Multithreaded N-Body Problem with CPU OpenMP and GPU CUDA Term Project, CSC 746, Fall 2024

Brandon Watanabe* SFSU

E. Wes Bethel† SFSU

Abstract

This project investigates the performance differences between CPU and GPU implementations for solving the 3D N-Body problem, leveraging OpenMP for CPU parallelism and CUDA for GPU acceleration. The N-Body problem, serves as a benchmark for assessing the efficiency of computational architectures. Performance was evaluated across various problem sizes and concurrency levels using metrics such as runtime, memory bandwidth utilization, and GFLOPs/s. While the GPU implementation was expected to outperform the CPU, the results showed that the CPU consistently achieved better performance for most problem sizes, except for the largest tested case, where the GPU demonstrated a runtime advantage. This outcome highlights potential inefficiencies in the GPU implementation, particularly uncoalesced memory accesses. The study provides insights into the computational trade-offs between CPU and GPU approaches and suggests directions for optimizing GPU-based solutions for large-scale physics simulations.

1 Introduction

This project will investigate the performance differences between CPU and GPU computations in solving the 3D N-Body problem, specifically assessing how multithreading can optimize performance on both devices. The N-Body problem, characterized by its time complexity of $O(n^2)$, involves predicting the movements of celestial objects based on gravitational interactions. This time complexity presents a good benchmark for evaluating performance improvements across different computational architectures. The inputs for each body include mass, velocity, and position, while the outputs will be the updated positions and velocities after a specified elapsed time.

To evaluate these programs, we used chrono_timer to measure runtime, percent of memory bandwidth utilized, and average memory latency. From the runtime, we calculated speedup of our parallel runs of our CPU-Only approach. These key metrics should give us a better understanding of how our code performs across various problem sizes and levels of concurrency.

Our results show that at almost every problem size and concurrency level, the CPU code out performed the GPU code. We believe that this is due to the problem sizes not being large enough to take advantage of the GPU. This study seeks to provide insights into the computational trade-offs of these approaches, aiding in the selection of approach for high-performance physics simulations.

2 Related Work

Prior research has established the N-Body problem as a fundamental challenge in computational physics, with various algorithms explored for its solution. Traditional CPU-based implementations often leverage parallelization techniques such as OpenMP. Meanwhile, recent advances in GPU computing, particularly through CUDA, have shown to significantly enhance performance for large-scale parallel tasks. Studies such as those by [4] have demonstrated the effectiveness of GPU implementations using complex algorithms to increase efficiency. Additionally, comprehensive descriptions of different algorithms, including the Naive, Barnes-Hut, and Fast Multipole Methods, can be found in [1], which also discusses multithreading strategies. The article [3] provides an in-depth analysis of the problem setup and the Naive algorithm.

*email:bwatanabe@sfsu.edu

†email:ewbethel@sfsu.edu

3 Implementation

In this section, we will first describe the N-Body Problem. Then we will explore the definition and implementation of each N-Body problem code. We will will also describe how we are storing the data for cache efficiency, then we will dive into how each of the implementations process the forces applied to each body. The CPU approach utilizes OpenMP while the GPU approach utilizes CUDA.

The N-Body problem deals with calculating gravitational forces applied to objects by other objects. To track the properties of each body, we created a struct, shown in Listing 3 that will keep track of the mass, current velocity, and current position. We do not store the forces applied to that body in this struct since this would not lead to contiguous memory accesses. We also do not store the velocity or position history in this struct because we want to keep them as small as possible to improve cache efficiency. After we calculate the forces applied, we will update the velocity and position of each object. There is an optional output mode that allows us to track the velocity and position history which allows us to plot the movement of each object using a python script using Matplotlib.

For each of our implementations, we randomly generated these initial properties of the bodies, then performed the N-Body calculation for a specified length of time. Calculations are made across each timestep throughout the runtime. During our tests, we had the timestep set to one hour and the final time set to 10 Earth years.

```
#define DIM 3
typedef double vect_t[DIM];
struct Body
{
   double mass;
   vect_t velocity;
   vect_t position;
};
```

3.1 The N-Body Problem

The N-Body problem is a classical problem in computational physics and astrophysics, involving the prediction of the motion of a system of bodies under mutual gravitational influence. Each body exerts a gravitational force on every other body, resulting in complex, time-dependent interactions that are challenging to compute as the number of bodies increases. The following is the formula to compute the forces of all bodies that are applied to body i:

$$F_i(t) = \sum_{j=0, j\neq i}^{n-1} G\frac{m_i m_j}{||r_i(t) - r_j(t)||^3}(r_i(t) - r_j(t))$$

where:

- $F_i(t)$ is the gravitational force exerted by all bodies on body i at time t, measured in newtons (N),
- G is the gravitational constant ($6.67430 \times 10^{-11}\,\mathrm{m^3\,kg^{-1}\,s^{-2}}$),
- m_i and m_j are the masses of bodies i and j, respectively, measured in kilograms (kg),
- $r_i(t) - r_j(t)$ is the distance between the two bodies, measured in meters (m), calculated as:

$$||r_i(t) - r_j(t)|| = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2},$$

where (x_i, y_i, z_i) and (x_j, y_j, z_j) are the Cartesian coordinates of bodies i and j, respectively.

Using Newton's Second Law, $F = ma$ we can determine the acceleration and therefore update the velocity and position of each body at each time t.

3.2 N-Body Problem on CPU with OpenMP Parallelism

In the CPU only implementation, we first have a for loop that iterates over the entire simulation time, which jumps by timestep, as shown in Listing 1. This means that for our tests, we calculated the velocities and positions for each body for every hour over 10 years. Inside this for loop, we have a loop that iterates over each body. We have used OpenMP to parallelize this loop such that each thread is assigned a body. Inside this loop, we call the `compute_forces`, as shown in Listing 2. After every thread has finished, we will update the velocities and positions using those newly computed forces. Storing the forces in this function instead of keeping it in the struct also helps with contiguous memory accesses when setting all the forces back to 0 during the next iteration of the timestep loop.

In the function `compute_forces`, as shown in Listing 2, we use a for loop to iterate over every body except the one we are currently computing for and calculate the force applied to it in all 3 dimensions. First, we calculate the distance between the two bodies i and j, then we get the magnitude of that difference and store it in *r_norm*. With these values, we can calculate the force applied in each dimension.

The function `update_bodies`, shown in Listing 3, will update the velocity and position of each body according to the new forces applied to it. This is also where we have the optional history output so that we can track the velocities and positions of each body over time.

As we can see, we have a nested loop structure that iterates over timesteps, then inside that we have a loop that iterates over N bodies. That loop calls a function which also iterates over N bodies. Therefore, we have a time complexity of $O(n^2)$.

3.3 N-Body Problem on the GPU using CUDA

In the CUDA implementation, shown in Listing 4, we are allocating the memory for the bodies and forces for the GPU. We then launch the kernel with a specified number of threads per block and number of blocks. The function that is called (Listing 5 is very similar to the CPU code version. This function is run by each thread and will call the `compute_forces` and `update_bodies` functions for each timestep.

Listing 6 and 7 show the implementations of `compute_forces` and `update_bodies` respectively. They are once again, very similar to the CPU implementation, but they are device functions. Instead of passing a body in as a parameter, we are calculating the index into the bodies array by getting the thread id.

The workload is spread such that each thread gets one body to calculate and update the forces for.

```
9  void do_nBody_calculation(Body* bodies, const int
       N, const int timestep, const unsigned long
       long final_time, const bool record_histories,
       double** velocity_history, double**
       position_history)
10 {
11     int history_index = 1;
12     double* forces = new double[N * DIM];
13     for(int t = 0; t < final_time; t+=timestep)
14         memset(forces, 0, N * DIM * sizeof(double)
       );

16         #pragma parallel omp for
17         for(int i = 0; i < N; i++)
18             double total_force[DIM] = {0.0, 0.0,
       0.0};
19             compute_forces(bodies, bodies[i], N,
       total_force);
20             for (int idx = 0; idx < DIM; idx++)
21                 forces[i * DIM + idx] =
       total_force[idx];

23         update_bodies(bodies, forces, timestep, N,
        record_histories, history_index,
       velocity_history, position_history);
24         history_index++;

26     delete[] forces;
27 }
```

Listing 1: This function will call the compute and update functions for each body for every timestep. The workload is split up by bodies such that each thread will calculate for forces for each body.

4 Evaluation

In this section, we provide a comprehensive overview of the computational platform and software environment used to evaluate the performance of our N-Body problem approaches. We outline the methodology for data collection and analysis, with a focus on key performance metrics such as runtime, and various performance metrics. These metrics are essential for assessing the efficiency of each implementation across different devices. Finally, we present and analyze the results, highlighting key findings.

4.1 Computational platform and Software Environment

We conducted our tests on the Perlmutter system provided by NERSC, which operates on SUSE Linux 12.3.0. Our testing utilized a single GPU-Accelerated Comute Node, equipped with one AMD EPYC 7763 (Milan) CPU. This processors features a base clock speed of 2.45 GHz and can boost up to 3.5 GHz. It includes 64 Zen 3 compute cores, with each core having a 32 KiB write-back data cache and a 512 KiB unified instruction/data L2 cache. Eight cores share a 32 MiB L3 cache, totaling 256 MiB. The CPU also supports AVX2 for 256-bit wide SIMD operations. The node is equipped with 512 GiB of 3200 MHz DDR4 memory, providing a maximum memory bandwidth of 204.8 GB/s per CPU. The system boasts a theoretical peak performance of 39.2 GFLOPs/s per core and 2.51 TFLOPs/s per socket. The AMD EPYC 7763 has 64 cores with 2 threads per core.

Each node has four NVIDIA Ampere A100 GPUs connected together using NVLink-3 and two NICs. The CPU is connected to these via PCIe 4.0. Each NVLink connection supports 25GB/s per directions, leading to a total of 100GB/s between 2 GPUs. The total bandwidth is 600 GB/s. Each GPU is partitioned into 8 GPU Processing Clusters (GPCs). Each GPC contains 8 exture Processing Clusters (TPCs), and each TPC contains 2 Streaming Multiproces-

```
void compute_forces(Body* bodies, Body i, int N,
    double* total_force)
{
    for(int j = 0; j < N; j++)
        if(i == bodies[j]) continue;

        double dx[DIM] = {0.0, 0.0, 0.0};
        double r = 0.0;
        double r_norm = 0.0;

        for (int idx = 0; idx < DIM; idx++)
            dx[idx] = bodies[j].position[idx] - i.
    position[idx];
            r += dx[idx] * dx[idx];

        r_norm = sqrt(r);
        if (r_norm == 0.0) continue;

        for (int idx = 0; idx < DIM; idx++)
            double f = (G * i.mass * bodies[j].
    mass * dx[idx]) / (r_norm * r_norm * r_norm);
            total_force[idx] += f;
}
```

Listing 2: This function will compute the forces applied by each body onto the current body we are iterating for.

```
void update_bodies(Body* bodies, const double*
    forces, const double dt, const int N, const
    bool record_histories, const int history_index
    , double** velocity_history, double**
    position_history)
{
   for (int i = 0; i < N; i++)
      for (int idx = 0; idx < DIM; idx++)
         bodies[i].velocity[idx] += forces[i * DIM
     + idx] / bodies[i].mass * dt;
         bodies[i].position[idx] += bodies[i].
    velocity[idx] * dt;

      if (record_histories)
         for (int idx = 0; idx < DIM; idx++)
            velocity_history[history_index * N + i
    ][idx] = bodies[i].velocity[idx];
            position_history[history_index * N + i
    ][idx] = bodies[i].position[idx];
}
```

Listing 3: This function will update the velocity and position for each body based on the new forces calculated.

sors (SMs), giving each GPU a total of 108 SMs. The A100 GPU is equipped with 12 total memory controllers, 10 being 512-bit memory controllers, for 40 GiB HBM2 at the maximum bandwidth of 1555.2 GB/s. An SM has 192KiB of unified L1 data cache and shared memory. The L2 cache of each GPU is 40 MiB, divided into two partitions for higher bandwidth and lower latency access. Each SM has 64 INT32, 64 FP32, 32 FP64, and 4 Tensor cores, totaling 6912 INT32, 6912 FP32, 3456 FP64, and 432 Tensor cores per GPU. The register file size is 256 KiB per SM. The A100 operates at a clock speed of 1410 MHz.

Theoretical peak performance for the GPU ranges based on the type of operation being done. The lowest value per GPU is 9.7TFLOPs/s while doing FP64 operations, while the highest is 311.9 TFLOPs/s while doing FP16 Tensor operations which can be boosted up to 623.7 TFLOPs/s with Sparsity.

For compiling our code, we utilized the Cray Programming Environment 2.7.30 OS, NVHPC 23.9.0 CXX compiler, and NVIDIA 12.2.91 CUDA compiler. All information was sourced from the NERSC Perlmutter documentation [2] as well as version checking via terminal commands.

4.2 Methodology

To evaluate the performance of our N-Body problem implementations, chrono_timer to measure runtimes, focusing specifically on the execution of the computation regions. We used this runtime to calculate the FLOPs/s, percent memory bandwidth utilization, and speedup. These metrics provide a detailed view of both the computational efficiency and memory behavior of the algorithms.

We measured the performance across various problem sizes $N = [256, 512, 1024, 2048, 16384]$ where the timestep is one hour and the total simulation time is 10 Earth years. For the CPU code, we ran with [1, 4, 16, 64] threads while the CUDA implementation ran with [32, 126, 256] threads per block and the corresponding number of blocks depending on problem size. These concurrency levels were chosen to cover a range of the devices capabilities from little utilization to a larger percent of utilization.

We used the measured runtime for the CPU to calculate the following metrics:

- **MFLOPs/s** (Million Floating Point Operations per Second) is calculated using the formula:

$$\text{MFLOPs/s} = \frac{\text{o}}{\text{t}} \tag{1}$$

  where:

  - o = Number of arithmetic operations
  - t = Runtime (seconds)

- **Percent of Memory Bandwidth Utilization** is defined as:

$$\%\ \text{Bandwidth} = \frac{\frac{\text{b}}{\text{t}}}{\text{p}} \tag{2}$$

  where:

  - b = Number of memory bytes accessed
  - t = Runtime (seconds)
  - p = Theoretical peak memory bandwidth

- **Speedup** is calculated using the formula:

$$\text{S(N, P)} = \frac{T^*(N)}{T(N,P)} \tag{3}$$

  where:

  - $T^*(N)$ = The runtime of the serial implementation
  - $T(N,P)$ = The runtime on size N using P parallel ranks

4.3 Scaling study of CPU/OpenMP code

We conducted a scaling study of our CPU-only implementation of the Sobel filter, measuring runtime across varying levels of concurrency. The results are shown in Figure 1, where we observe that as the number of threads increases, the GFLOPs/s also increases. For 1 thread, we see a peak of 24.45 GFLOPs/s; for 4 threads, we see a peak of 96.56 GFLOPs/s; for 16 threads we see a peak of 361.63 GFLOPs/s; and for 64 threads we see a peak of 1156.85 GFLOPs/s. This increase in GFLOPs/s comes from a reduction in runtime. This occurs because the workload is distributed among more threads,

```
61 void nBody_kernel(Body* bodies, const int N, const
       int timestep, const unsigned long long
     final_time, const bool record_histories,
     double* velocity_history, double*
     position_history, int threads_per_block, int
     num_blocks)
62 {
63     Body* d_bodies;
64     double* forces;
65     gpuErrchk(cudaMalloc(&d_bodies, N * sizeof(Body
       )));
66     gpuErrchk(cudaMemcpy(d_bodies, bodies, N *
       sizeof(Body), cudaMemcpyHostToDevice));
67     cudaMalloc(&forces, N * DIM * sizeof(double));

69     do_nBody_calculation<<<num_blocks,
       threads_per_block>>>(d_bodies, forces, N,
       timestep, final_time, record_histories,
       velocity_history, position_history);

71     gpuErrchk(cudaMemcpy(bodies, d_bodies, N *
       sizeof(Body), cudaMemcpyDeviceToHost));
72     gpuErrchk(cudaFree(d_bodies));
73     gpuErrchk(cudaFree(forces));
74 }
```

Listing 4: Allocates memory for the bodies and forces and launches the kernel.

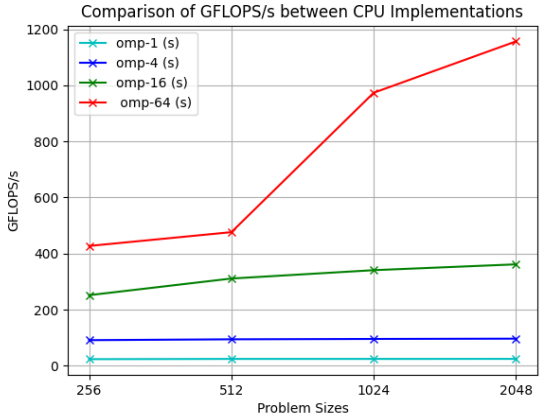

Figure 1: Comparison of GFLOPs/s between differing levels of concurrency for the CPU implementation. With increasing the problem size, we see better performance. We also see an increase when increasing the level of concurrency. The vertical axis indicates GFLOPs/s while the horizontal axis represents the number of bodies.

allowing each thread to process a smaller portion of the data. Using one thread, we reached 62.37% of the theoretical maximum of our system, while at 64 threads we reached just 46.08% of peak. This theoretical peak only takes into account the base clock of 2.45 GHz. These percentages get lower if we were to factor in the boost clock of 3.5GHz.

In Figure 2, we show the speedup achieved at different levels of concurrency, calculated using Equation 3. For 4 threads, we observe a speedup of 3.95; for 16 threads, the speedup increases to 14.79; and 64 threads, we achieve a speedup of 47.31. These results are close to the expected speedup, indicating efficient parallelization.

We can see that our speedup and GFLOPs/s are bound by Amdahl's Law, which states that the speedup gained from adding more threads is limited by the portion of the code that remains serial. In an ideal scenario, each processor rank P would handle of $\frac{N}{P}$ of the workload; however, Amdahl's Law indicates there is a ceiling on speedup in strong scaling. The greater the serial component of the code, the less overall speedup can be achieved. In our CPU implementation, there is very minimal serial portions of the code, therefore we are seeing speedup very close to expected. The only part that is done in serial is the updating of velocities and positions. We tried parallelizing that section as well, but it had a negligible effect on the speedup.

In Figure 3 we can see the percent memory bandwidth utilization of the system. In all of the different levels of concurrency, we see a decline as the problem size increases. To understand the decline, we need to look at how our problem sizes fit into our cache. Each body is 56 bytes. This comes from 8 bytes for mass, and 24 bytes for both velocity and position. Therefore, at 256 bodies we have 20 KiB which easily fits into our L1 cache. At Problem Size 512, 1024, and 2048, the data fits into only L2 and L3 caches. Since accessing the L1 cache is much faster than L2, we see a large decrease in memory bandwidth utilization from N = 256 and N = 512. From there, we see a slower and slower decline in utilization. We only get at most a 3.78% utilization, indicating that we may not be utilizing cache as efficiently as possible.

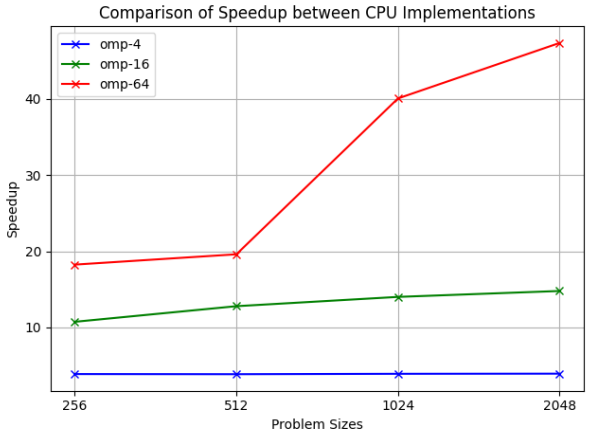

Figure 2: Comparison of speedup for the CPU implementation across varying levels of concurrency. The 4 threads almost reached a 4x speedup, 16 threads didn't get as close to the theoretical speedup, and 64 threads was even further. The vertical axis indicates speedup, and the horizontal axis represents the number of bodies.

```
__global__
void do_nBody_calculation(Body* bodies, double*
     forces, const int N, const int timestep, const
      unsigned long long final_time, const bool
     record_histories, double* velocity_history,
     double* position_history)
{
   int history_index = 1;

   for(int t = 0; t < final_time; t+=timestep)
      int i = blockIdx.x * blockDim.x + threadIdx.
     x;
      if (i < N)
         for (int j = 0; j < DIM; j++)
         {
               forces[i * DIM + j] = 0.0;
         }

      __syncthreads();

      compute_forces(bodies, forces, N);

      update_bodies(bodies, forces, timestep, N,
     record_histories, history_index,
     velocity_history, position_history);

      history_index++;
   }
}
```

Listing 5: This function is on the device. It iterates over each timestep and calls the compute_forces and update_bodies functions.

4.4 GPU-CUDA performance study

This study was to determine the performance differences of different configurations of threads per block and number of blocks when running our CUDA implementation. Figure 4 shows the GFLOPs/s between 32, 128, and 256 threads per block. As wee can see, there is not much difference between 32 and 128 threads per block, while 256 threads per block consistently sees poorer performance. This is likely due to the 256 threads per block using a larger amount of shared memory. Since we are reading from arrays throughout this computation, there is a lot of memory accesses. Additionally, in our implementation in Listing 6, we do not have coalessed memory in the j loop. This is likely the cause of our relatively poor performance across all thread counts and problem sizes. We are seeing only a peak of 153.54 GFLOPs/s while the system is capable of 9.7 TFLOPs/s of FP64 operations. This is only 1.58% of peak.

4.5 Findings and Discussion

Comparing the two codes was challenging as we could not get LIKWID nor ncu to work properly on Perlmutter. However, we do have a comparison in runtime and FLOPs/s. For the runtime comparison (Figure 5, we can see that the CPU code outperformed our GPU code in almost every problem size. The only exception is 1 thread at our 3 largest problem sizes and 4 threads at N = 2048. For every other configuration on this chart, the CPU beats the GPU. We ran one test at a very large problem size N = 16384. Our CPU code used 64 threads while the GPU code used 128 threads per block and 128 thread blocks. This gave us a very different result. The CPU code runtime ended up being 30.63 minutes while the GPU implementation runtime was just 15.23 minutes. At this very large problem size, the GPU code runs in about half the time of the CPU code. This is much different than our results at the smaller problem sizes. This suggests that perhaps the problem sizes chosen for this study favor the CPU code, but if we ran tests at larger problem sizes,

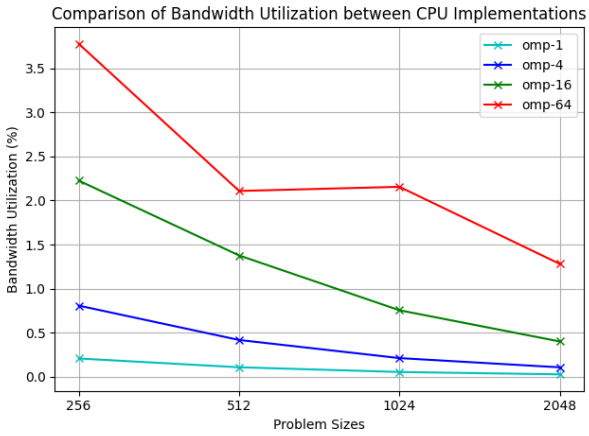

Figure 3: Comparison of percent memory bandwidth utilization CPU implementation across varying levels of concurrency. We see that increasing the number of threads increased our utilization, but increasing the problem size decreased the utilization. The vertical axis indicates percent memory bandwidth utilization while the horizontal axis represents the number of bodies.

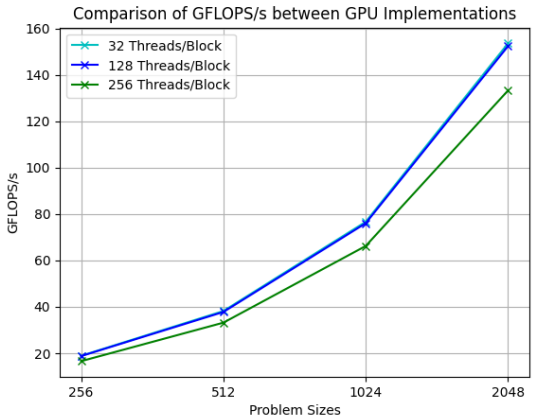

Figure 4: Comparison of GFLOPs/s between differering levels of threads per block. For all problem sizes, the 32 and 128 performed about the same while the 256 saw consistently worse performance. The vertical axis indicates GFLOPs/s while the horizontal axis represents the number of bodies.

```
__device__
void compute_forces(Body* bodies, double* forces,
    int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= N) return;

    double total_force[DIM] = {0.0, 0.0, 0.0};

    for(int j = 0; j < N; j++)
        if(i == j)
            continue;

        double dx[DIM] = {0.0, 0.0, 0.0};
        double r = 0.0;
        double r_norm = 0.0;

        for (int idx = 0; idx < DIM; idx++)
            dx[idx] = bodies[j].position[idx] -
    bodies[i].position[idx];
            r += dx[idx] * dx[idx];

        r_norm = sqrt(r);
        if (r_norm == 0.0)
            continue;

        for (int idx = 0; idx < DIM; idx++)
            double f = (G * bodies[i].mass *
    bodies[j].mass * dx[idx]) / (r_norm * r_norm *
     r_norm);
            total_force[idx] += f;

    for (int idx = 0; idx < DIM; idx++)
        forces[i * DIM + idx] = total_force[idx];
}
```

Listing 6: This function will compute the forces applied by each body onto the current body we are iterating for.

```
__device__
void update_bodies(Body* bodies, const double*
    forces, const double dt, const int N, const
    bool record_histories, const int history_index
    , double* velocity_history, double*
    position_history)
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   if (i >= N) return;

   for (int idx = 0; idx < DIM; idx++)
      bodies[i].velocity[idx] += forces[i * DIM +
    idx] / bodies[i].mass * dt;
      bodies[i].position[idx] += bodies[i].
    velocity[idx] * dt;

   if (record_histories)
      for (int idx = 0; idx < DIM; idx++)
         velocity_history[history_index * N * DIM
    + i * DIM + idx] = bodies[i].velocity[idx];
         position_history[history_index * N * DIM
    + i * DIM + idx] = bodies[i].position[idx];
}
```

Listing 7: This function will update the velocity and position for each body based on the new forces calculated.

perhaps the GPU code would have performed better. Unfortunately, we do not have the time to run tests that long on Perlmutter.

Our results differ from those published in previous studies. Those suggest that the GPU code should out perform the CPU code. Thus, our hypothesis going into this study was that the GPU approach would have had a faster runtime. The discrepancy is likely due to our implementation of the CUDA code. There is a large percent of uncoalessed memory accesses due to our loop structure. To fix this in future iterations, we may consider changing how we distribute the workload. Currently, computes all the forces for each body. A better way to approach this may be to have each thread calculate the force applied by one body. This would mean that each thread would have to do less work and we can use more threads total. This should significantly increase our performance in the GPU code.

In conclusion, in most of our problem sizes, the CPU with OpenMP code outperformed our CUDA implementation. When testing a very large problem size, the GPU approach finally beat the CPU results. These results were likely due to a poor implementation of our CUDA code and could likely be improved in the future.

4.6 Acknowledgements

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a Department of Energy Office of Science User Facility using NERSC project m3930 for 2024.

ChatGPT was used to aid in grammar and some LaTeX code in this report.

References

[1] Y. Markovsky, E. Strohmaier, and N. Sundaram. N-body methods. *University of California Berkely Department of Electrical Engineering and Computer Sciences*, 2024.

[2] NERSC. *System Details - NERSC Documentation*, September 2021.

[3] R. Spiteri. Part v - the n-body problem. *University of Saskatchewan Department of Computer Science*, pp. 2–36.

[4] H. Wu. Optimizing n-body simulation with barnes-hut algorithm and cuda. *Medium*, Jan. 2024.

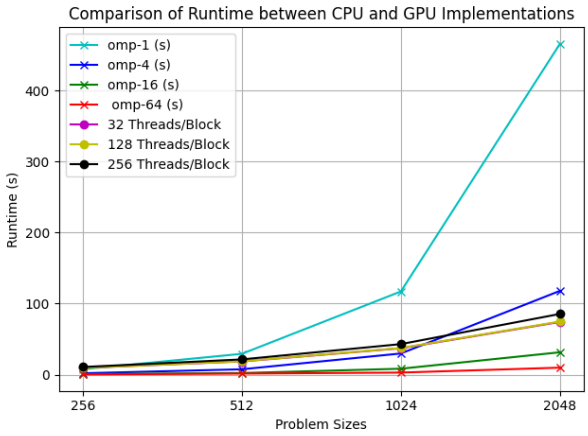

Figure 5: Comparison of runtime for the CPU with OpenMP and CUDA implementations. For most problem sizes, the CPU code outperforms the GPU code. The vertical axis indicates runtime in seconds while the horizontal axis represents the number of bodies.