Numerical simulation for the N-Body problem

Andrea Barletta, Luca Ballone,
Zhaochen Qiao, Rong Huang,
Yanlong Wang

Project for the *Advanced Method for Scientific Computing* course

High Performance Computing Engineering
Politecnico di Milano
A.Y. 2024/2025

Numerical simulation for the N-Body problem

Exploring different techniques and computing paradigms

Abstract

We present a numerical study of the n-body problem with a focus on performance-oriented methods suited for highly parallelized architectures. Our objective is to compare different simulation techniques across both implementation strategies and hardware platforms, specifically CPUs and GPUs. We analyze the accuracy and computational efficiency of several approaches, including a naive brute-force method, the Barnes–Hut algorithm, and a brief exploration of the Fast Multipole Method. By applying heuristic error analysis and performance benchmarking, we demonstrate the effectiveness of each method in large-scale scenarios. Our results show that, through optimized GPU implementations, it is possible to simulate millions of interacting bodies in real time on consumer-grade hardware, achieving both high accuracy and performance.

Contents

1. Introduction

1.1 Description of the problem

Simulating the interactions of multiple particles is a fundamental task across many scientific disciplines. In astrophysics, researchers model the motion of stars and galaxies to understand the structure and evolution of the universe. In chemistry and molecular biology, scientists simulate atomic and molecular dynamics to study chemical reactions, protein folding, and material properties. These problems, though diverse in context, share a common computational foundation: the n-body problem.

The n-body problem involves predicting the future positions and velocities of a collection of interacting particles based on their initial states and the forces between them. Due to the complexity of these interactions, especially when the number of particles is large, analytical solutions are rarely feasible. Instead, n-body solvers are used to approximate the system's behavior through numerical simulation.

In this project, we focus specifically on gravitational interactions, modeling systems such as star clusters or planetary systems. However, the code is designed with modularity and flexibility in mind, making it straightforward to extend the implementation to support other types of forces, such as electrostatic or Lennard-Jones potentials (see chapter 6).

1.2 Mathematical formulation

The gravitational N-Body problem concerns the motion of n point masses that interact solely through mutual gravitational attraction. The force between any two bodies is given by Newton's law of universal gravitation:

$$\vec{f}_{i,j}(t) = -G\frac{m_i m_j}{\left|\vec{s}_i(t) - \vec{s}_j(t)\right|^3}\left[\vec{s}_i(t) - \vec{s}_j(t)\right]$$

where:

- G is the gravitational constant ($\approx 6.7 \times 10^{-11}\ \ m^3 kg^{-1} s^{-2}$)
- $\vec{s}_k(t)$ is the position of the k-th body
- m_k is the mass of the k-th body

Each particle in the system experiences a total force that is the sum of the pairwise forces from all other particles:

$$\vec{F}_i(t) = \sum_{j \neq i} \vec{f}_{i,j}(t)$$

From Newton's second law, we can compute the particle's acceleration as:

$$\vec{a}_b(t) = \frac{\vec{F}_b(t)}{m_b}$$

This yields a system of second-order ordinary differential equations (ODEs) that govern the evolution of particle positions and velocities over time.

1.3 Challenges

Although the mathematical formulation of the gravitational n-body problem is conceptually straightforward, its numerical solution poses several well-known challenges.
Time integration instability is a key challenge, as particles with varying speeds or close separations require very small time steps for accurate resolution. Large time steps can lead to instability or unphysical results, whereas small steps increase computational cost. Moreover, close encounters between particles worsen this issue, where forces diverge as the distance approaches zero. Finally, many-body systems are inherently chaotic, so even tiny numerical errors can grow exponentially, rendering long-term trajectories unreliable.

1.4 Evaluating accuracy: Energy

In any closed system, the total energy (kinetic + potential) should be conserved. Therefore, if the computed total energy drifts significantly, it indicates accumulating integration or round-off errors. Small, bounded fluctuations are acceptable and expected, but a systematic increase or decrease signals instability. Comparing energy conservation across different integrators or time steps helps evaluate their accuracy. Thus, energy error serves as a practical benchmark for diagnosing and controlling numerical reliability [7].

2. The Naive method

2.1 First numerical formulation

As a first numerical approach, we implemented the forward Euler method to integrate particle positions and velocities over time. This method is simple and computationally inexpensive, however it is only conditionally stable and tends to accumulate energy errors rapidly, making it unsuitable for long-term or high-precision simulations.
In particular, we compute the updated velocities and positions at time $T + \Delta t$ as follows:

$$\vec{s}_q(t + \Delta t) \approx \vec{s}_q(t) + \Delta t \vec{v}_q(t)$$

$$\vec{v}_q(t + \Delta t) \approx \vec{v}_q(t) + \Delta t \vec{a}_q(t)$$

2.1.1 First optimizations

Newton's third law states that the force particle i exerts on particle j is equal in magnitude and opposite in direction to the force j exerts on i. By computing the interaction once and applying it to both particles, we can reduce the total number of force calculations by half.

2.1.2 Accuracy for the naive method

As previously discussed, we use total energy as a key metric to evaluate the accuracy of our simulation. The forward Euler method, while simple to implement, introduces noticeable energy errors at each time step, causing the total energy to drift over time, as shown in the figure below.

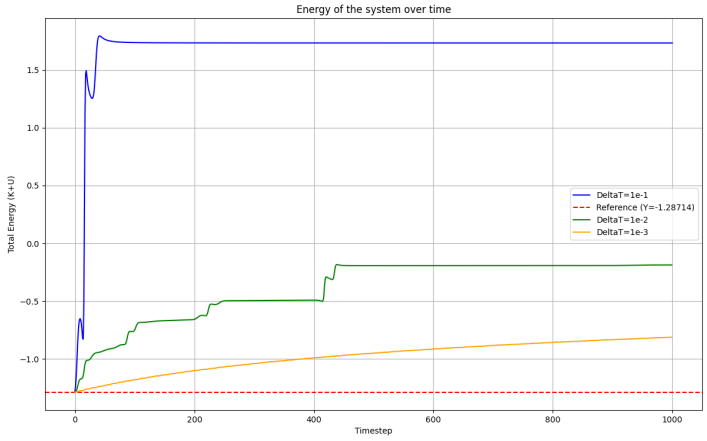


Figure 2.1: Comparing the energy behavior for varying Δt

2.2 Improving the integration scheme

A better integration scheme would be one that is symplectic, which is designed to maintain long-term stability and better conserve energy by respecting the underlying physics of the system.
We will use one of the simplest symplectic integrator, the Leapfrog integrator [12], which updates the velocities and positions at time $T + \Delta t$ as follows:

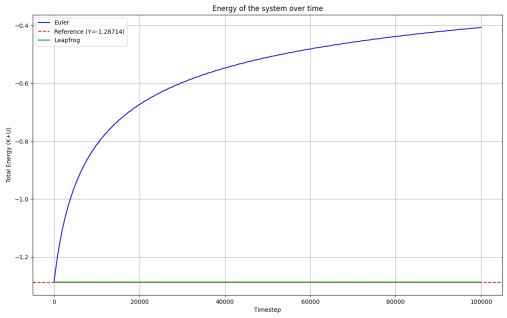
$$\vec{v}_q(t + \frac{\Delta t}{2}) = \vec{v}_q(t) + \vec{a}_q(t)\frac{\Delta t}{2}$$

$$\vec{s}_q(t + \Delta t) = \vec{s}_q(t) + \Delta t\vec{v}_q(t + \frac{\Delta t}{2})$$

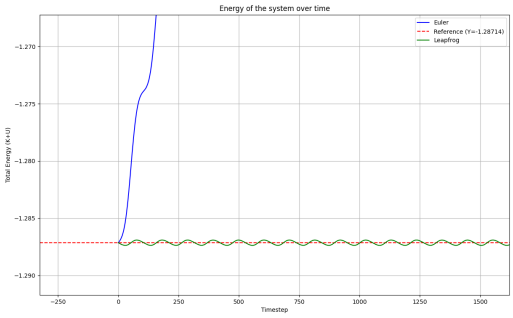
$$\vec{v}_q(t + \Delta t) = \vec{v}_q(t + \frac{\Delta t}{2}) + \vec{a}_q(t + \Delta t)\frac{\Delta t}{2}$$

2.2.1 Comparison of the two methods

By plotting the variation of total energy over time, we can clearly see the different behavior of the two integration methods. The forward Euler method shows a clear drift in energy. In contrast, the leapfrog method demonstrates much more stable energy behavior, with only small, bounded oscillations around the true value.



(a) Euler vs Leapfrog



(b) Euler vs Leapfrog, zoomed at origin

2.2.2 Handling collision

When computing pairwise interactions, a softening parameter $\epsilon^2 > 0$ is used to limit the magnitude of the gravitational force $\mathbf{f}_{ij}$, which would otherwise grow unbounded as the particles get too close, removing the need for explicit collision handling [9].

$$\mathbf{r}_{ij} = \mathbf{x}_j - \mathbf{x}_i$$

$$\mathbf{f}_{ij} = Gm_im_j\frac{\mathbf{r}_{ij}}{\left(\|\mathbf{r}_{ij}\|^2 + \epsilon^2\right)^{\frac{3}{2}}}$$

Here, we tested a system with many collisions to better understand the role of the softening factor in stabilizing the simulation.

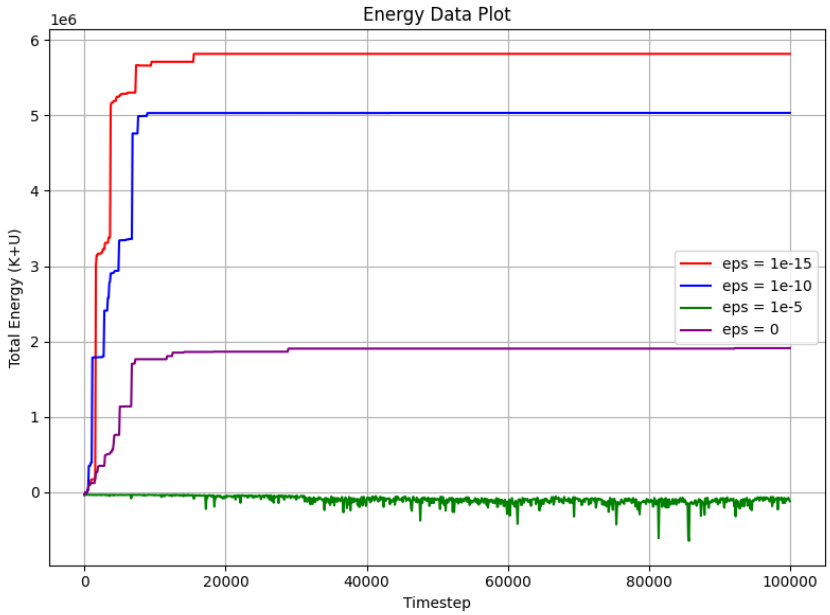


Figure 2.3: 100 Bodies in a small box, collisions are invetiable

It's important to note that introducing a softening factor adds an additional parameter to the simulation, which must be carefully tuned to balance accuracy and stability. If the softening length is too small, it may not effectively prevent large forces during close encounters; if it's too large, it can over-smooth interactions. Choosing an appropriate softening value is therefore critical, and this tuning often depends on the specific physical context and the spatial resolution of the simulation, making it a nontrivial but essential aspect of model design.

2.3 Going parallel: OpenMP

OpenMP [11] can be used to significantly speed up N-body simulations by parallelizing the force calculation loop. Since each pairwise interaction is independent, the workload can be distributed across multiple threads, allowing forces on different particles to be computed simultaneously. OpenMP enables efficient scaling on multicore CPUs with minimal changes to the code, resulting in substantial performance gains without compromising accuracy.

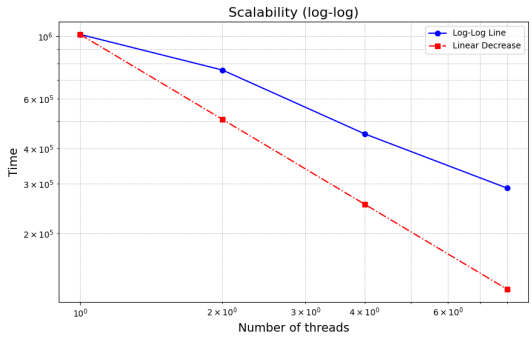


Figure 2.4: Strong scalability plot (using 100 bodies, time measured µs): Sub-linear speedup

Implementing the OpenMP solution was relatively straightforward and provided a clear performance boost, but its scalability remains limited. For simulations involving tens of thousands of bodies or more, this approach becomes impractical. To handle larger systems efficiently, more advanced simulation techniques are required, as discussed in the following chapters.

2.4 Further Scaling: CUDA

Using the same integration scheme and naive method, we aim to evaluate the performance improvements achieved by GPU acceleration with CUDA, in comparison to the parallelized OpenMP version.

2.4.1 Implementation

In our CUDA implementation (taken from [9]), each thread is assigned to compute the total acceleration acting on a single particle by iterating over all others, allowing fine-grained parallelism that efficiently utilizes the GPU cores. To reduce the overhead of global memory access, the particle data are stored in shared memory using a pointer of type `Body<Real, dim>*`. Data transfer between host and device is handled using `cudaMemcpy`, ensuring that input and output values are correctly exchanged before and after kernel execution. We also implemented shared-memory tiling to allow threads within a block to collaborate and minimize redundant memory access (however, this last optimization resulted in only minor performance improvements, likely due to limited problem size or memory access patterns that did not fully benefit from tiling)

2.4.2 Energy Stability of CUDA Version

Since the CUDA implementation uses the same Leapfrog integration scheme as the OpenMP version, the energy stability of the simulation should remain effectively identical. Both versions perform the same sequence of arithmetic operations, just distributed differently across hardware.

2.4.3 Benchmark

We compare the average execution time per simulation step in the three implementations: CPU, OpenMP, and CUDA, while keeping the time step Δt and the total simulation time T fixed. This allows us to analyze and compare how each version scales with an increasing number of particles. In particular, we examine the performance trend of the CUDA implementation as the number of particles grows, highlighting its scalability and computational efficiency on the GPU.

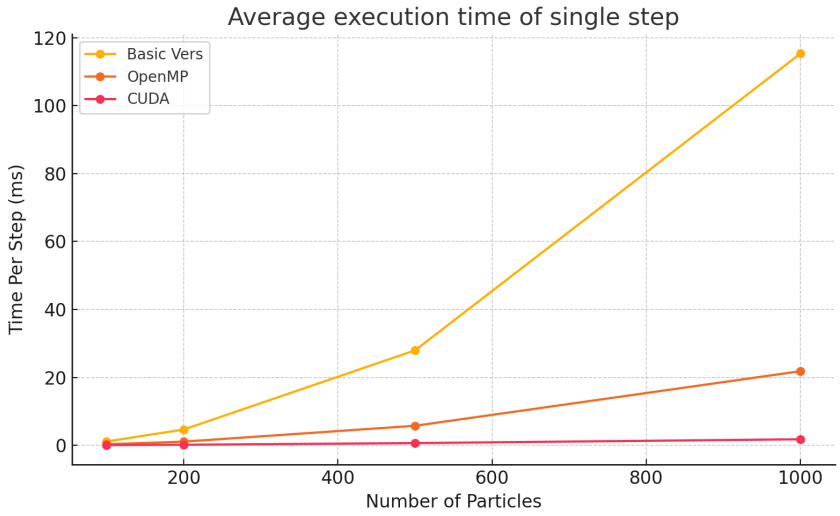


Figure 2.5: Comparison of the various methods ($\Delta t = 0.001$ and $T = 1$)

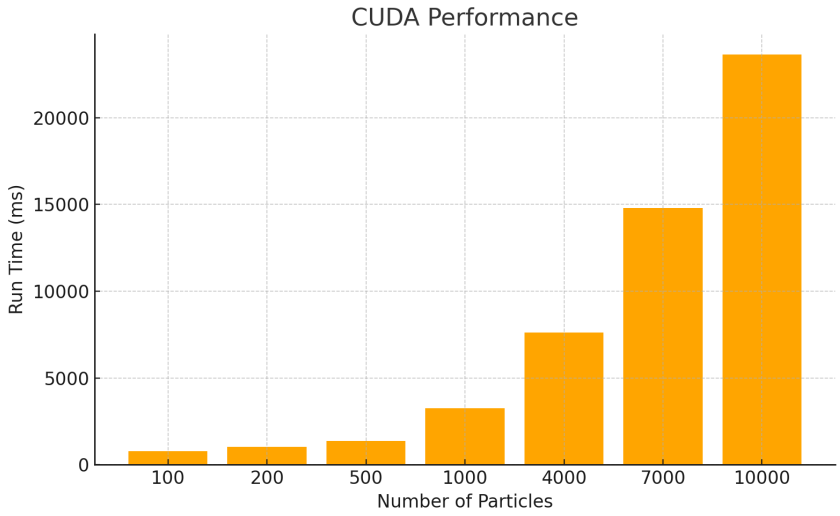


Figure 2.6: Total runtime for the CUDA version ($\Delta t = 0.01$ and $T = 1000$)

2.4.4 Further improvements

There are several possible ways to improve the performance of the CUDA implementation. One approach is to merge kernels where possible, reducing the overhead of multiple kernel launches. Switching from `double` to `float4` can also boost performance by taking advantage of faster single-precision operations and better memory alignment. Loop unrolling may help reduce instruction overhead and improve throughput in performance-critical sections. Additionally, using CUDA-specific functions such as `_fsqrt_rn` instead of the standard `sqrt` can provide more efficient, hardware-optimized operations. Finally, tuning the tile size based on the number of particles can help ensure better occupancy and memory access efficiency on the GPU.
But to have better scalability, the only way is to try a completely different approach.

3. The Barnes-Hut method

3.1 Basic algorithm

The Barnes-Hut algorithm leverages space partitioning to reduce the runtime of the naive method to $O(n \log n)$ by approximating distant clusters of particles as single point masses [3]. At each timestep, the particles are inserted into an octree, or quadtree in 2d, which is then traversed from the root in order to compute the net force acting on a single body. In the basic version, the nodes of the tree are either approximated as single bodies, or their children are recursively visited, based on a distance criterion of the type:

$$d > \frac{s}{\theta} \tag{3.1}$$

where d is the distance of the current particle to the barycenter of the given node, s is the size of the node (e.g. the length of one side of the enclosed volume) and $\theta \in [0, 1]$ is a parameter that governs the strength of the approximation. Smaller values of θ result in more nodes being visited and thus increase the overall accuracy of the force evaluation, while larger values allow trading accuracy with speed. The leaves of the tree are the particles themselves, and if reached, they have their force contribution being directly computed, hence when $\theta = 0$, all of the nodes are visited and the algorithm turns into the naive $O(n^2)$ method.

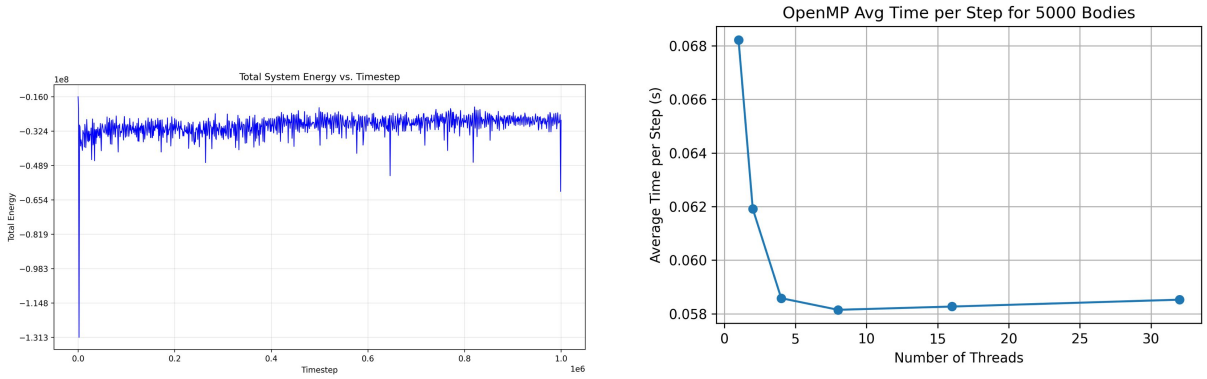
3.1.1 Parallelization with OpenMP

Traversing the tree can be performed in parallel for each particle, as well as for the individual integration updates. In this implementation, OpenMP was used to parallelize both the force computation and the update step across all particles.

A parallel loop is applied during the traversal of the Barnes-Hut tree to compute forces acting on each body. Since the force computation for each particle is independent, this part is well-suited for parallelization. The `#pragma omp parallel for` directive distributes the workload across available threads, significantly improving performance as the number of particles increases.

The integration step, where velocities and positions are updated based on the computed forces, is also parallelized using a similar directive. Race conditions are avoided as each thread updates only its assigned particle.

The results produced by the OpenMP-parallelized version are consistent with the serial implementation. As shown in Figure 3.1a, the total energy of a 50-body system remains nearly constant throughout the simulation, confirming both numerical stability and physical correctness.



(a) Energy evolution of 50-body system.

(b) Average time per step vs thread count (5000 bodies).

Figure 3.1: Results from OpenMP-parallelized Barnes-Hut simulations.

Performance Scaling with OpenMP

The average computation time per step in a 5000-body simulation was evaluated with varying numbers of OpenMP threads. As shown in Figure 3.1b, a significant reduction in time per step is observed when increasing the thread count from 1 to 8, indicating effective parallel scalability in this range.

For thread counts beyond 8, the performance improvement plateaus and slightly deteriorates. This behavior is attributed to increased parallel overhead, such as thread management, synchronization, and memory bandwidth contention. These findings highlight that OpenMP parallelization is most efficient up to a certain number of threads, after which diminishing returns are encountered.

3.2 Fully parallel GPU implementation with CUDA

A fully parallel 3D version of the Barnes-Hut algorithm, entirely running on the GPU, was implemented from scratch using CUDA [10] and OpenGL, and it was able to scale to systems of a few millions of particles on a laptop GPU (Figure 3.2). The particles are both simulated and rendered in real-time to avoid I/O latencies, and the memory required for the whole simulation is mostly pre-allocated on the GPU, requiring negligible data transfer with the CPU. Thrust and Cub are used for a few highly optimized parallel primitives (e.g. sorting), while the rest of the implementation runs on custom kernels. A structure of array (SoA) layout is used to store buffers for particles and octree nodes, favouring memory coalescing in most of the situations.

3.2.1 Parallel octree construction

The octree is built in a massively parallel manner, similarly to what described by Karras [8], by first constructing a binary radix tree on the z-order encoded particles position and then converting it to a linear octree. Differently from Karras, the octree leaves are allowed to contain more than one particle using the approach presented by Cazalbou [4], and the nodes are laid out contiguously in memory in breadth-first order, to allow for a more efficient vectorization during tree traversal [2].

The main ideas behind the octree construction are outlined, highlighting some implementation details and the novelties with respect to the referenced works.

Morton order The Morton order, or z-order curve, is a space filling curve that preserves spatial locality, efficiently computed by interleaving the bits of the grid-mapped coordinates of a point. The idea is to sort the particles based on their Morton code, and consider the set of unique codes as leaf nodes of the binary radix tree defined by Karras [8]. The fixed resolution of the z-order curve imposes a limit on the maximum depth of the tree nodes, more than one particle could in fact be mapped to the same code, providing no control over the number of particles assigned to the same leaf, but in practice, 64-bit codes are more than enough to mitigate the issue.

Sorting the particles by their Morton code is a crucial step that enables several optimizations during both construction and traversal of the octree. Indeed, particles belonging to any node of the octree are always contiguous in memory, requiring to keep track only of two indices per node, and allowing coalesced memory accessed when evaluating interactions during traversal. The spatial locality offered by the z-order curve is also exploited when traversing the tree, allowing effective group traversal and minimizing threads divergence, since spatially close particles will walk through similar paths down the octree.

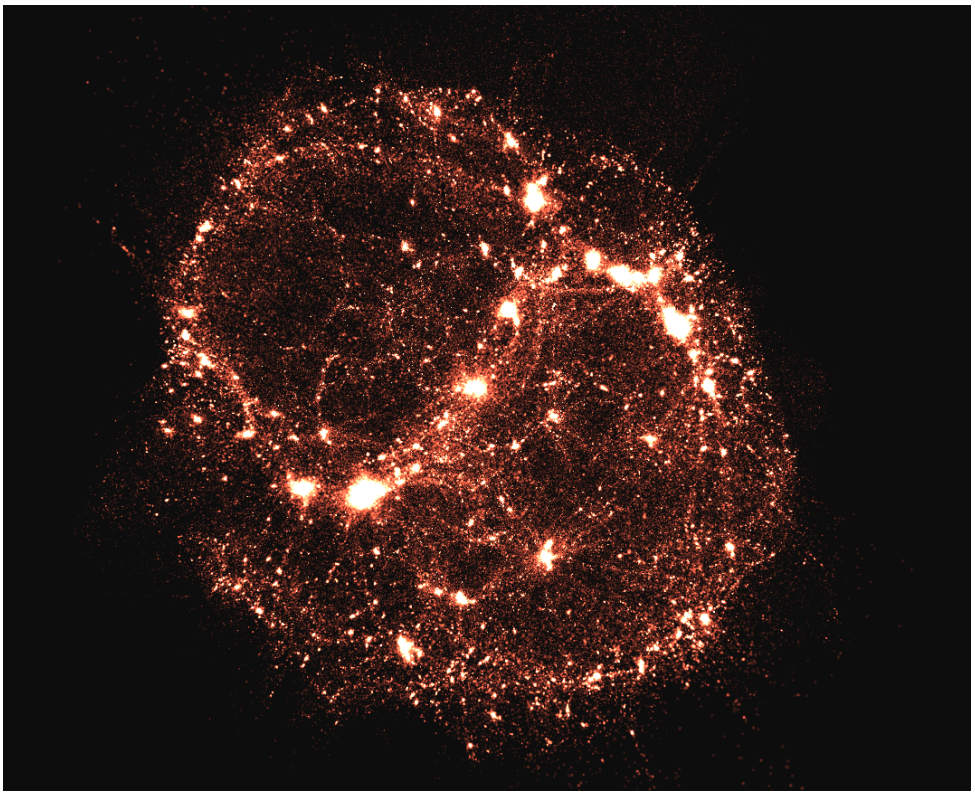


Figure 3.2: Screenshot from a simulation of 1 million bodies running on a NVIDIA RTX 500 Ada Laptop GPU at ~20 fps.

Merging leaves A common approach to reduce the memory footprint of the octree while guaranteeing shallower traversals, is to reduce the maximum tree node depth by letting the leaves contain up to L_{max} particles. Cazalbou's [4] approach was implemented in order to recursively merge groups of adjacent Morton codes into the same leaf, provided that the L_{max} threshold is preserved. It should be noted that, in this implementation, L_{max} refers to the maximum number of codes per leaf, which could be smaller than the actual number of particles that thus get assigned to that leaf. However, as mentioned earlier, switching from 32-bit to 64-bit Morton codes drastically reduced leaves overcrowding, making L_{max} a reliable proxy to control the height of the octree.

For each of the resulting coarser leaves, only their first (smaller) code is considered for the binary radix tree construction, which is also enough, together with the number of particles per code, to identify the corresponding ranges of particles.

Breadth-first layout Laying out the octree nodes contiguously in memory in breadth-first order (level by level) is of paramount importance for an efficient group traversal, while also reducing the bookkeeping required to store the nodes hierarchy by having each node only keep track of the index of its first child and its number of children.

Karras's method outputs tree nodes in strict depth-first order, so in order to achieve a breadth-first ordering, the binary radix tree nodes are sorted by depth, with a stable sort,

before the conversion to an octree. It is worth mentioning that only those radix tree nodes that map directly to valid octree nodes contribute to the depth used for sorting. This depth is thus computed in parallel for all nodes with a modified pointer jumping algorithm, capable of skipping the invalid nodes. Unfortunately, sorting the nodes while using the layout of the original algorithm leads to issues with the ordering of the leaf nodes, but surprisingly, if the leaves are treated as internal nodes and interleaved in memory with respect to the actual internal nodes, this sorting operation is able to preserve the relative arrangement imposed by the algorithm definition, without interfering with the correctness of the subsequent conversion to an octree.

This kind of layout simplifies the radix tree to octree conversion, which is done slightly differently in Cazalbou's work. Due to the level by level ordering, it is sufficient to look for the leftmost and rightmost children that directly map to valid octree nodes to determine the range of children of the parent. Additionally, vectorizing the bottom-up tree reduction used to compute the center of mass of the octree nodes becomes even more trivial.

3.2.2 Grouped tree traversal

The octree is traversed in parallel by groups of N_{group} contiguous particles, which are forced to walk along the same path with an iterative BFS (breadth-first-search), reminiscent of the approach discussed by Bédorf et al. [2]. Each group is mapped to a CUDA thread block, where each thread accumulates the forces for a single particle in the group while cooperating to evaluate the node opening criterion of the nodes in the visit queue.

The queues for all the groups are allocated in global memory, and their size is determined by the configurable amount of GPU memory to dedicate to the simulation, which gets partitioned across the individual queues. The maximum number of nodes appended to the queue can in fact increase considerably when scaling to a few millions of particles, based on the values of θ, L_{max} and N_{group}, thus having to be heuristically determined.

The nodes in the current visit queue are distributed in round-robin fashion to the available threads in the block, which can evaluate the opening criterion by using the minimum distance of the group to the node's barycenter as d (See Equation 3.1). This implies that for the same values of θ, this traversal strategy leads to a more accurate force evaluation than the standard Barnes-Hut algorithm, since it suffices that one particle in the group satisfies the node opening criterion for the entire group to visit that node.

Before being evaluated, the nodes to be approximated are buffered in shared memory until the buffer size exceeds N_{group}, while nodes that do not meet the opening criterion are immediately appended to the next visit queue. When a leaf node is reached, all pairwise interactions need to be computed, making the value of L_{max} an important choice that trades off queues memory usage against leaves evaluation speed.

A softening factor ϵ is used for implicit collision handling, as mentioned in Section 2.2.2.

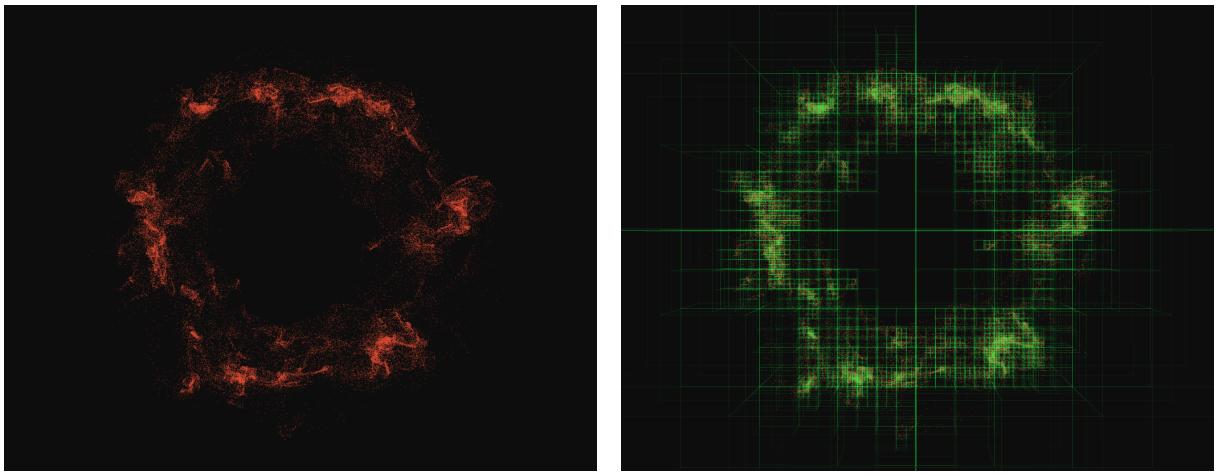


Figure 3.3: Screenshots from a simulation of a few thousands of bodies, optionally rendering the nodes of the octree.

3.2.3 Particles integration

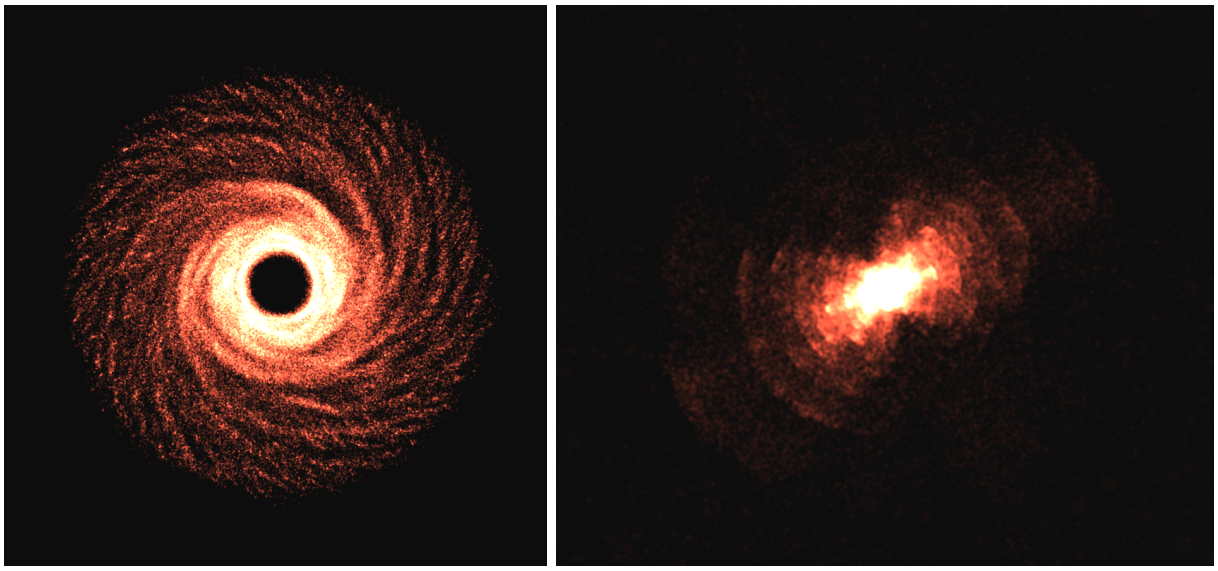
The updates to velocities and positions are done in an embarassingly parallel manner, using the Leapfrog integrator in the "kick-drift-kick" form (2.2), by launching one CUDA thread per particle, maximizing coalescing due to the SoA memory layout, thus contributing minimally to the execution time of a step of the simulation.

Reflective boundary conditions are used to maintain the particles enclosed in a cube. If the octree was instead built on the bounding-box of the particles and no boundary conditions were applied, the volume enclosed by the octree could indefinitely grow in the presence of particles moving away from each other, gradually leading to loss of precision in the Morton encoding and, consequently, to the leaves overcrowding problem previously mentioned.

3.2.4 Rendering with OpenGL

Consideration was also put to allow real-time rendering of millions of particles on screen, which are drawn as billboard spheres using additive blending, as shown in Figure 3.4. A single quad mesh is sufficient to draw a pixel perfect circle by using the distance from the center of the mesh to discard fragments that are not contained within the inscribed circle. The radius of each fragment can also be used to shade the circle to give the illusion of a sphere, and in this case it is used to scale down the alpha value, which in conjuction with additive blending creates smooth brighter spots when lots of particles are clustered together. The particles position buffers are allocated as OpenGL SSBOs (Shader Storage Buffer Objects) and mapped for read and write access by CUDA, and using instancing, a single draw call is sufficient to allow the quad mesh to be scaled and translated by the position of the particle in world space.

A simple arcball camera was also implemented so that the view can orbit around the origin, and the octree nodes are optionally rendered using the same instancing trick with a cube mesh, as shown in Figure 3.3.



(a) Self-gravitating disk *(b) Shell galaxy after cold collapse of a uniform sphere*

Figure 3.4: Screenshots from simulations rendering particles as billboard spheres with additive blending.

3.2.5 Accuracy evaluation

Self-gravitating disk To evaluate the correctness of the implementation, the particles positions were distributed by sampling a uniform angle $\phi \in [0, 2\pi]$ and a radius r with a probability density function of type:

$$p(r) = \begin{cases} Cr^{\alpha} & r \in [r_{min}, r_{max}] \\ 0 & r \notin [r_{min}, r_{max}] \end{cases}$$

where C is chosen such that $p(r < r_{max}) = 1$. The coordinates can then be computed as $x = r\cos(\phi)$, $y = r\sin(\phi)$ and $z = rh$, where h is taken from a normal distribution of zero mean and standard deviation σ. Each of these N particles is assigned a mass $m = M_{disk}/N$, while an additional particle with mass M_{star} is placed at the center of the disk with null velocity. The initial velocities for the particles in the disk are chosen such that $v_x = V\sin(\phi)$, $v_y = -V\cos(\phi)$, with $V = \sqrt{GM(r)/r}$, where $M(r)$ is the total mass of the system enclosed in the radius r.

Some screenshots of the evolution of the resulting distribution are shown in Figure 3.5.

Energy conservation and acceleration error Figure 3.6 shows the total energy of the system $(K + U)$ for a self-gravitating disk of $N = 32768$ particles, with $r_{min} = 0.03$, $r_{max} = 0.2$, $\alpha = -1.5$, $\sigma = 0.01$, $M_{star} = 10.0$, $M_{disk} = 1.0$, $G = 1.0$, $\epsilon = 0.01$, $\Delta t = 0.001$, $L_{max} = N_{group} = 32$, using single-precision. As expected, by reducing the value of θ the energy converges to that of the all-pairs implementation.

The relative step-local acceleration error $\Delta a/a$ [2] is also computed against the all-pairs

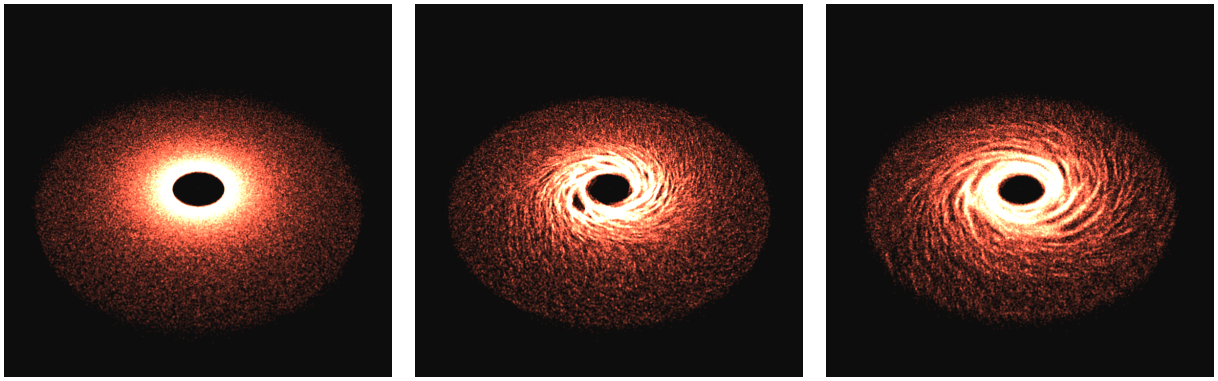


Figure 3.5: Screenshots from a simulation of a self-gravitating disks of 2^{18} particles.

implementation, for each of the N particles, and then averaged as

$$\Delta a/a = \frac{1}{N}\sum_{i}^{N}\frac{\|\mathbf{a}_i^{tree} - \mathbf{a}_i^{ap}\|}{\|\mathbf{a}_i^{ap}\|}$$

The results are shown in Figure 3.7, once again hinting at the convergence of the algorithm.

3.2.6 Performance evaluation

The average execution time slices for the octree construction and traversal, in single-precision, are shown in Figure 3.8, for the cold collapse of a uniform sphere, with $\theta = 0.75$ and $L_{max} = N_{group} = 32$. Integrating particles position requires a negligeble fraction of the total runtime of a step, hence it is not shown. All tests were run on a NVIDIA RTX 500 Ada Generation Laptop GPU (2048 CUDA cores).

As expected, traversing the tree represents the bottleneck of the simulation, and by further profiling, the traversal kernel showed to be compute bound by the fp32 pipeline, meaning that it could greatly benefit from an increase in the number of CUDA cores available.

Figure 3.9 shows the impact of the choice of θ on the average execution time of a single step of the previous simulation, for an increasing number of particles N. For most of the samples, $N * 1KB$ of memory was allocated for the traversal queues, while choosing $\theta = 0.25$ for $N \geq 17$ required bumping to $N * 2KB$ of memory. It is likely that the amount of memory allocated for traversal for the previous tests overshoots the actual amount of memory accessed, and tweaking the values of N_{group} and L_{max} could result in a more optimized memory usage, nevertheless the program was optimized for execution speed, and future developments could focus on finding a good heuristic for the size of the traversal queues, as well as moving part of the queues on the on-chip shared memory, which was not considered due to its size limitations on the testing hardware.

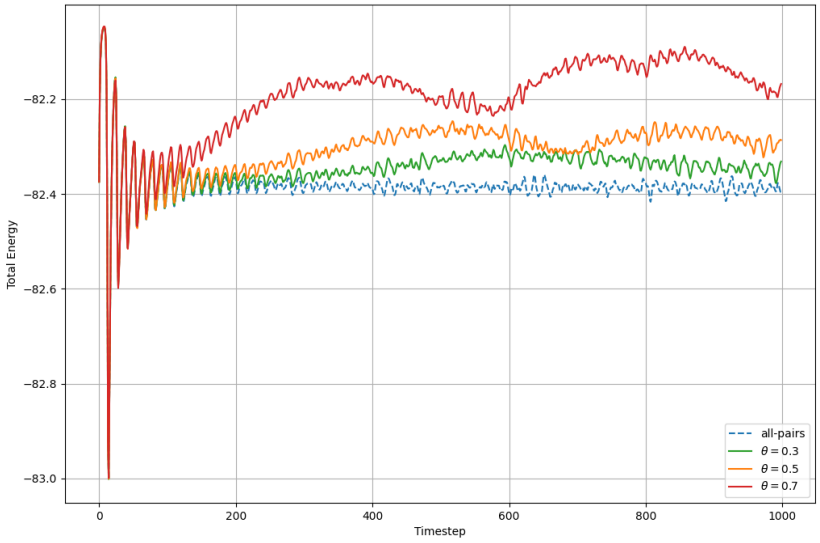


Figure 3.6: Energy conservation of a simulation of a self-gravitating disk of 32768 particles. An all-pairs (direct sum) GPU accelerated implementation is taken as baseline.

3.2.7 Code structure

The implementation is organized around a set of CUDA-enabled classes, each defined in its own `.cu` file under `src/cuda` and paired with declarations in `include/cuda`. The `Points` class encodes particle positions into Morton order and sorts them, the `Btree` class builds the binary radix tree and the `Octree` class converts it into an octree, following the approaches described in the previous sections. The `BarnesHut` class traverses the octree to compute the approximate forces, utilizing some routines from `physics_common.cu` (e.g., positions integrator), and similarly the `Validator` class uses `physics_common.cu` to optionally compare against an all-pairs reference implementation. Each class encapsulates its own device memory allocation and kernel launch logic within its implementation file. `SoABtreeNodes`, `SoAOctreeNodes` and `SoAVec3` encapsulate some Structure of Array device buffers to favour memory coalescing.

The `Simulation` class makes use of the above components in order to class orchestrates the overall simulation loop after having initialized the particle distributions using `initial_conditions.cuh`. Rendering and user interactivity are managed by a `Renderer` class, which creates and drives the `Simulation` instance while leveraging `Camera` and `ShaderProgram` classes to handle view transformations and shader management.

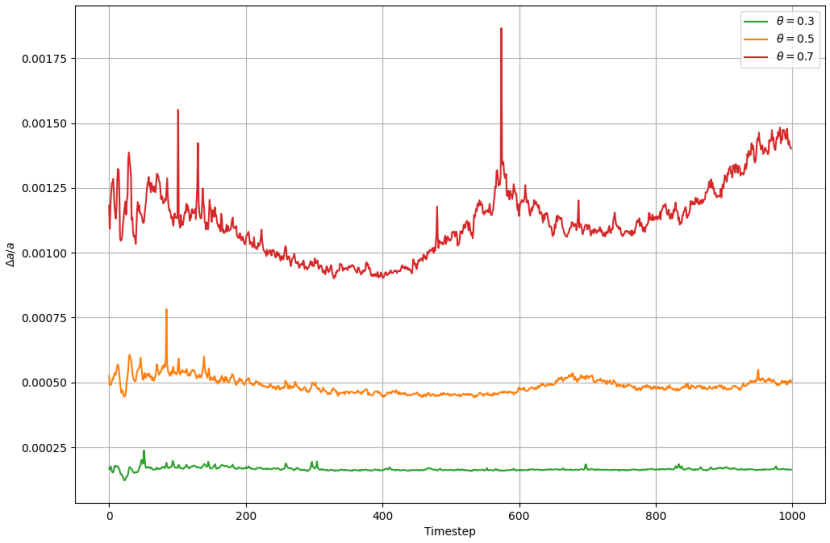


Figure 3.7: Relative step-local acceleration error of a simulation of a self-gravitating disk of 32768 particles against an all-pairs implementation.

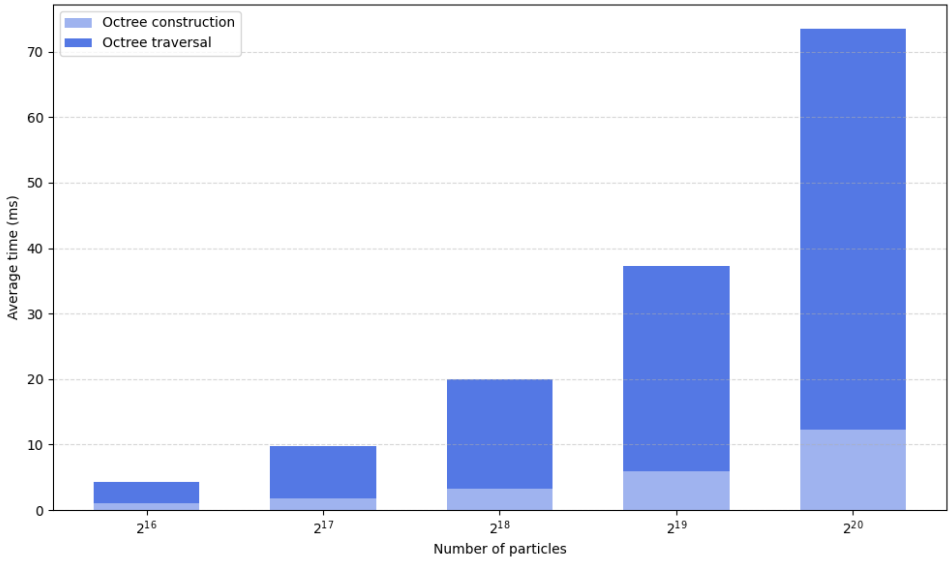


Figure 3.8: Average execution times in milliseconds for the octree construction and traversal, simulating a uniform sphere with $\theta = 0.75$ *and* $L_{max} = N_{group} = 32$.

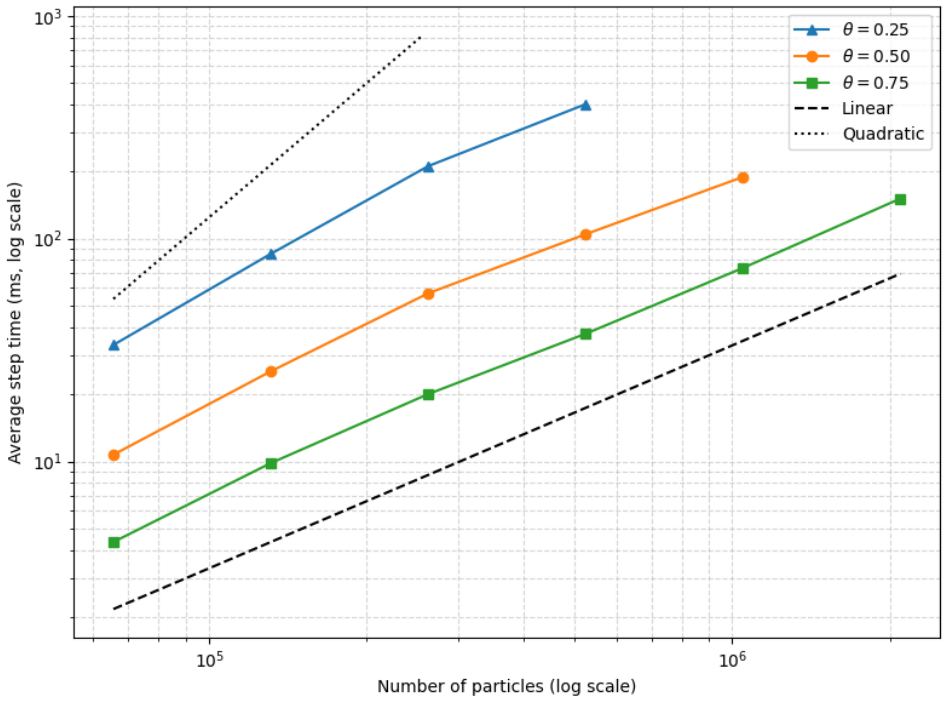


Figure 3.9: Scaling the number of particles of a uniform sphere for different values of θ, with $L_{max} = N_{group} = 32$. For $\theta = 0.25$ and $N \geq 2^{17}$ the queue size per traversal group was set to $64KB$, while $32KB$ was sufficient for the rest of the samples.

4. The Fast Multipole method

4.1 FMM General Introduction

The Fast Multipole Method (FMM) [5], pioneered by L. Greengard and V. Rokhlin in the late 1980s, is a breakthrough algorithm that exploits the mathematical properties of electrostatic and gravitational potentials in conjunction with a hierarchical divide-and-conquer approach. This allows the evaluation of N-body interactions to be performed with almost near $O(N)$ computational cost, making large-scale simulations feasible that would otherwise be intractable with direct $O(N^2)$ methods. Furthermore, the FMM provides a tunable accuracy parameter ε, which controls the precision of potential and force evaluations. The time complexity of the FMM is more precisely given by

$$O\left(N \log^{d-1}\left(\frac{1}{\varepsilon}\right)\right)$$

in d dimensions, where ε is the user-specified error tolerance.

4.1.1 Poisson Equation For 2D Newtonian Gravity

Particle interactions are frequently described by a potential field $\phi(\mathbf{x})$ generated by and depending on parameters such as position and charge or mass associated with each particle. And the total potential of a system of N particles is given by the superposition:

$$\phi(\mathbf{x}) = \sum_{i}^{N} \phi_i(\mathbf{x}) \tag{4.1}$$

In classical gravity, the interaction between masses is described by a potential field. The gravitational potential $\Phi(\mathbf{r})$ satisfies **Poisson's equation**:

$$\nabla^2 \Phi(\mathbf{r}) = 4\pi G \rho(\mathbf{r}) \tag{4.2}$$

where $\Phi(\mathbf{r})$ is the gravitational potential at position $\mathbf{r}$, G is the gravitational constant (with a value that depends on the dimension), and $\rho(\mathbf{r})$ is the mass density. In two dimensions, Poisson's equation adapts slightly:

$$\nabla^2 \Phi(\mathbf{r}) = 2\pi G \rho(\mathbf{r}) \tag{4.3}$$

So we can find out that gravity actually won't work quite like it does in our real three-dimensional universe. Typically when we simulate gravity in a flat plane we usually use the three dimensional inverse square law and restrict motion to a plane (like what we have done previously). But for this fmm method we treat it differently and compute gravity that really only lives in two dimensions, which will give a lot of benefits to the kernels of FMM.

4.1.2 Real 2D Force And Potential Energy For FMM

From previous different poisson equations, we can find that the potential and force laws differ fundamentally from their three-dimensional counterparts. For example , the electrostatic potential of a point source charge q located at $\mathbf{x}_0 = (x_0, y_0) \in \mathbb{R}^2$ is given at a point $\mathbf{x} = (x, y) \in \mathbb{R}^2$ by

$$\phi_{\mathbf{x}_0}(\mathbf{x}) = -q \ln \|\mathbf{x} - \mathbf{x}_0\| \tag{4.4}$$

while the electrostatic field is obtained by taking the negative gradient of the potential,

$$\mathbf{E}_{\mathbf{x}_0}(\mathbf{x}) = -\nabla\phi(\mathbf{x}) = q\frac{\mathbf{x} - \mathbf{x}_0}{\|\mathbf{x} - \mathbf{x}_0\|^2} \tag{4.5}$$

For every harmonic function u of two real variables, there exists a holomorphic function $w : \mathbb{C} \to \mathbb{C}$ s.t. $u = \mathrm{Re}(w)$, which is unique up to an additive constant. For this, the vector $\mathbf{x} = (x, y) \in \mathbb{R}^2$ is identified with the complex number $z = x + iy \in \mathbb{C}$, and the electrostatic potential $\phi_{\mathbf{x}_0}$, seen to be proportional to the real part of the complex logarithm, is identified with the complex potential

$$\phi_{z_0}(z) = q \ln(z - z_0), \tag{4.6}$$

to which it is related by $\phi_{\mathbf{x}_0}(\mathbf{x}) = \mathrm{Re}(-\phi_{z_0}(z))$. Both the theory and the practical implementation of the two dimensional FMM are based on this formulation in terms of complex analysis, and we will in the following refer to equation (4.6) and derived expressions as the potential.

In practical applications, the electrostatic field needs to be computed alongside the potential frequently, which necessitates establishing a connection between (4.5) and (4.6). This connection is provided by the following lemma.

Lemma 4.1 *If $u(x, y) = \mathrm{Re}(w(x + iy))$ describes the potential field at (x, y), then the corresponding force field is given by*

$$-\nabla u = -(u_x, u_y) = (-\mathrm{Re}(w'),\ \mathrm{Im}(w')) \tag{4.7}$$

where $w' = \frac{dw}{dz}$ denotes the complex derivative of w.

Codes: In our code, we represent the 2D spatial variable (x, y) as a complex number $z = x + iy$, so that the field at any evaluation point is then efficiently determined by evaluating the derivative of the complex potential expansion and the real and imaginary parts are extracted to obtain the respective vector components.

4.2 Mathematical Preliminaries

In this section we want to prepare some foundational math concepts for fmm method. We will just give the most crucial formulas that we have also implemented in codes. If wanting to dig deeper to the background proves and details, just refer the original author's paper [1].

4.2.1 Multipole Expansion

Consider now a point charge of strength q, located at z_0. A straightforward calculation shows that for any z with $|z| > |z_0|$,

$$\phi_{z_0}(z) = q \log(z - z_0) = q \left(\log(z) - \sum_{k=1}^{\infty} \frac{1}{k} \left(\frac{z_0}{z} \right)^k \right). \tag{4.8}$$

This provides us with a means of computing the multipole expansion due to a collection of charges [6].

Theorem 4.2.1 (ME)**.** *Suppose that m charges of strengths $\{q_i,\ i = 1, ..., m\}$ are located at points $\{z_i,\ i = 1, ..., m\}$, with $|z_i| < r$. Then for any z with $|z| > r$, the potential $\phi(z)$ induced by the charges is given by*

$$\phi(z) = Q \log(z) + \sum_{k=1}^{\infty} \frac{a_k}{z^k}, \tag{4.9}$$

where

$$Q = \sum_{i=1}^{m} q_i \qquad \text{and} \qquad a_k = \sum_{i=1}^{m} \frac{-q_i z_i^k}{k}. \tag{4.10}$$

Furthermore, for any $p \geq 1$,

$$\left| \phi(z) - Q \log(z) - \sum_{k=1}^{p} \frac{a_k}{z^k} \right| \leq \frac{1}{p+1} \alpha \left| \frac{r}{z} \right|^{p+1} \leq \left(\frac{A}{p+1} \right) \left(\frac{1}{c-1} \right) \left(\frac{1}{c} \right)^p, \tag{4.11}$$

where

$$c = \left| \frac{z}{r} \right|, \quad A = \sum_{i=1}^{m} |q_i|, \quad \text{and} \quad \alpha = \frac{A}{1 - \left| \frac{r}{z} \right|}. \tag{4.12}$$

Here, the amount of compression achieved depends both on the desired precision and the distance of the target from the sources.

Codes: This expasion is implemented in the file 'mutipole_expansion.hpp' and some useful coefficients are stored at 'fmm_table' and 'fmm_utility'

4.2.2 Translation Of A Multipole Expansion

Theorem 4.2.2. (M2M) *Suppose that*

$$\phi(z) = a_0 \log(z - z_0) + \sum_{k=1}^{\infty} \frac{a_k}{(z - z_0)^k} \tag{4.13}$$

is a multipole expansion of the potential due to a set of m charges $q_1, q_2, \ldots, q_m$, all of which are located inside the circle D of radius R with center at z_0. Then for z outside the circle D_1 of radius $(R + |z_0|)$ and center at the origin,

$$\phi(z) = a_0 \log(z) + \sum_{l=1}^{\infty} \frac{b_l}{z^l}, \tag{4.14}$$

where

$$b_l = -\frac{a_0 z_0^l}{l} + \sum_{k=1}^{l} a_k z_0^{l-k} \binom{l-1}{k-1}, \tag{4.15}$$

with $\binom{l}{k}$ *the binomial coefficients. Furthermore, for any* $p \geq 1$,

$$\left| \phi(z) - a_0 \log(z) - \sum_{l=1}^{p} \frac{b_l}{z^l} \right| \leq \left(\frac{A}{1 - \frac{|z_0|+R}{|z|}} \right) \left(\frac{|z_0| + R}{|z|} \right)^{p+1} \tag{4.16}$$

with A defined in (4.12).

4.2.3 Local Expansion

Theorem 4.2.3 (LE)**.** *Suppose that n charges of strengths* q_i, $i = 1, \ldots, n$ *are located at points* z_i *with* $|z_i| > r$ *for some positive* $r \in \mathbb{R}$, *then for any* $z \in \mathbb{C}$ *with* $|z| < r$, *the potential* $\phi(z)$ *is given by*

$$\phi(z) = \sum_{k=0}^{\infty} b_l z^l, \tag{4.17}$$

where

$$b_0 = \sum_{i=1}^{n} q_i \ln(-z_i), \qquad b_l = -\frac{1}{l} \sum_{i=1}^{n} \frac{q_i}{z_i^l} \quad \text{for } l \geq 0. \tag{4.18}$$

Furthermore, for any $p \geq 1$,

$$\left| \phi(z) - \sum_{k=0}^{p} b_l z^l \right| \leq \frac{A}{c-1} \left(\frac{1}{c} \right)^p, \qquad \text{where} \quad c = \left| \frac{r}{z} \right| \tag{4.19}$$

There is a symmetry between theorems 4.2.1 and theorems 4.2.3:

The multipole expansion converges outside the smallest disk containing all sources, while the local expansion converges inside the largest disk excluding all sources.

The FMM is efficient because it constructs a local expansion for each evaluation point that includes all distant sources, leaving only a small number of nearby sources to handle directly. Then, a key step should be converting a multipole expansion into a local expansion.

Code: This local expansion is implemented at file 'local_expansion.hpp'.

4.2.4 Conversion Of A Multipole Expansion Into A Local Expansion

Theorem 4.2.4 (M2L)**.** Suppose that n sources of strengths q_i, $i = 1, \ldots, n$ are located inside the disk $D_R(z_0)$ and that $|z_0| > (c+1)R$ where $c > 1$. The multipole expansion of the potential due to the sources converges inside of $D_R(0)$ and is there described by a local expansion

$$\phi(z) = \sum_{l=0}^{\infty} b_l z^l, \tag{4.20}$$

where

$$b_0 = a_0 \ln(-z_0) + \sum_{k=1}^{\infty} \frac{a_k}{z_0^k}(-1)^k, \qquad b_l = -\frac{a_0}{l z_0^l} + \frac{1}{z_0^l} \sum_{k=1}^{\infty} a_k z_0^{-k} \binom{l+k-1}{k-1} (-1)^k \quad \text{for } l \geq 1. \tag{4.21}$$

Furthermore, for any $p \geq \max(2, \frac{2c}{c-1})$,

$$\left| \phi(z) - \sum_{l=0}^{p} b_l z^l \right| < \frac{A(4e(p+c)(c+1)+c^2)}{c(c-1)} \left(\frac{1}{c}\right)^{p+1}, \tag{4.22}$$

with A as defined in (4.12) and e denotes Euler's number.

Similar to the multipole expansion case, the FMM also requires combining local expansions centered at different points. The following lemma provides this result.

4.2.5 Translation Of A Local Expansion

Theorem 4.2.5 (L2L)**.** Let the local expansion of the potential due to a set of n charges of strengths q_i, $i = 1, \ldots, n$, located outside of the disk $D_R(z_0)$ be given by (3.24). Then for $R > |z_0|$ and $z \in D_{R-|z_0|}(0)$,

$$\phi(z) = \sum_{l=0}^{\infty} a_l z^l, \quad \text{where} \quad a_l = \sum_{k=l}^{\infty} b_k \binom{k}{l} (-z_0)^{k-l}. \tag{4.23}$$

Furthermore, for any $p \geq 1$,

$$\left| \phi(z) - \sum_{l=0}^{p} a_l z^l \right| \leq \frac{A}{\frac{R-|z_0|}{|z|} - 1} \left(\frac{1}{\frac{R-|z_0|}{|z|}} \right)^{p+1}, \tag{4.24}$$

with A as defined in (4.12). The translation formula (4.23) is exact. When translating a series truncated to p-th order, the summation for a_l contains at most p terms and the shifted local expansion enjoys the same error bound as the original expansion. An efficient algorithmic realization of (4.23) is possible via the Horner scheme [1].

4.3 Algorithm Details Description

In this chapter we want to show how can we arrive the optimal complexity version, by leveraging the knowledge from last section. Firstly let's look at the $N \log N$ version which shared similar ideas with previous Barned-Hut method.

4.3.1 The O(NlogN) Complexity Algorithm

To efficiently account for groups of particles using multipole expansions, the computational domain is hierarchically partitioned into a regular grid of boxes at multiple refinement levels.

Refinement level 0 consists of the entire domain. Each subsequent level $l + 1$ is obtained by subdividing each box at level l into 2^d equal parts (its children), resulting in a quadtree for $d = 2$ and an octree for $d = 3$. The following definitions regarding the relationships between boxes are essential for the method:

Definition 4.3.1 (near neighbours)**.** Two boxes are said to be **near neighbours** if they are at the same refinement level and share a boundary point. (A box is a near neighbour of itself.)

Definition 4.3.2 (well separated)**.** Two boxes are said to be **well separated** if they are at the same refinement level and are not near neighbours.

Definition 4.3.3 (interaction list)**.** With each box i is associated an **interaction list**, consisting of the children of the near neighbours of i's parent which are well separated from box i.

The definition of the interaction list as shown in Figure 4.1 ensures that multipole expansions are evaluated at the coarsest valid level, optimizing computational efficiency. For a box b at level ℓ, the interaction list consists of the children of the near neighbours of b's parent that are well separated from b. This guarantees that contributions from distant sources are handled by multipole approximations, while nearby interactions are computed directly or at finer levels. Therefore, the $N \log N$ verison FMM algorithm can be summarized as follows:

1. Choose the expansion order p for a given accuracy ε.
2. Recursively subdivide the computational domain into a hierarchical tree of boxes (e.g., quadtree or octree), halting at $\sim \log N$ levels.
3. At each level, compute p-th order multipole expansions for the sources within each box.
4. To evaluate the potential at a point z, traverse the tree from the root to the leaf containing z, accumulating contributions from the interaction lists at each level, and finally sum the direct contributions from sources in near neighbours at the leaf level.

This procedure ensures that the total computational cost for constructing and evaluating the expansions is $O(N \log N)$, since the tree has $O(\log N)$ levels and the size of each interaction list is bounded.

4.3.2 The O(N) Comlexity Fully Optimal FMM

The $O(N \log N)$ Fast Multipole Method (FMM) can be **improved to an $O(N)$ algorithm via two main observations:**

- **Upward Pass:** Multipole expansions are first formed at the leaf level for each box, and then recursively shifted and combined up the tree to parent boxes. This reduces the cost of constructing all expansions to $O(N)$.

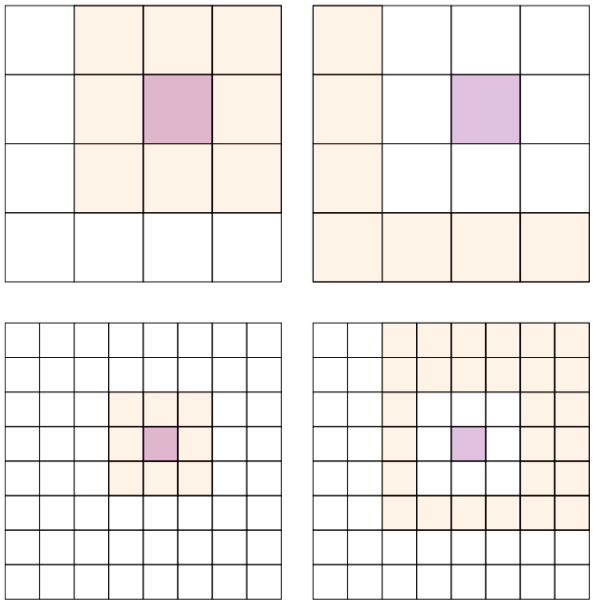


Figure 4.1: Near neighbours (left 2) and interaction lists (right 2) for a box at refinement levels two (top 2) and three (bottom 2)

- **Downward Pass:** Rather than evaluating multiple multipole expansions for each evaluation point, the expansions from boxes in a given box's interaction list are converted into a single local expansion around the box's center. This local expansion is then shifted and combined downward through the tree. As a result, each evaluation point requires only one local expansion to be evaluated, plus direct computation for sources in near neighbour boxes.

Actually it's not hard to find that these two passes share the same ideas as the **restriction operator** R and the **prolongation operator** R^T from the terminology of multigrid methods. In summary, it leads to fast $O(N)$ actually by low rank approximation plus hierarchical decomposition.

The resulting FMM algorithm consists of three main phases:

1. **Initialization:** Select refinement levels and expansion order for the desired accuracy, and build the hierarchical tree.
2. **Upward Pass:** Construct multipole expansions at the leaves and combine them upward through the tree.
3. **Downward Pass:** Convert multipole expansions in each interaction list into local expansions, shift these downward, and sum them at each box.

After the downward pass, each leaf box contains a local expansion that represents the potential from all distant sources. Nearby sources (bounded in number) are handled directly. This approach ensures the total computational cost is $O(N)$.

4.3.3 The Parallel FMM Algorithm With OpenMP

The Fast Multipole Method (FMM) is well suited for parallelization due to the following properties:

- **Upward and downward passes** can be processed in parallel across all boxes at each refinement level, since each box only depends on information from its children (upward) or its parent and interaction list (downward). However, different levels must be processed sequentially.
- **Potential and force evaluations** are fully parallelizable after all expansions are constructed, since they involve only read-only operations on local and multipole expansions.

The sequential implementation was parallelized by using OpenMP version 4.5.

4.4 Experiments: Accuracy And Efficiency Tests

In this section, we will present the results of evaluating our implementation respect to both accuracy and performance. All the tests are based on a same uniform squared initial configurations with universe's size 10*10 and velocity all 0, as shown in the below image.

All measurements displayed in this section were taken on an *AMD Ryzen 7 6800H* processor with Radeon Graphics, running at 3.20 GHz. This CPU features *8 cores* and *16 threads*.

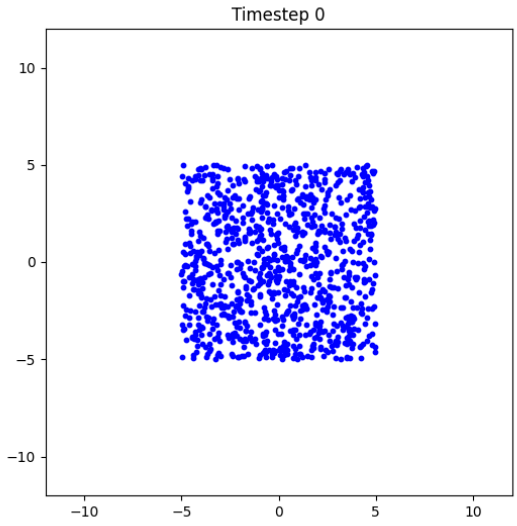


Figure 4.2: Initial Data Setup For All The Following Test

4.4.1 Accuracy

We will use particle's potential energy difference between FMM and direct sum method as a criterion. The absolute total error $E_{\text{abs,tot}}$ is quantified by the 2-norm of the deviation between the direct (double precision) solution vector q and the approximate solution vector $\tilde{q}$,

$$E_{\text{abs,tot}} = \|q - \tilde{q}\|_2 = \sqrt{\sum_i (q_i - \tilde{q}_i)^2}, \tag{4.25}$$

while the relative total error $E_{\text{rel,tot}}$ is defined by

$$E_{\text{rel,tot}} = \frac{\|q - \tilde{q}\|_2}{\|q\|_2} = \frac{\sqrt{\sum_i (q_i - \tilde{q}_i)^2}}{\sqrt{\sum_i q_i^2}}. \tag{4.26}$$

For the potential, the sum goes over the indices $i = 1, \ldots, N$. The maximum component errors are defined as

$$E_{\text{abs, max}} = \|q - \tilde{q}\|_\infty = \max_i |q_i - \tilde{q}_i|, \tag{4.27}$$

$$E_{\text{rel, max}} = \max_i \frac{|q_i - \tilde{q}_i|}{|q_i|}, \tag{4.28}$$

Furthermore, the letters N, s and ε are used to denote the input parameters of the FMM, i.e., particle number, max items per leaf and desired accuracy, etc.

Accuracy Vs Time Integration

In this test we want to see if our method can keep stable with a long timestamps.

N	Items/leaf	d	Extent	dt	n_{steps}	$\varepsilon_{\text{fault}}$	$\varepsilon_{\text{soften}}$
1000	32	2	10.0	0.0001	10000	0.01	0.01

Table 4.1: Simulation parameters for this accuracy vs time testing.

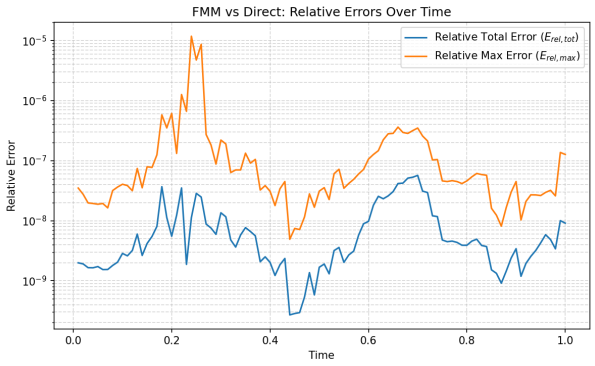


Figure 4.3: Test the two kinds of relative errors over 1 s

From the plot we can find even for the collapsing and explosion time interval $0.2s - 0.3s$, the error is still controlled below 10^{-5}. So we can say our method is very accurate over a long time interval and ever-changing distributions.

Accuracy Vs Fault Tolerance Epsilon

For this test, we want to see how our initial maximum fault tolerance value ϵ_{fault} will affect the accuracy curve.

N	Items/leaf	d	Extent	$\varepsilon_{\text{fault}}$	$\varepsilon_{\text{soften}}$
1000	32	2	10.0	1/0.1/0.01/0.001/0.0001/0.00001	0.01

Table 4.2: Simulation parameters for accuracy vs fault tolerance epsilon.

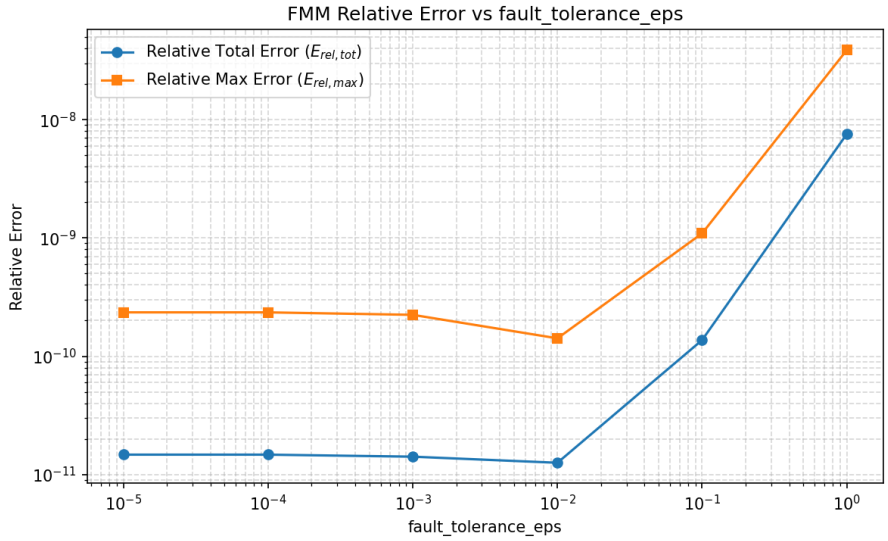


Figure 4.4: Test the relative errors for different falut tolerance eps at timestep=0

From the result, we can find that the elbow value is 0.01 for this kind of setup. If $\epsilon_{fault} > 0.01$, then the error will increase.

Accuracy Vs MaxItemsPerLeaf

This MaxItemsPerLeaf is one of the stopping criterion of fmm method by controlling the max depth of branches. It will affect both the accuracy and performance.

N	Items/leaf	d	Extent	$\varepsilon_{\text{fault}}$	$\varepsilon_{\text{soften}}$
1000	4/8/16/32/128/256	2	10.0	0.01	0.01

Table 4.3: Simulation parameters for accuracy vs max items per leaf.

If this value is large, it means the tree is shallow and most particles don't leverage the fmm approximation at all, so that we degerate to direct sum method. The accuracy should be better but the efficiency should be worse. For small value, it's vice verse. Let's check if the test results conform our theoretical expectation.

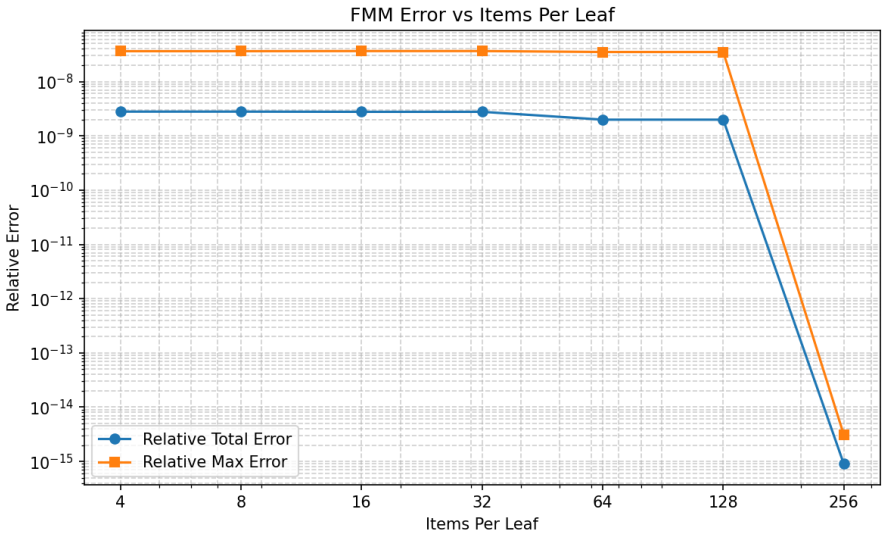


Figure 4.5: Test the relative errors for different max items per leaf at timestep=0

From the curve, we can say the test results coincide with theory.

Spatial Error Distribution Visualization

As a final note on accuracy, it is informative to examine the spatial distribution of FMM errors, not just their magnitudes. We divide the universe to 200*200 grids and sum all the total error over each grid. The figure 4.6 shows contour plots of the relative potential errors across grids for this kind of uniform particle distributions.

N	Items/leaf	d	Extent	$\varepsilon_{\text{fault}}$	$\varepsilon_{\text{soften}}$
1000	32	2	10.0	0.01	0.01

Table 4.4: Simulation parameters for spatial error plot.

From the plot, higher error regions are observed near box (node of quad tree) boundary lines and the highest errors are founded in box corners.

This pattern is likely related to the convergence properties of the local expansion, where the largest errors occur farthest from the box center.

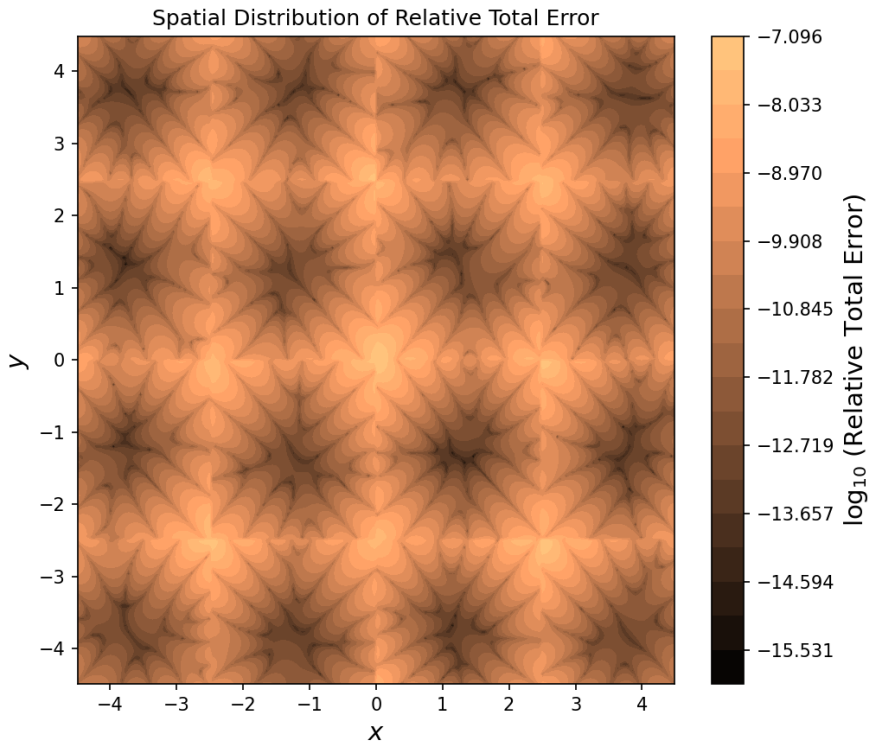


Figure 4.6: Relative total potential errors for square uniform distribution.

4.4.2 Performance

In this section we will check the time efficiency for our method. We still use leapfog integration method. From statistic view, we should test 10-100 steps and use the average step time as a criterion. Also we should use small Δt like 0.0001 to let the distribution of system keep consistent without collapsing or exploding, otherwise our curve is unstable like a zigzag.

FMM's Time Complexity

From previous theory part we demonstrate that we can achieve a nearly O(N) complexituy version. So we should firstly verify this. By using the following parameters we get this curve:

N	Items/leaf	d	Extent	dt	n_{steps}	$\varepsilon_{\text{fault}}$	$\varepsilon_{\text{soften}}$
500/2000/8000/32000/128000	32	2	10.0	0.0001	100	0.01	0.01

Table 4.5: Simulation parameters for this complexity testing.

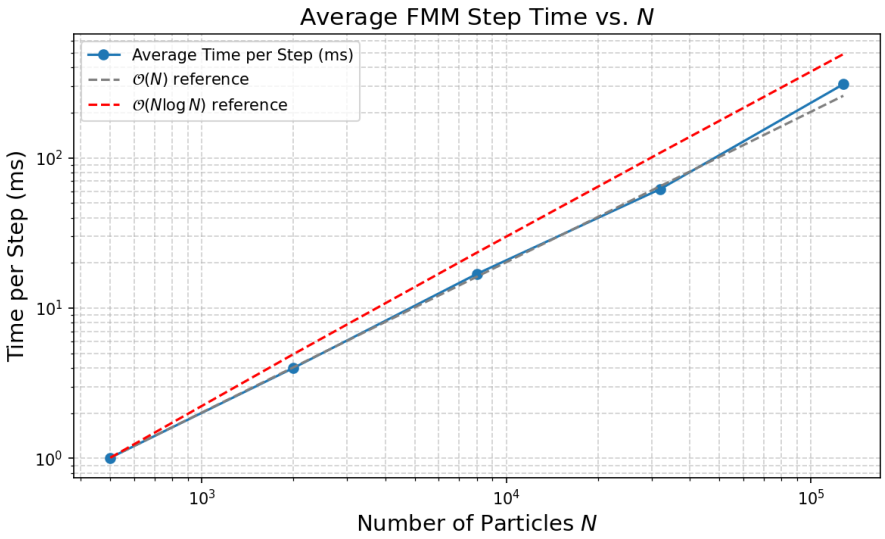


Figure 4.7: Verify the complexity of our method by comparing with O(N) and O(logN)

From the above curve we can say our algorithm complexity is about $O(pN)$ and p is very small integer so that we almost reach the optimal complexity!

Time Efficiency Vs Fault Tolerance Epsilon

From last section we know the parameter ϵ_{fault} can affect the accuracy. Now we want to test its effect to efficiency.

N	Items/leaf	d	Extent	dt	n_{steps}	$\varepsilon_{\text{fault}}$	$\varepsilon_{\text{soften}}$
8196	32	2	10.0	0.0001	100	$10^{-1}/10^{-3}/10^{-5}/10^{-7}/10^{-9}$	0.01

Table 4.6: Simulation parameters for fault tolerance efficiency testing.

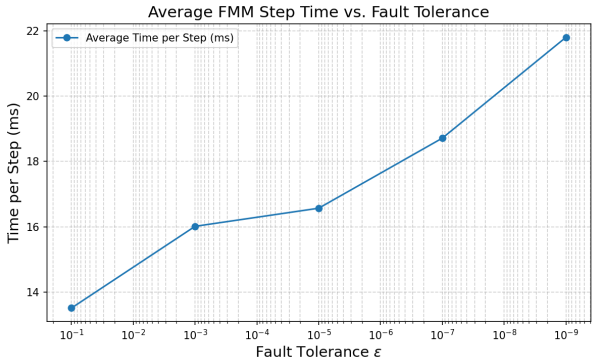


Figure 4.8: Efficiency test for different fault tolerance value.

Time Efficiency Vs Max Items Per Leaf

Finally, we want to test the influence from the parameter: 'Max Items Per Leaf' to see when the body number is fixed, which value is best suitable.

N	Items/leaf	d	Extent	dt	n_{steps}	$\varepsilon_{\text{fault}}$	$\varepsilon_{\text{soften}}$
8196	16/32/64/128/256/512	2	10.0	0.0001	100	0.01	0.01

Table 4.7: Simulation parameters for max items per leaf efficiency testing.

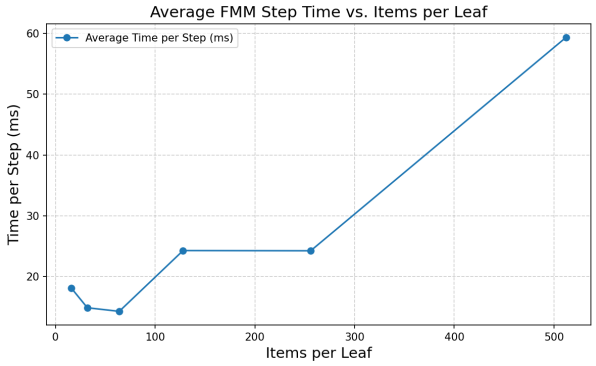


Figure 4.9: Efficiency test for different max items per leaf value when N is fix.

From the results we find that neither too small value or too large value is suitable. The best value should take both the depth of tree and max items per leaf that will be performed direct sum into consideration and then find a balance between them.

5. Comparisons

Cross Comparison Of All The Openmp Version Methods

Finally we use the previous same uniform distribution to test the three different versions of openpm parallel method to see the performance different.

Here are each method's crucial parameters setting: (common parameters: N [2000, 4000, 8000, 16000], $\Delta t = 0.0001$, steps = 100, $\epsilon_{soften} = 0.01$)

- Barnes-Hut: $\theta = 0.5$ and $maxleaves = 32$;
- FMM: $\epsilon_{fault} = 0.01$ and $itemsperleaf = 32$

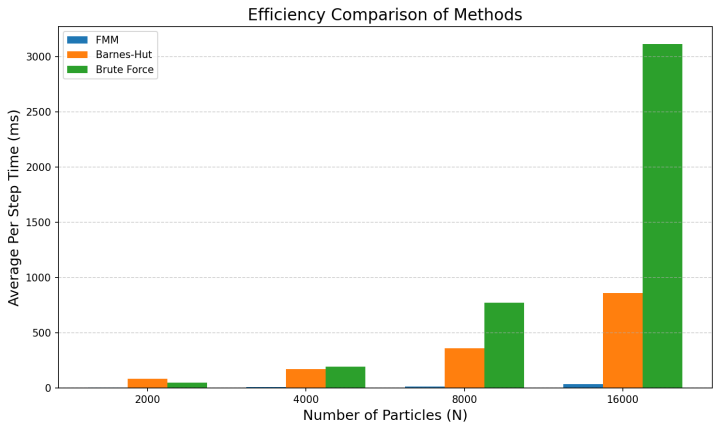


Figure 5.1: Compare different openmp versions methods performance

From the result we can find fmm always has a big advantage. And when the particles number scales up the brute fore method has very bad performance. However, for openmp barnes-hut it's acceptable by leveraging the benefits of quad tree structure and approxiamtion.

Comaparison Of Two CUDA Versions

We can now compare different algorithms using the same CUDA-based implementation. Starting with the Barnes-Hut method versus the naive all-pairs approach, it's clear that Barnes-Hut offers significantly better scaling behavior, as expected.

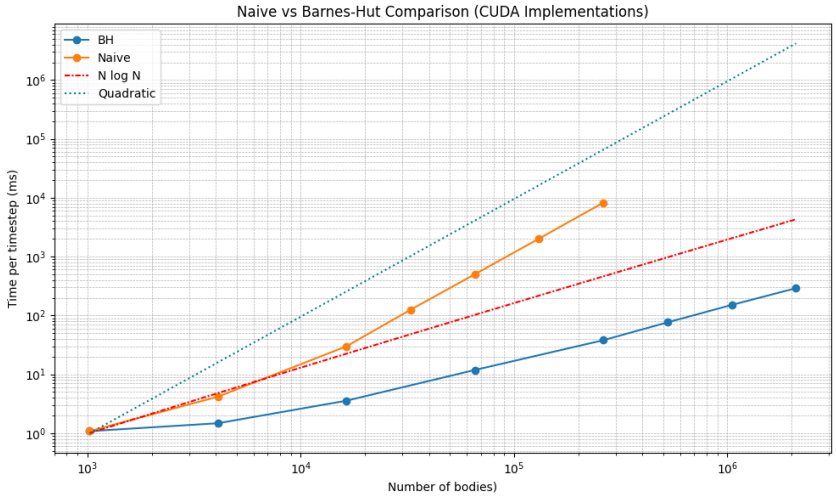


Figure 5.2: Note that the Naive implementation plot has been shifted to start from the same point as BH, but the scaling is still worse

Scaling beyond this number of particles renders the naive method impractical, as its computational cost becomes prohibitively high and execution time grows rapidly. This highlights the need for more efficient algorithms like Barnes-Hut when simulating large systems, even when running on GPU.

6. Conclusions and Future Works

In this project, we explored and compared various approaches to the N-body problem, considering both algorithmic strategies and hardware implementations. Our analysis clearly demonstrated the significant advantage of GPU-based solutions in handling massively parallel computations, where hundreds of thousands of particle interactions can be computed simultaneously with high efficiency. Furthermore, we observed that incorporating approximation techniques, such as the Barnes-Hut algorithm, can further accelerate simulations by reducing computational complexity while still maintaining acceptable accuracy when parameters are carefully tuned. These findings highlight the importance of selecting both the right numerical method and the appropriate hardware architecture to achieve scalable and accurate simulations in large-scale particle systems.

In this project, we did not explicitly address collision handling, which is a complex and numerically sensitive aspect of N-body simulations. Several strategies exist to mitigate this issue: adaptive time stepping reduces the integration interval dynamically during close encounters; regularization methods apply coordinate transformations to handle singularities analytically; and more sophisticate methods that rely on collision simulation (for example perfectly inelastic collisions). Each of these approaches comes with trade-offs in terms of physical accuracy and computational cost. Additionally, we did not explore alternative force models such as the Lennard-Jones potential, which is widely used in molecular dynamics and includes a repulsive term to naturally prevent particles from colliding.

Bibliography

[BG97] R. Beatson and Leslie Greengard. "A short course on fast multipole methods". English. In: *Wavelets, Multilevel Methods, and Elliptic PDEs*. Ed. by M. Ainsworth et al. Numerical Mathematics and Scientific Computation. Oxford University Press, 1997, pp. 1–37.

[BGP12] Jeroen Bédorf, Evghenii Gaburov, and Simon Portegies Zwart. "A sparse octree gravitational N-body code that runs entirely on the GPU processor". In: *Journal of Computational Physics* 231.7 (2012), pp. 2825–2839. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2011.12.024. URL: https://www.sciencedirect.com/science/article/pii/S0021999111007364.

[BH86] J. Barnes and P. Hut. "A hierarchical $O(N \log N)$ force-calculation algorithm". In: *Nature* 324 (1986), pp. 446–449. URL: https://www.nature.com/articles/324446a0.

[Caz+24] Robin Cazalbou et al. "Hybrid Multi-GPU Distributed Octrees Construction for Massively Parallel Code Coupling Applications". In: *Proceedings of the Platform for Advanced Scientific Computing Conference*. PASC '24. Zurich, Switzerland: Association for Computing Machinery, 2024. ISBN: 9798400706394. DOI: 10.1145/3659914.3659928. URL: https://doi.org/10.1145/3659914.3659928.

[CGR88] J. Carrier, L. Greengard, and V. Rokhlin. "A Fast Adaptive Multipole Algorithm for Particle Simulations". In: *SIAM Journal on Scientific and Statistical Computing* 9.4 (1988), pp. 669–686. DOI: 10.1137/0909044. URL: https://doi.org/10.1137/0909044.

[GR87] L. Greengard and V. Rokhlin. "A fast algorithm for particle simulations". In: *Journal of Computational Physics* 73.2 (1987), pp. 325–348. ISSN: 0021-9991. DOI: 10.1016/0021-9991(87)90140-9. URL: https://www.sciencedirect.com/science/article/pii/0021999187901409.

[Hay95] Wayne Hayes. "Efficient Shadowing of High Dimensional Chaotic Systems with the Large Astrophysical N-body Problem as an Example". MA thesis. University of Toronto, 1995. URL: https://www.cs.toronto.edu/~wayne/research/thesis/msc/hayes-95-msc.pdf.

[Kar12] Tero Karras. "Maximizing parallelism in the construction of BVHs, octrees, and k-d trees". In: *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*. EGGH-HPG'12. Paris, France: Eurographics Association, 2012, pp. 33–37. ISBN: 9783905674415. URL: https://dl.acm.org/doi/pdf/10.5555/2383795.2383801.

[NHP09] Lars Nyland, M. Harris, and Jan Prins. “Fast N-body simulation with CUDA”. In: *GPU Gem, Vol. 3* (Jan. 2009), pp. 677–695. URL: https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-31-fast-n-body-simulation-cuda.

[NVI] NVIDIA. *CUDA Toolkit Documentation*. URL: https://docs.nvidia.com/cuda/.

[OMP] OMP. *OpenMP Website*. URL: https://www.openmp.org/.

[Wik] Wikipedia. URL: https://en.m.wikipedia.org/wiki/Leapfrog_integration#.