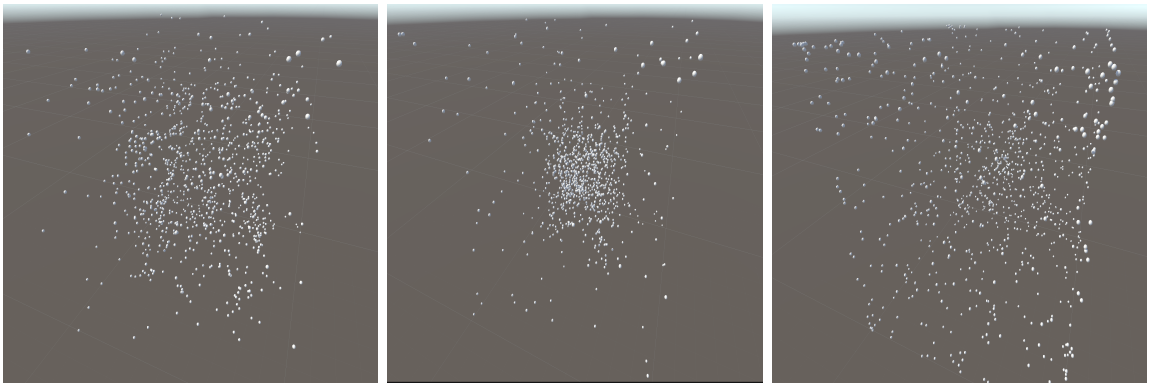
Next, the positions of the particles are uniformly randomly assigned to be within the bounds of the cube container. The velocities and accelerations of the particles are initialized to zero in all 3 directions.

Unless otherwise specified, all code is compiled with the highest level of optimization, and no special tactics (such as loop enrolling) are employed. Finally, timing is performed with the $std :: chrono :: high\_resolution\_clock$ on the CPU and $cudaEvent$ on the GPU, and the reported times **do not** include the time taken to print out the simulation results.

3 Overview of Results. Demonstration of Project

3a. Simulation of individual particles The initial phase of the project focused on simulating and visualizing individual particles as a foundation for subsequent development. This stage established the capability to track particle positions, velocities, and accelerations for each frame. It involved updating these lists by applying gravitational force between particles, recording particle positions in a text file, and measuring the execution time of specific code segments.

Figure 1: Simulation results for individual 1000 particles

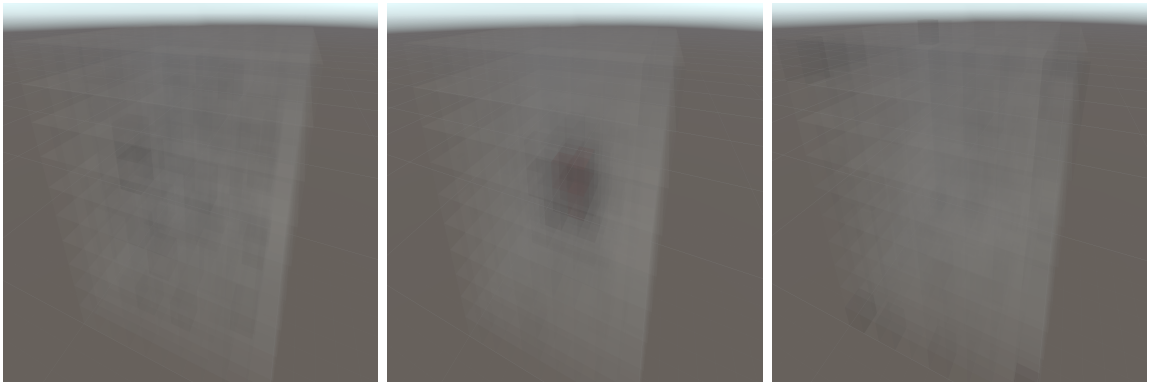


Individual particles are visualized as white spheres. The images above are arranged from earliest frame on the left to latest on the right and show a pattern of contraction and subsequent expansion.

3b. Simulation of particle densities

The next step of the project involved the creation of a system for visualizing the density of particles in each region of the cube container. This was necessary because the application was incapable of rendering unique game objects for each particle when the number of particles was beyond around 10,000 objects. The figure below shows the results of a visualization generated with 1000 regions:

Figure 2: Simulation results for 1000 regions



Regions are visualized as transparent gray squares that become more red and opaque the more particles are in that region during that frame. The images above visualize the same particle positions as in Figure 1 at roughly the same time.

3c. OpenMP

Subsequently, the project delves into leveraging OpenMP to enhance the simulation's speed. In each instance, 8 threads were employed. While the main task in the simulation code involves the loop for updating frames, individual iterations are not independent since the outcome of the next frame relies on the positions, velocities, and accelerations updated in the previous frame. Consequently, the main loop is not an ideal candidate for parallelization using OpenMP. However, other segments of the code don't face this constraint and can be successfully parallelized. For example, the loop responsible for correcting particles' positions if they would have intersected through the box in the current frame can be parallelized as follows:

```
void handle_collisions(float* pos, float* vel, float* acc, int num_particles) {
    #pragma omp parallel for
    for (int i = 0; i < num_particles; ++i) {
        // Check for positions in each dim greater than the box dims and clip them
    }
}
```

The above code demonstrates the appeal of OpenMP as a simple and flexible interface for writing and running parallel code. The example below shows a slightly more nuanced application of OpenMP that uses a reduction clause with multiple accumulators:

```
#pragma omp parallel for
for (int i = 0; i < num_particles; ++i) {
    float acc_x = 0.0f;
    float acc_y = 0.0f;
    float acc_z = 0.0f;

    #pragma omp parallel for reduction(+:acc_x, acc_y, acc_z)
    for (int j = 0; j < num_particles; ++j) {
        if (i != j) {
            float dx = positions[j * 3 + 0] - positions[i * 3 + 0];
            float dy = positions[j * 3 + 1] - positions[i * 3 + 1];
            float dz = positions[j * 3 + 2] - positions[i * 3 + 2];
            float r = std::sqrt(dx * dx + dy * dy + dz * dz);

            if (r > 0 && !std::isnan(r)) {
                float force = (G * particle_mass * particle_mass) / (r * r);
                acc_x += force * (dx / r);
                acc_y += force * (dy / r);
                acc_z += force * (dz / r);
            }
        }
    }

    accelerations[i * 3 + 0] = acc_x;
    accelerations[i * 3 + 1] = acc_y;
    accelerations[i * 3 + 2] = acc_z;
```

The reduction clause is used in the inner loop to perform a reduction on the accumulator variables. This ensures that the updates to these variables are done safely and avoid data races wherein multiple threads attempt to update the same location in memory and cause incorrect results. Other valid approaches would have been to use atomic operations or critical regions.

3d. CUDA

The parallel CUDA code for the project looks much the same as the OpenMP code above with a few modifications. First, the relevant functions were converted into CUDA kernels using the `__global__` specifier like this:

```
__global__ void handle_collisions_kernel(float* positions,
float* velocities, float* accelerations, int num_particles)
```

Then, these kernels are invoked by the host and executed on a device for each frame of the simulation using wrapper functions that look like:

```
void handle_collisions_cuda(float* p, float* v, float* a, int num_particles) {
    int num_threads = 256;
    int num_blocks = (num_particles + num_threads - 1) / num_threads;
    handle_collisions_kernel<<<num_blocks, num_threads>>>(p, v, a, num_particles);
    cudaDeviceSynchronize();
}
```

The above thread and block configurations were chosen to maximize GPU occupancy without requiring too much tweaking. The thread count was chosen because it is a multiple of 32, and the block configuration was chosen in order to ensure that the necessary calculations would be run for each particle with only one partially- occupied block. The use of `cudaDeviceSynchronize()` ensures synchronization between the host and the device, allowing for accurate timing measurements and guaranteeing that kernel execution is completed before proceeding. Two separate kernels, one to update the state vectors and one to handle collisions are used. This is necessary because the kernel that updates the positions, velocities, and accelerations of the particles for a given frame must complete before checking for collisions based on the new predicted positions for that frame. Finally, it should be noted that the arrays were initialized with CUDA's managed memory system, which automatically handles data migration between CPU and GPU, simplifying memory management in hybrid CPU-GPU applications like this one.

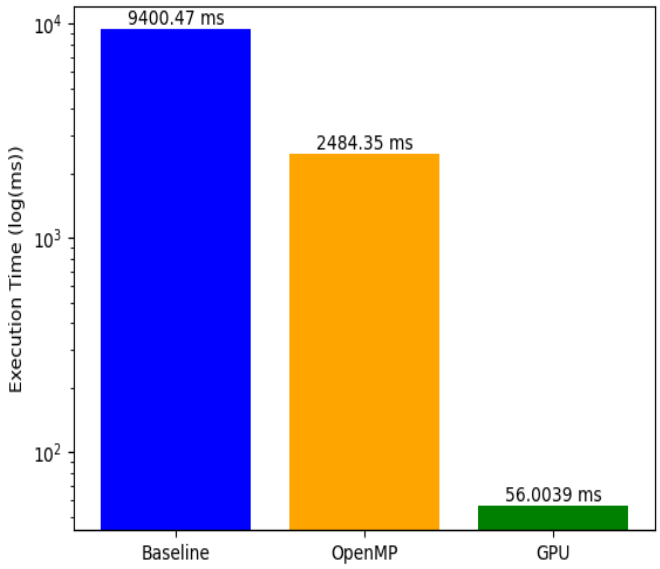
3e. Timing Results

As shown in Figure 3, the OpenMP implementation with 8 threads generated a 3.8x speedup over baseline while the CUDA implementation generated a 167.9x speedup over baseline. In summary, the nature of the N-Body problem seems to have a substantial degree of parallelism, and the CUDA implementation can exploit this parallelism to a much greater extent than the OpenMP implementation.

3f. Visualization of Results

While running, the simulator has the ability to print out the results of the simulation to a text file. Depending on the simulation configuration, it can print out the positions of each particle for every frame of the simulation or the density map, the number of particles in each region, for every frame. Because file I/O happens asynchronously on the Android OS, opening the files required Unity's built-in networking library to make web requests to access the local files. Once the file was loaded, the application looped through the file contents and spawned or modified game objects in the world space at 1 frame per-second. The position visualization script places a small white sphere at the particle's location for that frame. The density visualization script modifies the opacity and color of each transparent cube that represents one region.

Figure 3: Comparison of Execution Time: Baseline vs OpenMP vs CUDA



Timing results for the various implementations of the simulation. In each case, 10,000 particles were simulated for 10 frames. The time taken to print out the results was not included.

4 Deliverables: Building & Running the Project

Included in the project are three SLURM scripts that can be used to launch the three versions of the simulation (baseline, multi-threaded CPU, and GPU) on a computing cluster like UW-Madison's Euler. In addition, the project repository on GitLab includes two .mp4 videos of the simulation when viewed from within a VR headset.

5 Conclusions and Future Work

In summary, this project explores three implementations of a simplified simulation addressing the N-Body problem, showcasing a remarkable 168x speedup with the GPU implementation over the single-threaded CPU baseline. This underscores the substantial benefit the N-Body problem gains from the fine-grained parallelism offered by general-purpose GPU computing.

Given the time constraints of the CS759 course, the project introduced simplifications, opening avenues for potential future enhancements. A scale analysis, evaluating the GPU kernels' performance with varying threads per block as the number of particles increases, would aid in selecting optimal parameters. Similarly, a scale analysis for the OpenMP implementation with different thread counts would provide valuable insights. The OpenMP implementa-

tion's reliance on a preliminary assessment of atomic operations versus the reduction clause suggests room for a more comprehensive analysis of time and memory usage. Profiling tools could offer a deeper understanding of memory-related aspects and other critical performance details beyond clock time. Evaluating additional optimizations like loop unrolling, compiler flags, single-instruction multiple data (SIMD), and other tools for their impact on the simulation is essential. Incorporating features such as inter-particle collisions, precise boundary collisions (instead of clipping), and accommodating particles with diverse sizes and masses would further enrich the simulation. Lastly, the system for displaying particle densities in 3D could benefit from improvements, as the current reliance on transparent regions may introduce a slight lack of clarity. It is probable that a more effective system for visualizing particle densities in 3D has already been developed or could be devised.

Thanks for reading!