Evaluating N-Body Simulation Optimizations

Nick Twum

December 20, 2025

1 Introduction

N body simulations have played a vital role to understanding the overwhelming cosmological structures that hold up our universe. To better understand not only how the universe works, but it's origin and future, astrophysicits and cosmologist have attempted to simulate as many bodies as possible often (With some of the largest reaching 4 trillion individual bodies)[1]. New ways are constantly being discovered to optimize these simulations and seeing the incredibly creative optimizations implemented is what drew me to embark on this project!

- **Problem Statement:** Comparing N-Body Simulation Optimization
- **Objectives:** Start with a basic N body Simulation and apply various optimization techniques, multi-threading and SIMD to improve simulation runtime performance

2 Methodology

2.1 Baseline Implementation

2.1.1 Physics Model

The simulation will employ the Brute Force N-body method, which is a direct approach in which the gravitational attraction between each pair of particles is calculated. At a given time step, the net force acting on a particle is computed, and its position and acceleration are updated using Euler's Method for numerical integration as follows:

$$\mathbf{a}_i^n = -G \sum_{\substack{j=1 \\ j \neq i}}^{N} m_j \frac{\mathbf{r}_j^n - \mathbf{r}_i^n}{\left(\|\mathbf{r}_j^n - \mathbf{r}_i^n\|^2 + \epsilon^2 \right)^{3/2}}$$

The equation shows the acceleration a particle experiences is the cumulative effect of the gravitational force from all other particles which is important to note when employing SIMD optimizations.
We update our velocity and position like below:

$$\textbf{Velocity update:} \quad \mathbf{v}_i^{n+1} = \mathbf{v}_i^n + \mathbf{a}_i^n \Delta t$$
$$\textbf{Position update:} \quad \mathbf{r}_i^{n+1} = \mathbf{r}_i^n + \mathbf{v}_i^n \Delta t$$

The softening factor ϵ is super important, because since we are not gonna implement collisions, our particles theoretically will be on top of each other, and thus the distance between them will be so infinitesimally small that the force will explode to such an incomprehensibly high number that our particles will zip out of existence.

2.1.2 Brute Force Algorithm

Analysis will mainly be focused on the Brute Force Algorithm of the N Body simulation. This is the worst case implementation of an N Body simulation, given that we are computing interactions of each and every possible pair of particles. Thus this algorithm has $O(N^2)$ time complexity. Which means, our runtime scales exponentially as the number of particles increases which is not ideal. However, the purpose of this project is to show that, even with the exponential scaling with increasing input, a myriad of optimizations can be applied in order to increase the performance of our simulation to be usable.

[1] "Astrophysicists Reveal Largest-Ever Suite of Universe Simulations." Simons Foundation, 17 June 2025, www.simonsfoundation.org/2021/10/24/astrophysicists-reveal-largest-ever-suite-of-universe-simulations/.

For our simulation, it is necessary to define the structure of our particle/object as follow.

```
Object::Object()
    : position({0.0f, 0.0f}), velocity({0.0f, 0.0f}), radius(1.0f), mass(1.0f) {}

Object::Object(std::array<float, 2> position, std::array<float, 2> velocity, float radius, float mass)
    : position(position), velocity(velocity), radius(radius), mass(mass) {}

void Object::accelerate(float x, float y, float dt){
    velocity[0] += x * dt;
    velocity[1] += y * dt;
}

void Object::updatePos(float dt){
    position[0] += velocity[0] * dt;
    position[1] += velocity[1] * dt;
}
```

Listing 1: Object Class Implementation

```
std::vector<float> computeAcceleration(const Body& obj1, const Body& obj2, float s_gamma, float s_soft)
    {
    std::vector<float> acc(2, 0.0f);

    // Return zero acceleration if the objects are the same
    if (&obj1 == &obj2) {
        return acc;
    }

    float x1 = obj1.position[0];
    float y1 = obj1.position[1];
    float x2 = obj2.position[0];
    float y2 = obj2.position[1];

    float m1 = obj1.mass;
    float m2 = obj2.mass;

    // Compute softened distance to avoid singularity
    double r = sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2) + s_soft);

    if (r > 0) {
        double k = s_gamma * m2 / (r * r);
        acc[0] += k * (x2 - x1);
        acc[1] += k * (y2 - y1);
    } else {
        acc[0] = acc[1] = 0;
    }

    return acc;
}
```

Listing 2: Acceleration Computation Function

The code snippet above, shows an implementation of the brute force algorithm. The `computeAcceleration` function will be run in a nested for loop on every possible pair of bodies/particles.

2.1.3 Simulation Setup

The simulation will be done in C++ and will be visualized using the OpenGL library. Our simulation, will involve computing the acceleration experienced by each body, updating it's velocities and positions, and rendering this change in position using OpenGL. However before moving on to implement our simulation it's important to consider how the different simulation optimizations will be evaluated

2.1.4 Simulation Evaluation

Given that this exploration is concerned about how fast our simulation can be run, the following performance metrics will be evaluated.

- Force Calculation Time: This is important to consider as it is most likely where most of the computation will be in. In N Body simulations, Force Calculation is usually the most computationally intensive, especially as we increase our number of particles. This project will thus mainly be focused on optimizing this part of the simulation

- Render Time: This could potentially have a significant computation impact since the simulation will be rendering thousands of particles. The algorithm used to render the particles will be evaluated and compared in order to see if it causes any potential bottlenecks in the simulation.
- Frames Per Second: Given that this a real-time simulation, it is important to consider how smooth it appears to the user, hence this is an important factor to measure. A FPS value of 30 should be targeted as that is typically the standard considered acceptable for interactivity.

2.1.5 Profiling Setup

Unfortunately, there is no external way to monitor the performance metrics mentioned above, thus a custom profiler was built in order to record the performance metrics of the simulation. Before the start of each simulation run the following vectors will be initialized

- forceTimes: This ,or each step of the simulation, is a vector storing how long it took to compute the forces experienced by each and every single particle
- renderTimes: This, for each step of the simulation stores, how long it took to render every single particle.
- stepTimes: This is a vector, storing how long each step took to run. (It includes the time of both the force calculation and the render time.
- fpsValues: This stores the FPS value of each simulation.

These vectors will accumulate their associated values for each time step. At the end of the program, the collected performance metrics will be stored in a CSV file for future analysis.

```
void writeToCSV(const std::string& filename, int num_bodies, int steps,
    /*
      forceTimes:  Time to compute forces per step
      renderTimes: Time to render particles per step
      stepTimes:   Total time (calculation + render) per step
      fpsValues:   FPS value of each simulation step
    */
    const std::vector<double>& forceTimes,
    const std::vector<double>& renderTimes,
    const std::vector<double>& stepTimes,
    const std::vector<double>& fpsValues) {

    std::ofstream csvFile(filename, std::ios::app);

    static bool writeHeader = true;

    // Add Column names if this is the first write
    if (writeHeader) {
        csvFile << "num_bodies,step,force_time,render_time,total_time,fps\n";
        writeHeader = false;
    }

    // Add values to the csv file
    for (int step = 0; step < steps; ++step) {
        csvFile << num_bodies << ","
                << step + 1 << ","
                << forceTimes[step] << ","
                << renderTimes[step] << ","
                << stepTimes[step] << ","
                << fpsValues[step] << "\n";
    }

    csvFile.close();
}
```

Listing 3: CSV Export Function

2.1.6 Real Time Profiling

Instead of having to wait for the simulation to finish running, Real Time Performance metrics was implemented in order to get an idea of how the simulation performance changes as the simulation is running. At the end of each simulation step, the performance metrics are printed out at the terminal as seen below

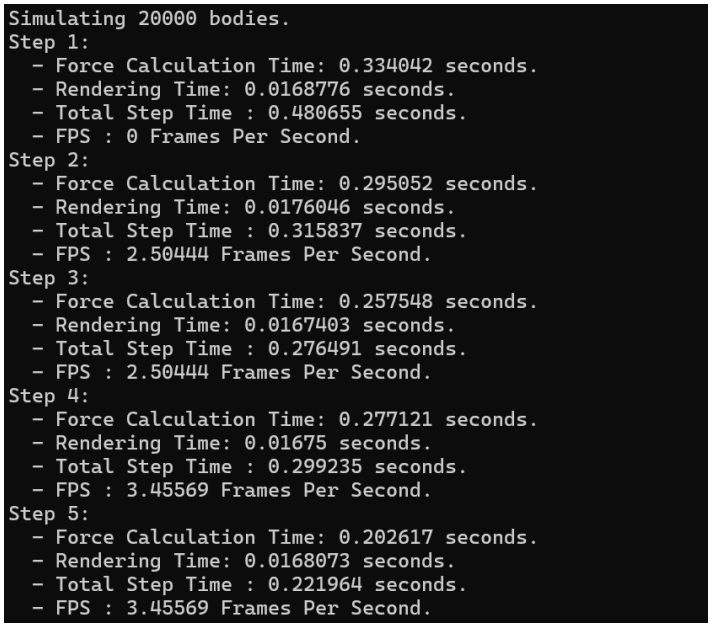

Figure 1: Real-Time Performance Metrics Demonstration. This figure shows the profiling output per step of the simulation.

2.1.7 Performance Summary

At the end of the simulation, to provide insight of the performance overall, a short performance summary is provided at the end of each simulation iteration.

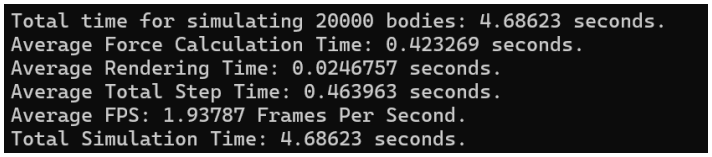

Figure 2: End Of Simulation Sumamry, shows the average of the mesasured performance metrics

2.1.8 Factors to Consider When Profiling

Now that a way to evaluate the simulation has been set up, there are certain factors that need to be considered while benchmarking the simulation. The performance of any software is affected by not only the underlying algorithms, but also other external and internal factors. The most important factors for evaluating the simulation performance is noted below.

- **Background Processes:** Other processes running on the computer can compute for CPU, GPU and Memory Resources, which may lead to reduced performance of the simulation. To account for this, all background processes are closed to ensure consistent and accurate profiling results.
- **On Charge VS On Battery:** Typically, when on battery, the computer enables power saving features which throttles both the CPU and GPU. Moreover, the GPU of a computer can have a higher power draw than the battery can provide on its own, thus leading to lower performance. All Simulations will thus be run On Charge.
- **Internal Temperature:** The Internal Temperature of the system is another important factor to consider. Higher Internal Temperatures can damage the internal components of the computer, thus CPUs/GPUs typically lower throttle their performance to prevent overheating. To account for this, simulations are started at a base temperature range of 55-60 degrees celcius.

2.1.9 Hardware Specifications

It is important to note that the performance of this simulation is largely dependant on the HardWare specifications. Simulations will be ran on a Lenovo Legion 5 Laptop fitted with an RTX 3070 Laptop GPU and an AMD 5800H Laptop CPU with 16GB of RAM.

2.1.10 Base Simulation Implementation

For our base simulation, the code snippets above gives a general idea of how it works. The nested for loops to evaluate each pair is seen below.

```
// Start of force calculations
for (int i = 0; i < planet_list.size(); ++i) {
    std::array<float, 2> total_acc = { 0.0f, 0.0f };

    // Calculate net acceleration on planet i due to all other planets
    for (int j = 0; j < static_cast<int>(planet_list.size()); ++j) {
        if (i == j) continue;
        std::array<float, 2> acc_from_other =
            physics.obtain_accelerations(planet_list[i], planet_list[j]);
        total_acc[0] += acc_from_other[0];
        total_acc[1] += acc_from_other[1];
    }

    // Update planet i's velocity and position
    planet_list[i].accelerate(total_acc[0], total_acc[1]);
    planet_list[i].update_position();
}
```

Listing 4: Force Calculation Loop for N-Body Simulation

2.1.11 Optimizations Considered

While making the base simulation, several decisions were made to ensure an efficient and decently optimized first implementation. These optimizations are listed below:

- **Using fixed sized arrays versus vectors for the Particle class:** Fixed-sized arrays were chosen to represent position, velocity, and acceleration in the `Particle` class. This decision was made to reduce memory overhead and improve cache locality, as arrays have a smaller memory footprint compared to dynamic vectors.
- **Reducing the amount of attributes in the particle class:** The `Particle` class was designed with only the most essential attributes necessary for simulation, such as position, velocity, acceleration, and mass. Avoiding redundant or unnecessary parameters ensures that each simulation step executes with minimal memory overhead.
- **Ensuring high precision values were avoided:** The use of `float` instead of `double` for most calculations reduces the computational cost of arithmetic operations while still maintaining sufficient precision for simulation accuracy. This minimizes the processing load, particularly in force calculation loops.

2.1.12 Initial Results

With the completed simulation code, and profiling, the performance of the algorithm can now be analyzed. The algorithm was run on different number of bodies namely: 1000, 2000, 5000, 10000, and 20000 bodies respectively. This provided a good enough range to see the exponential increase of the Brute Force algorithm. For each N, the simulation was ran for 10 steps. This is due to the fact that it provided the perfect balance of not having too little steps that our results are sensitive to random error and also not too much that it is time consuming, as the higher the number of bodies, the exponentially longer it takes. Moreover, the given range was chosen, because 1000,is small enough to provide a smooth simulation and 20,000 is big enough to show the drastic increase in runtime while not being time consuming to profile. N=50,000 and N = 100,000, will be used once suffiecient optimizations have been applied.

Table 1: Simulation Performance Metrics and Frame Rates Across Varying Body Counts

Bodies (N)	Total Time (s)	Render Time (s)	Force Time (s)	Avg. FPS
1,000	0.0989	0.0083	0.0894	5.0826
2,000	0.3566	0.0158	0.3405	2.5059
5,000	2.1046	0.0366	2.0674	0.4750
10,000	8.8213	0.0805	8.7401	0.1136
20,000	36.3942	0.1899	36.2028	0.0275

In running the simulation, we get an average of 5 FPS, for N = 1000, and an average of 0.027510 FPS for N = 20,000. This is a 184 $\times$ reduction in FPS for a 20 $\times$ increase in N. Confirming the exponential scaling of our algorithm.

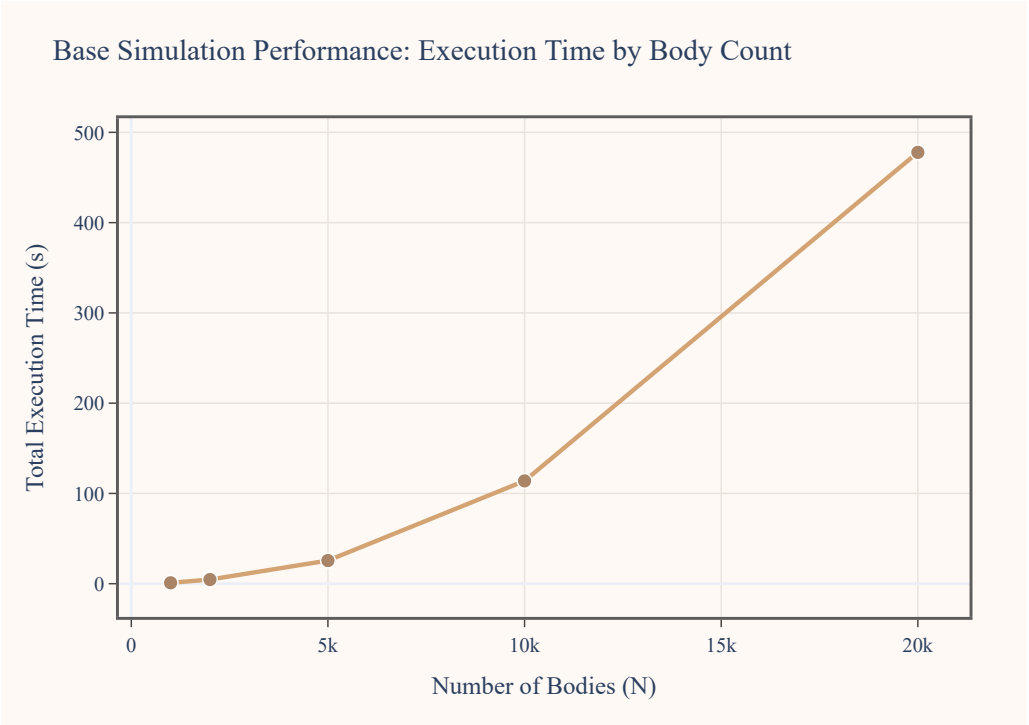

Figure 3: Simulation execution time on Number of Bodies. Here we see an exponential scaling of runtime as N increases.

The Graph above shows a perfect visualization of the $O(N^2)$ runtime complexity of the Brute force algorithm created. Obviously this is not ideal, as waiting 36.4 seconds to move from one frame to the next is not fulfilling the user requirements of a real time simulation.

2.1.13 Visual Studio Performance Analyzer (Perf) to Identify Bottlenecks

At a glance it seems obvious that it is the Force Calculation which takes up most of the CPU Usage, however, it is important to look at a detailed breakdown of the computational load to get a better idea of what optimization techniques should be used.

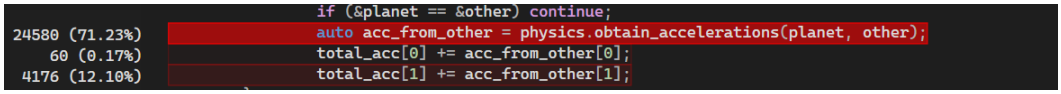

Figure 4: Performance Profiler

This demonstrates that the function to compute accelerations takes much of the computational load from the CPU within our simulation loop. The other overhead is caused by summing the accelerations for the second time. This initially did not make sense as the first and second additions take vastly different computational loads. This could be the compiler provided optimizations which change the structure of the code, or the CPU waiting for function calls interfering with the profiler. Either way, it's evident that most of the optimization efforts, should be geared towards the `obtain_accelerations` function specifically the square root function.

2.2 Optimizations Without SIMD

Thanks to the Performance Profiler, its confirmed that, the force calculation, specifically, the square root function, takes up must of the computational resources taking up to 29 clock cycles, compared to an add which takes just 1 clock cycle. Thus, with $20,000^2$ which evaluates to $400,000,000$ force calcuations and thus $400,000,000$ square root operations only.

$$400,000,000 \times 29 = 11,600,000,000 \text{ Clock Cycles}$$

11.6 Billion clock cycles for just the square root function for a single frame is substantial. Looking at an AMD 5800H, which has a 3.2GHZ clock speed for a single core, our theoretical max performance, for N = 20,000 will be:

$$\frac{11,600,000,000}{3,500,000,000} = 3.31 \text{ Seconds}$$

Given that, it took the Brute Force Implementation 36.2028 seconds per frame, there is still some performance we can squeeze from it, even counting the fact that some clock cycles where used for other parts of the force calucation, (Additon Multiplication ETC). This fact also confirms that it is not possible to achieve real time simulation on a single core, as we are taking more than 3 seconds per frame at an absolute best, far greater than the 30 FPS required. However, to get as close to that as we possibly can to maximize single core performance for this algorithm, there are some optimization techniques we can apply.

2.2.1 Techniques Applied

- Inlining: When a function is called, the CPU has to save the current state on a stack, jump to a new memory address and return to it's stored state onces its done performing the function. Given that our algorthim is $0(N^2)$, at N = 20,000, we are making 400 million of these jumps and back when we call `calcforces`, which take up clock cycles, creating a serious bottle neck as a result. To help with this, **Inlining** is enabled to put the function within the loop to completely get rid of the need to make jumps, as the function is now adjacent in memory. For this simulation, inlining as enabled using. Link Time Code Generation (LTCG) in Visual Studio allows the compiler to put together all header and implementation files in one singular file, allowing for aggressive inlining to take place.
- O2/Ox Compiler Optimizations: The 02 tag applies all sorts of optimization tricks to make a program faster. It allows the compiler to find ways to translate our code into fast and efficient assembly code. The 02 tags includes inlining, vectorization, and code structure based optimizations to allow the programme to perform effectively.
- FastMath: The fastmath optimization prioritizes execution speed over accuracy. It allows reordering of operations to prioritize performance as well as treating divisions as a multiplication operation with decimals.

2.2.2 Multi-threading

Now that, we have applied these above optimization tags in order to maximize single thread performance, the computational load, can be distributed between the threads that make up the computer's CPU. (AMD 5800H is capable of running 16 threads.) Given that, our force calculations is done over a nested for loop, and each thread will not need to edit the same data structure simultaneously. Multi-threading can be implementeted using `OpenMP`. The `pragma omp parallel for` is applied on the outer loop of the algorithm in order to minimize overhead from creating and destroying threads, which otherwise will happen many more times if the inner loop had been parallelized.

2.2.3 Performance Evaluation

The simulation with the applied optimizations was profiled together with the parallel implementation giving the following results.

Table 2: Performance Comparison (Total Simulation RunTime) of N-Body Simulation Implementations

	Execution Time (s)			**Speedup Factor**		
Bodies (N)	**Unopt.**	**Opt.** *(Single)*	**Parallel** *(Multi)*	**Compiler** *(Unopt→Opt)*	**Parallel** *(Opt→Par)*	**Total** *(Unopt→Par)*
1,000	1.01	0.22	0.30	4.7×	0.72×	**3.4**×
2,000	4.59	0.72	0.46	6.4×	1.55×	**9.9**×
5,000	25.65	3.36	1.00	7.6×	3.36×	**25.7**×
10,000	113.88	13.15	2.45	8.7×	5.42×	**47.0**×
20,000	477.85	51.04	7.41	9.4×	6.89×	**64.5**×
50,000	–	303.11	34.94	–	8.67×	–
100,000	–	1303.52	144.89	–	9.00×	–

From the table above, it is evident that even with just single-threaded performance, we achieve up to a 9.4× performance improvement for $N = 20,000$ bodies. These were simple optimizations that required no major refactoring of the algorithm. The most notable performance improvements appear when we implement multithreading. Compared to the unoptimized algorithm, this approach offers a massive 64.5× improvement in runtime at $N = 20,000$. Furthermore, as the simulation scales to $N = 100,000$, the parallel implementation has a higher performance increase, achieving a 9× speedup over the

single-threaded optimized version, effectively utilizing the available hardware threads. This demonstrates that for $\mathcal{O}(N^2)$ problems, combining compiler optimizations with parallel computing is essential for viable real-time performance.
The graph below shows the performance comparison of the optimized and unoptimized algorithms. The graph, also shows a separate line for the parallel implementation. It shows that even with performance and multi-threading, our algorithm still demonstrates $O(N^2)$ time complexity

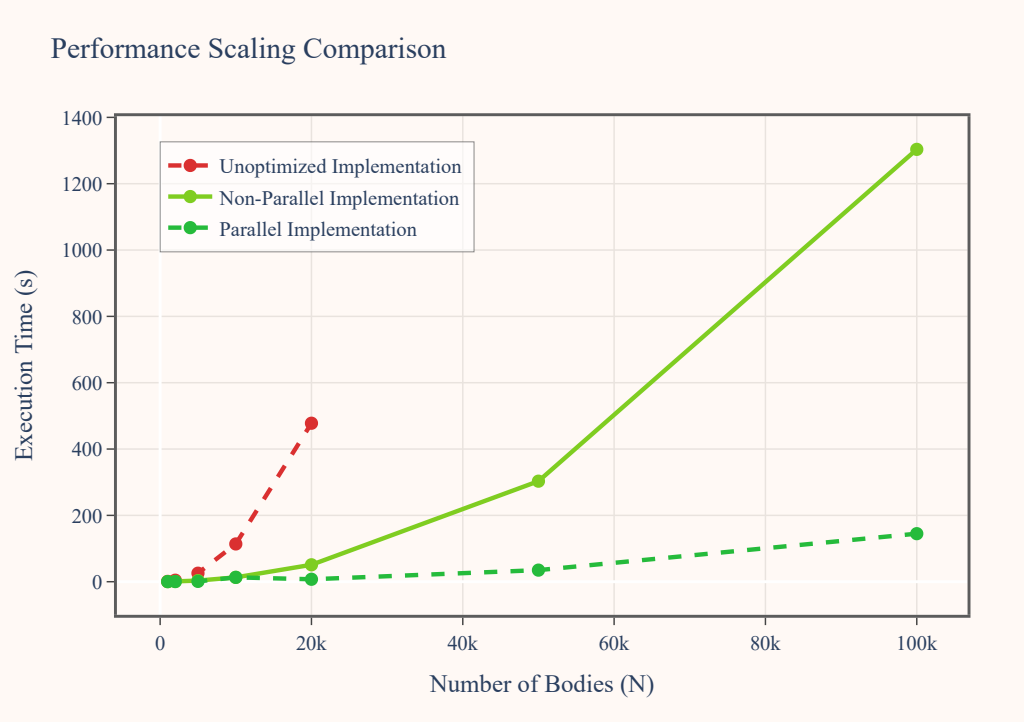

Figure 5: Comparing Parallel, Non Parallel and Optimized Implementations

It is important to note that even though multi-threading was performed, the expected 16x improvement is not really shown, offering at most a 9.00 × improvement for 100,000 bodies. For smaller N, the computational cost of calculating the forces is much lower than the overhead in managing the threads, which explains the lower performance improvement. However for higher, number of bodies, there should be an expected 16x improvement.

Figure 6: CPU Usage Across threads

CPU analytics in Task Manager show utilization across all 16 logical threads, indicating that the simulation effectively exploits hardware parallelism. Looking deeper, the Ryzen 7 5800H contains 8 physical cores, each supporting two logical threads via simultaneous multi-threading (SMT). Physical cores execute instructions, while SMT allows multiple instruction streams to share the same execution resources, improving utilization when one thread is stalled due to memory access or pipeline hazards. Thus, it makes sense to obtain a 9.0 × improvement, as the simulation could have obtained an 8 × improvement.

Thus, the simulation has been significantly optimized offering more than a 65 × improvement for N= 20,000, and potentially higher speedup improvements for higher number of N. However, there is still potential to squeeze greater performance out of the CPU through SIMD.

2.3 SIMD

SIMD (Single Instruction, Multiple Data) is a technique used by modern CPUs to perform the same mathematical operation on multiple data simultaneously. This implementation of SIMD will use AVX2, which provides 256 bit registers to store data which can be operated on and was inspired by the paper: **N-body simulation for self-gravitating collisional systems with a new SIMD instruction set extension to the x86 architecture, Advanced Vector eXtensions**[2]

2.3.1 Data Structure Implementation

The implementation, as mentioned will be using 256 bit registers. For the implementation of SIMD, the programme will, divide the particles into 2 categories. (I Particles) and (J Particles). In implementating SIMD, our algorithm will work on computing the attractions of pairs of formed with Two I particles and Four J particles simulataneously. For this, new data structures Iparticles and Jparticles where created.

```
struct Iparticles {
    double xpos[8];
    double ypos[8];
    float id[8];
};

struct Jparticles {
    double xpos[4];
    double ypos[4];
    float mass[8];
    float id[8];
};
```

Listing 5: Struct definitions for SIMD particle data

Iparticles have no mass attribute because these are the target particles we are computing the forces for, and given that we are ultimately solving for acceleration, the mass of the target particle cancels out. The acceleration $\mathbf{a}_i$ of a target particle i depends only on the masses of the source particles j:

$$\mathbf{a}_i = \frac{\mathbf{F}_{total}}{m_i} = \frac{1}{m_i} \sum_j G \frac{m_i m_j}{r_{ij}^2} \hat{\mathbf{r}}_{ij} = \sum_j G \frac{m_j}{r_{ij}^2} \hat{\mathbf{r}}_{ij}$$

`xpos` and `ypos` are arrays of 8 doubles which contain the x and y coordinates of two `Iparticles` i_0 and i_1, which are duplicated 4 times to fill the arrays.

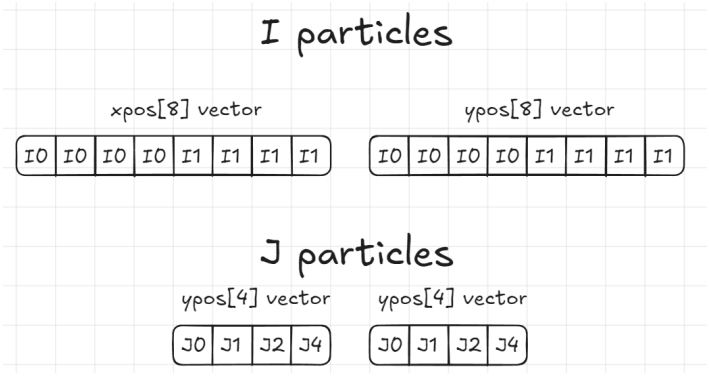

Figure 7: Diagram of J and I Particles

The j particles are stored in arrays of 4 doubles. The I particles and J particles are paired up as seen below, and the attraction between each of those pairs are computed simulataneously once SIMD has been implemented.

[2]Tanikawa, Ataru, et al. "N-Body Simulation for Self-Gravitating Collisional Systems with a New SIMD Instruction Set Extension to the X86 Architecture, Advanced Vector Extensions." arXiv.Org, 6 Sept. 2011, arxiv.org/abs/1104.2700.

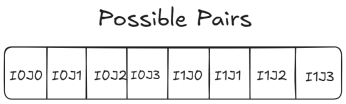

Figure 8: The above figure shows the paired attractions that are computed simultaneously

2.3.2 Force Calculation Implementation

With the data structure defined, the physics loop can now be implemented. The first step is to load the particle data into m256 registers, which will allow for simultaneously instruction execution.

```
// We load J-particles into the 256-bit registers. Each register can hold 4 double (64-bit) values
__m256d j_x = _mm256_load_pd(j_part.xpos);
__m256d j_y = _mm256_load_pd(j_part.ypos);

// We load I-particles. Only two distinct positions are loaded, but duplicated across the register slots
__m256d i0_x = _mm256_load_pd(&i_part.xpos[0]);
__m256d i1_x = _mm256_load_pd(&i_part.xpos[4]);
__m256d i0_y = _mm256_load_pd(&i_part.ypos[0]);
__m256d i1_y = _mm256_load_pd(&i_part.ypos[4]);
```

Listing 6: Loading particle data into AVX registers

Next, calculations are performed using AVX intrinsics simultaneously on the particles stored in the m256 registers.

```
// Calculate displacement vectors for pairs of I and J particles
__m256d dx_i0 = _mm256_sub_pd(j_x, i0_x);
__m256d dx_i1 = _mm256_sub_pd(j_x, i1_x);
__m256d dy_i0 = _mm256_sub_pd(j_y, i0_y);
__m256d dy_i1 = _mm256_sub_pd(j_y, i1_y);

//Convert displacements to single precision to fit 8 values per 256-bit register
__m256 dx = _mm256_insertf128_ps(
    _mm256_castps128_ps256(_mm256_cvtpd_ps(dx_i0)),
    _mm256_cvtpd_ps(dx_i1), 1
);
__m256 dy = _mm256_insertf128_ps(
    _mm256_castps128_ps256(_mm256_cvtpd_ps(dy_i0)),
    _mm256_cvtpd_ps(dy_i1), 1
);

//We now calculate the distance, between the pairs of I and J particles(Compute r^2 and approximate 1/r using AVX intrinsics)
__m256 r2 = _mm256_add_ps(_mm256_mul_ps(dx, dx), _mm256_mul_ps(dy, dy));
__m256 inv_r = _mm256_rsqrt_ps(r2);

//Handle self-interaction: mask out particles with identical IDs
__m256 id_i = _mm256_load_ps(i_part.id);
__m256 id_j = _mm256_load_ps(j_part.id);
__m256 mask = _mm256_cmp_ps(id_i, id_j, _CMP_NEQ_OQ);
inv_r = _mm256_and_ps(inv_r, mask); // Zero out inverse distance if IDs match

//Compute Gravitational Force(G * mass * 1/r^3)
__m256 inv_r3 = _mm256_mul_ps(_mm256_mul_ps(inv_r, inv_r), inv_r);
__m256 factor = _mm256_mul_ps(_mm256_set1_ps(s_gamma), _mm256_load_ps(j_part.mass));
factor = _mm256_mul_ps(factor, inv_r3);
```

Listing 7: AVX Force Calculation Kernel

Now that accelerations are computed, they are stored in the accumulator, which is a data structure which stores the computed accelerations in a m256 register. For a particle i_0, the attraction between i_0 and four other j particles are simultaneously computed. This results in 4 accelerations, which are stored in the accumulator.

```
//Compute acceleration components in single precision
__m256 ax_sp = _mm256_mul_ps(dx, factor);
__m256 ay_sp = _mm256_mul_ps(dy, factor);

//Extract lower/upper halves, convert back to double, and accumulate
acc.ax_i0 = _mm256_add_pd(acc.ax_i0,
    _mm256_cvtps_pd(_mm256_extractf128_ps(ax_sp, 0)));

```

```
acc.ax_i1 = _mm256_add_pd(acc.ax_i1,
    _mm256_cvtps_pd(_mm256_extractf128_ps(ax_sp, 1)));

acc.ay_i0 = _mm256_add_pd(acc.ay_i0,
    _mm256_cvtps_pd(_mm256_extractf128_ps(ay_sp, 0)));

acc.ay_i1 = _mm256_add_pd(acc.ay_i1,
    _mm256_cvtps_pd(_mm256_extractf128_ps(ay_sp, 1)));
```

Listing 8: language=C++, Accumulating acceleration components

2.3.3 Evaluating SIMD Performance

Now that SIMD instructions has been implemented, benchmarking can be run in order to evaluate the performance improvement as a result of SIMD. The physics simulation loop is set up to perform the force calculations on all possible pairs of particles, and parallelized with `pragma omp parallel for`

Table 3: Total Simulation Times and Speedup (SIMD vs. Parallel)

Number of Bodies (N)	**Total Time (s)**	**Speedup (vs Parallel)**
1,000	0.342057	0.9×
2,000	0.117792	**3.9**×
5,000	0.136929	**7.3**×
10,000	0.408232	**6.0**×
20,000	1.210390	**6.1**×
50,000	6.715220	**5.2**×
100,000	25.444100	**5.7**×

10 steps of N = 100,000 were run in 25 seconds, which is a substantial improvement from the parallel implementation which ran N = 100,000 in 149 seconds. Which is an approximately 6 × increase in performance.

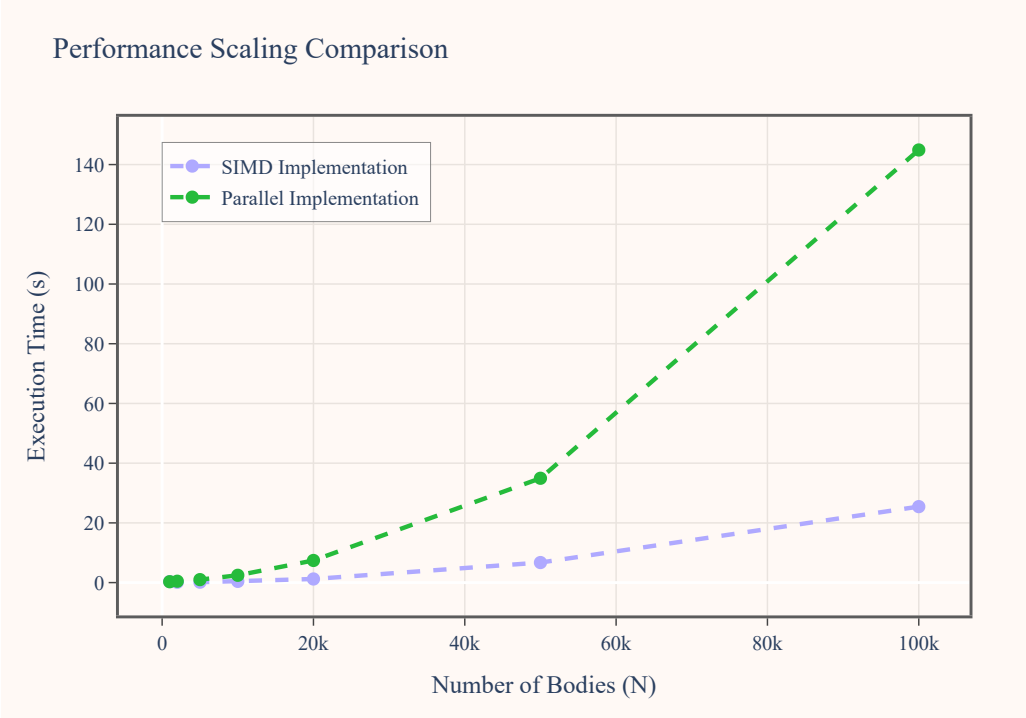

Figure 9: Comparing Runtime of SIMD and Parallel Optimizations

This performance gain of approximately 6× aligns well with, and slightly exceeds, the theoretical expectations for this specific hardware implementation. Since AVX2 registers are 256 bits wide, they can hold 4 double-precision values (4×64 bits). Thus, a naive vectorization would be expected to yield a speedup factor close to 4×. However, the observed speedup

is higher because the implementation utilizes *mixed-precision arithmetic*. While particle positions are loaded as doubles (preserving the 4× memory bandwidth limit), they are converted to single-precision floats for the force calculation, allowing the arithmetic units to process 8 interactions in parallel. Additionally, the use of the hardware-approximate inverse square root intrinsic (`_mm256_rsqrt_ps`) replaces the computationally expensive division and square root operations, further boosting the simulations through put
The simulation has gone under significant performance improvements from where it originally started. The next section will look at an overview of the total performance increase from all the optimizations that have been applied to this simulation program.

3 Results

Table 4: Performance Comparison: Unoptimized vs. Optimized vs. Parallel vs. SIMD

	Execution Time (s)				**Speedup Factor**			
Bodies (N)	**Unopt.**	**Opt.** *(Single)*	**Par.** *(Multi)*	**SIMD** *(AVX2)*	**Comp.** *(Unopt→Opt)*	**Par.** *(Opt→Par)*	**SIMD** *(Par→SIMD)*	**Total** *(Unopt→SIMD)*
1,000	1.01	0.22	0.30	0.34	4.7×	0.7×	0.9×	**3.0**×
2,000	4.59	0.72	0.46	0.12	6.4×	1.6×	3.9×	**39.0**×
5,000	25.65	3.36	1.00	0.14	7.6×	3.4×	7.3×	**187.3**×
10,000	113.88	13.15	2.45	0.41	8.7×	5.4×	6.0×	**279.0**×
20,000	477.85	51.04	7.41	1.21	9.4×	6.9×	6.1×	**394.8**×
50,000	–	303.11	34.94	6.72	–	8.7×	5.2×	–
100,000	–	1303.52	144.89	25.44	–	9.0×	5.7×	–

From the base model to the SIMD Optimized version, the simulation obtained a total of approximately 400 × performance increase. This performance increase is very significant as it took the simulation from computing N = 20,000 for 10 steps in 477 seconds, to 1.21 seconds, which is approximately 11 frames per second. While not close to the desired 30 frames per second, it still provides a nice visualization of gravitational attractions and is certainly better than having to wait close to a minute to view a single frame. Thus SIMD implementation with AV2 intrinsics combined with compiler optimizations and multi threading should seriously be considered in the development of simulations or other similar programs.

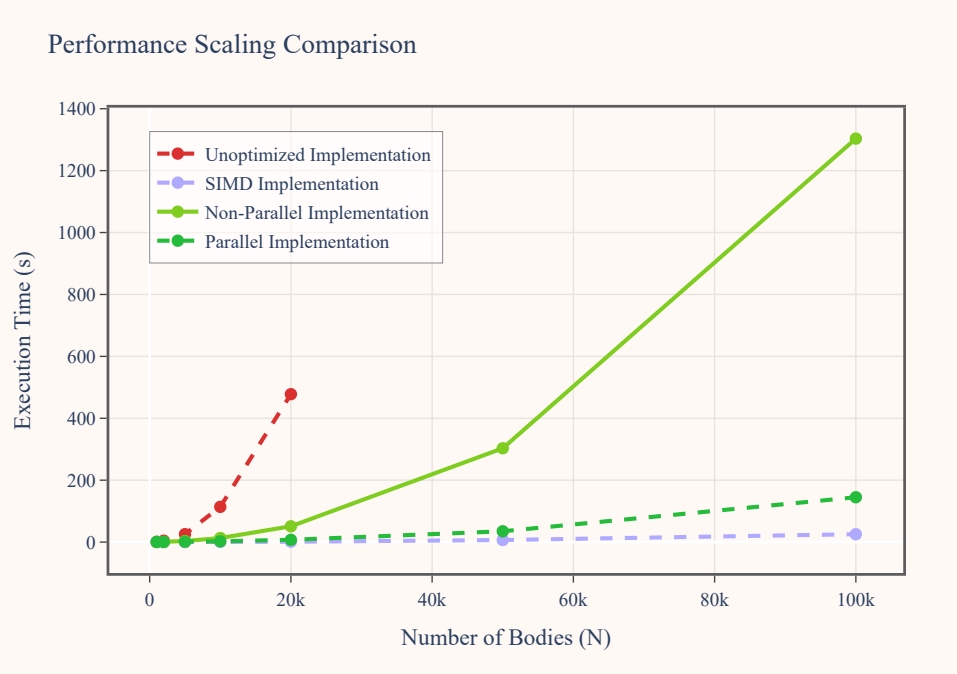

Figure 10: Comparing all performance improvements

The graph above visualized the performance increase of the algorithm, better demonstrating the 400 × performance gain.

4 Evaluation

4.1 Bottlenecks and Challenges

- Full L1/L2 Cache: It's possible that due to high number of particles, especially for N = 100,000, the CPU's L1 and L2 cache gets filled up. Thus the CPU is forced to use slower memory to store, stalling the CPU as it has to wait for data
- Use of Doubles: The use of the double data type in the m256 registers reduces the number of values that can be stored. Using doubles means only four values can fit in the register compared if single precision floating points were used, which would allow 8 values to fit into the m256 registers, effectively doubling the memory bandwidth. The trade off of this is accuracy, which is something that is not considered a priority for this project.
- AVX2 vs AVX-512 instructions: The SIMD implementation used AVX2 instructions which provided 256 registers, compared to AVX-512 which provides 512, hence almost half the performance is lost using AVX2 instead of AVX-512.
- Accuracy: While simulation accuracy is not a priority, it is still something that needs to be considered and is a significant downside to this implementation as accuracy was not evaluated or adjusted for.
- Inefficient Brute Force ($O(N^2)$) Algorithm: The Brute Force algorithm is a significant contributer to the computation load, especially for high values of N, as no amount of optimizations to the Brute Force algorithm will stop it from scaling poorly with increasing N.

4.2 Future Improvements

- Using Tiling To Fix Cache Issues: The working set of the simulation can be adjusted to use only an amount of data that can fit in the L1/L2 cache, thus preventing data to be tossed away only to have to be loaded from slower memory.
- Changing From Doublesto Floating Point: This doubles the memory bandwidth of the simulation, allowing for a theoretical 2x performance increase.
- Using AVX-512 Instructions: AVX-512, doubles the amount stored in the registers thus doubling the bandwidth, combined with using floating point values, this could potentially quadruple the performance of the SIMD implementation.
- Implementing better N Body Algorithms: Spatial Partitioning algorithms such as the Barnes Hut Tree reduces the time complexity down to $O(NLog(N))$ which allows the simulation to scale much better with increasing N

5 Conclusion

In short, multi-threading and SIMD instructions allow drastic and significant gains of performance, performing approximately 400 $\times$ faster than the original implementation (which already had optimizations implemented in it). This however is not close to the maximum performance that can be squeezed out. Without changing the algorithm, there is a theoretical 1600 $\times$ runtime improvement if AVX-512 is used with single precision floating points. Thus, there is still a lot of work to be done. However, this endeavor revealed the importance of profiling and evaluating the performance of programs, which is something I wish to develop my skills on in the future as I work on implementing faster algorithms for N body simulations.

References

[1] Simons Foundation. *Astrophysicists Reveal Largest-Ever Suite of Universe Simulations*. 17 June 2025. https://www.simonsfoundation.org/2021/10/24/astrophysicists-reveal-largest-ever-suite-of-universe-simulations/

[2] Ataru Tanikawa, Keigo Nitadori, and Junichiro Makino. *N-Body Simulation for Self-Gravitating Collisional Systems with a New SIMD Instruction Set Extension to the X86 Architecture, Advanced Vector Extensions*. arXiv preprint arXiv:1104.2700, 2011. https://arxiv.org/abs/1104.2700