Development of High–Performance Methods for Modeling Gravitational Interaction on Large Scales

Roman Naumenko
Faculty of Applied Sciences
Ukrainian Catholic University
L'viv, Ukraine
roman.naumenko@ucu.edu.ua

Anastasiia Beheni
Faculty of Applied Sciences
Ukrainian Catholic University
L'viv, Ukraine
anastasiia.beheni@ucu.edu.ua

Olesia Omelchuk
Faculty of Applied Sciences
Ukrainian Catholic University
L'viv, Ukraine
olesia.omelchuk@ucu.edu.ua

Sofiia Folvarochna
Faculty of Applied Sciences
Ukrainian Catholic University
L'viv, Ukraine
sofiia.folvarochna@ucu.edu.ua

Mykhailo Moroz
Lead 3D Research Engineer
ZibraAI
Kyiv, Ukraine
michael08840884@gmail.com

***Abstract*—The project demonstrates the implementation of high-performance methods for modeling gravitational interaction in a dynamic system with a large number of homogeneous particles in three-dimensional space. To optimize the calculations, we approximated the task's solution using the Barnes-Hut algorithm. It groups nearby bodies and approximates them as a single body.**

For effective parallel building of the BVH tree that minimizes divergence and maximizes occupancy, we used Linear BVH construction that first chooses the order in which the leaf nodes are stored in the tree and then generates the internal nodes in a way that respects this order. We choose that order using the space-filling z-order curve that associates each particle with the corresponding Morton code depending on its position in space. Using sorted with Bitonic sort Morton codes, we are able to build a BVH tree processing each node in parallel.

To maximize and utilize the possible substantial gains for the mentioned algorithms in a parallel setting, we perform all the calculations on the Graphics Processing Unit (GPU). We used Open Graphics Library (OpenGL) and the compute shaders written in Graphics Library Shader Language (GLSL) to interact with the GPU.

***Index Terms*—N-body simulation, OpenGL, compute shaders, GLSL, Barnes-Hut Algorithm, BVH Tree, Bitonic Sort, Morton Codes, Z-order curve, GPU**

I. Introduction

Direct computations of gravitational interactions between particles require a number of calculations proportional to the square of the number of particles. For the effective solution of this problem for millions of particles, not only a large computation power is needed, but also methods that make it possible to speed up the solution. Such a task is usually solved on large clusters [1] of computers with special algorithms that allow simulating billions of particles. The trivial approach of calculating the gravitational interactions of each particle with every other requires an unacceptably large computation power.

This project is dedicated to implementing high-performance methods for solving this task in three-dimensional space. It shows the level of acceleration we achieved using the Barnes-Hut algorithm on the way to simulating gravitational interaction for one million homogeneous particles. This work mainly concentrates on the methods of effective BVH tree construction, which is the basis of the Barnes-Hut algorithm.

II. Overview of approaches

A. Brute-force algorithm

The brute-force algorithm has $O(n^2)$, time complexity, where n is the number of particles because it involves computing the gravitational forces between all pairs of particles and updating their positions and velocities over small–time steps. Implementing it is simple, but more efficient approaches for large-scale simulations exist.

B. Particle mesh

Particle Mesh (PM) is another computational method for determining the forces in a system of particles. A system of particles is converted into a grid of density values, and forces are applied to each particle based on what cell it is in and where in the cell it lies.

The potential energy of each cell can be determined from the differential form of Gauss's law, which gives rise to the Poisson equation that is easily solved after applying the Fourier transform:

$$\nabla^2\phi = 4\pi G\rho,$$

where ϕ is the gravitational potential, G is the gravitational constant, and ρ is the local mass density.

Then, we compute for each body the resulting gravitational force:

$$\mathbf{g} = -\nabla\phi.$$

PM does not model close interaction between particles well, but it is much faster than a trivial approach [2], [3]

C. P3M

In addition to the PM calculation, P3M (P^3M) uses a straight particle-particle sum between nearby particles. It is more difficult to implement but does not have better accuracy than the Barnes-Hut algorithm [3].

D. Barnes-Hut algorithm

The algorithm stores groups of particles in a BVH tree. Each leaf node represents a single particle. Each internal node represents the part of the three-dimensional space and stores the center-of-mass and the total mass of all its children particles that are in that part. If the group is sufficiently far away, we can approximate its gravitational effects using its center of mass. If two bodies have positions (x_1, y_1) and (x_2, y_2), and masses m_1 and m_2, then their total mass m and center of mass (x, y) are given by [4]:

$$m = m_1 + m_2$$
$$x = \frac{x_1 * m_1 + x_2 * m_2}{m}$$
$$y = \frac{y_1 * m_1 + y_2 * m_2}{m}$$

III. Algorithm Parallelization

We decided to choose Barnes-Hut as it is easily understandable from a physical point of view, and we could concentrate on the parallelization we want to master through this project. Effective tree construction is essential because, ideally, we need to reconstruct the tree in every frame as the particles move.

A. Morton codes

All objects are sorted along a space-filling Z-order curve (Fig. 1) to locate them close to each other in 3D space and reside nearby in the hierarchy of the tree.

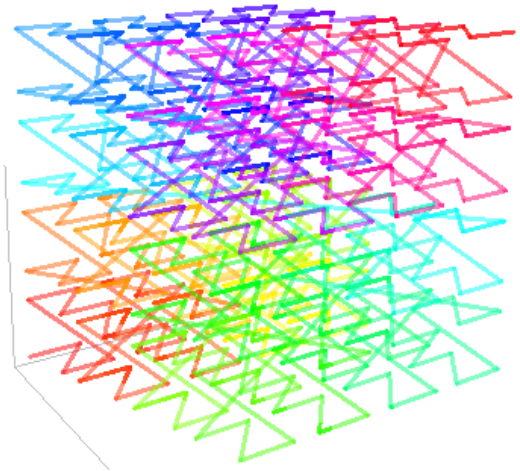

Fig. 1. Z-order curve [5].

The Z-order curve is defined in terms of Morton codes (Fig. 2). First, we take the fractional part of the binary fixed-point representation of the normalized coordinates of the given 3D point (each coordinate is in the range [0,1]) and expand it by inserting two gaps after each bit. The next step is to interleave the bits of all three coordinates together to form a single binary number, assign them to each object, and sort these objects accordingly. In this way, we effectively step along the Z-order curve in 3D. For this project, we used 30-bit Morton code.

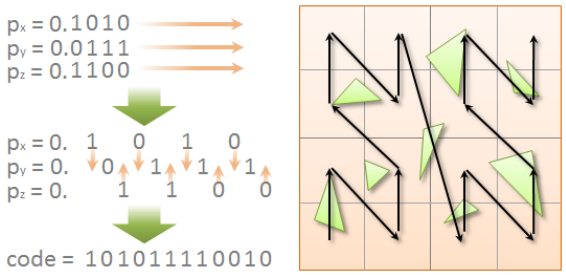

Fig. 2. Calculating Morton codes and 2D representation of Z-order curve [6].

B. Bitonic sort for Morton codes

To sort Morton codes, we use a bitonic sorting network (a sequence of compare-and-swap operations), as it maps well onto parallel hardware and has constant and relatively low–performance complexity $O(nlog^2(n))$.

To show how the bitonic sort works, a diagram of a relatively small sorting network is shown in Fig. 3.

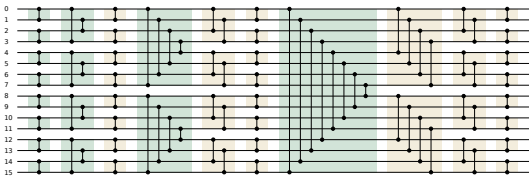

Fig. 3. Sorting diagram [7].

Each worker thread performs a compare-and-swap operation on the pair of sortable elements with indices marked with the vertical line's start and end points. Each execution step is represented as a block of a different color and follows a distinct pattern for how each worker thread grabs pairs of sortable elements:

1) Green block (flip): doubles in height until it reaches a total number of elements. Each sortable element index pair can be labeled as $T_{0,\ldots,t}$, where t represents the worker thread index to which the pair is assigned:
$$T_t = [t : h - t - 1], \text{ for } t = 0, \ldots, h/2.$$
2) Yellow block (disperse): after each green block, there is a cascade of yellow blocks, each half the previous height, until its height spans only two sortable elements. Each sortable element index pair can be labeled as $T_{0,\ldots,t}$, where t represents the worker thread index to which the pair is assigned:
$$T_t = [t : t + h/2], \text{ for } t = 0, \ldots, h/2.$$

The next step is to define those rules for the total number of sortable elements n.

1) The rule for a row of green block elements:

$$T_t = [q{+}(t \bmod \frac{h}{2}) : q{+}h{-}(t \bmod \frac{h}{2}){-}1], \text{ for } t = 0, \ldots, n/2$$

2) The rule for a row of yellow block elements:

$$T_t = [q{+}(t \bmod \frac{h}{2}) : q{+}(t \bmod \frac{h}{2}){+}\frac{h}{2}], \text{ for } t = 0, \ldots, n/2,$$

In the expressions above, $(t \bmod \frac{h}{2})$ limits the output for the term for any given t to $0, \ldots, h/2$; $q = (\lfloor \frac{2t}{h} \rfloor) * h$ represents the offset.

The algorithm relies on the concept of a bitonic sequence, which is a sequence that first increases and then decreases or vice versa. We recursively sort the input sequence into a bitonic sequence, then repeatedly merge pairs of bitonic sequences to form larger ones. Because of this, the algorithm continues to establish partial order among the elements, and then the entire sequence becomes sorted.

C. Building tree

Tero Karras first suggested the algorithm we use for parallel radix tree construction in 2012 [6]. We used this approach as a base for constructing our BVH tree. The approach associates a Morton code with each particle, arranging these codes in a sorted manner and creating a hierarchical tree structure where each subtree corresponds to a consecutive range of sorted particles. This sorting mechanism effectively groups together primitives that are spatially close to each other in three dimensions, ensuring their proximity within the resulting tree. Utilizing the fact that any binary tree with N leaf nodes always has exactly $N - 1$ internal nodes, we can generate the entire hierarchy, as illustrated by the pseudocode in Algorithm 1. The algorithm begins by creating an array of N-1 internal nodes and processes them concurrently. Each thread performs the following steps: first, it identifies the object range associated with its node using a clever technique. Then, it proceeds with the regular range-splitting process. Lastly, it assigns children to the node based on their respective sub-ranges. If a sub-range contains only one object, the child node is a leaf, and the corresponding leaf node is directly referenced. Otherwise, another internal node from the array is referenced. An example can be seen in Fig. 4.

D. Traversing tree

To calculate the total force on the exact particle A, we have to traverse the constructed tree starting from the root:

1) If the current node is a leaf node (represents the exact particle) $\rightarrow$ calculate its force on A
2) If the current node is an internal node (represents a group of particles) $\rightarrow$ calculate the ratio s/d (s – the longest side of the bounding box of that internal node, d – the distance between A and the center-of-mass of the internal node) and compare it with the threshold θ (we take $\theta = 0.5$ as it is commonly used in practice):
   - if $s/d < \theta$ (the internal node is sufficiently far away from A to be considered as a single body): take the total mass of that internal node and calculate its force on A
   - otherwise, recursively traverse each of the current node's children

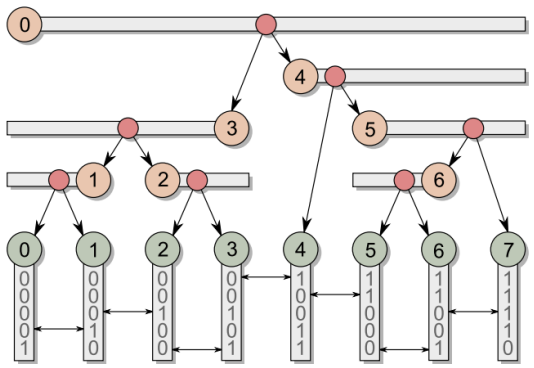

Fig. 4. Example of node layout. Each internal node covers a linear range of keys, which it partitions into two subranges according to their first differing bit.

IV. Visualization and calculations on GPU

For visualization and access to GPU, we used OpenGL – API (a specification to be more precise), which allows developers to interact with the GPU to render and perform calculations. It provides a high-performance and parallel processing environment through programmable shaders, customizable pipelines, and features for GPU computing. The OpenGL specification specifies exactly what the result/output of each function should be and how it should perform.

A. Shader graphics pipeline

In OpenGL, everything is in 3D space, but our screen is in 2D. To transform 3D coordinates into 2D colored pixels, OpenGL provides a graphics pipeline consisting of shaders – programs running on the GPU for each pipeline step.

1) Vertex shader: sets the position of each vertex, taking into account camera rotation, projection, and perspective.
2) Fragment shader, which runs for each fragment, calculates the final color of each pixel.

B. Compute shaders

To perform our algorithm efficiently, we will use GPU not only for graphics but also for computational tasks: so-called General Purpose Computing on Graphics Processing Units (GPGPU), as GPU performs floating-point calculations much faster than current CPU.

This can be done using compute shaders, which are general-purpose shaders that are not a part of the graphics pipeline. We can define the number of executions and initiate them by ourselves using OpenGL functions. If other shaders mentioned above ran per vertex or fragment, compute shaders work on the so-called "work item." The compute space of the compute shader is defined by the workgroup, which consists of some

Algorithm 1 Pseudocode for constructing a binary radix tree

for each internal node with index $i \in [0, n-2]$ in parallel **do**
  $d \leftarrow sign(\delta(i, i+1) - \delta(i, i-1))$
  $\delta_{min} \leftarrow \delta(i, i-d)$
  $l_{max} \leftarrow 2$
  while $\delta(i, i + l_{max} * d) > \delta_{min}$ **do**
    $l_{max} \leftarrow l_{max} * 2$
  end while
  $l \leftarrow 0$
  for $t \leftarrow \{l_{max}/2, l_{max}/4, ..., 1\}$ **do**
    if $\delta(i, i + (l + t)) * d > \delta_{min}$ **then**
      $l \leftarrow l + t$
    end if
  end for
  $j \leftarrow i + l * d$
  $\delta_{node} \leftarrow \delta(i, j)$
  $s \leftarrow 0$
  for $t \leftarrow \{\lceil l/2 \rceil, \lceil l/4 \rceil, .., 1\}$ **do**
    if $\delta(i, i + (s + t) * d) > \delta_{node}$ **then**
      $s \leftarrow s + t$
    end if
  end for
  $\gamma \leftarrow i + s * d + min(d, 0)$
  if $min(i, j) = \gamma$ **then**
    $left \leftarrow L_\gamma$
  else
    $left \leftarrow I_\gamma$
  end if
  if $max(i, j) = \gamma + 1$ **then**
    $right \leftarrow L_{\gamma+1}$
  else
    $right \leftarrow I_{\gamma+1}$
  end if
  $I_i \leftarrow (left, right)$
end for

number of compute shader invocations (for each, a uniquely determined set of inputs is defined).

To perform all the necessary calculations on GPU, a few different compute shaders are used:

1) Morton codes: each invocation (one for every particle) finds its corresponding Morton code and writes it into the following buffer.
2) Bitonic sort: each invocation performs some part of the sorting algorithm to sort the buffer with Morton codes.
3) Building tree: each invocation corresponds to one internal node – defines its children, calculates the total mass, the center of mass, and the bounding box metric used in the tree traversing (it was defined as the longest side of the bounding box).
4) Traversing tree: each invocation traverses the constructed tree for the corresponding leaf node (exact particle) and calculates the total force, position, and speed change.

V. Results

A. Interface

After executing the program, the user can see the menu where they can set up initial conditions for the simulation (Fig. 5).

- The speed, the mass of particles, and the building tree rate can be set up using a slider. There are buttons "Simulation" and "Rotation" to run or stop the simulation and the rotation accordingly.
- The number of particles is defined by "Start count sqrt.". It will equal "Start count sqrt" * "Start count sqrt".
- The button "Enter fly mode" allows the user to rotate the camera in different directions.
- The button "Regenerate" regenerates the simulation with new initial conditions.

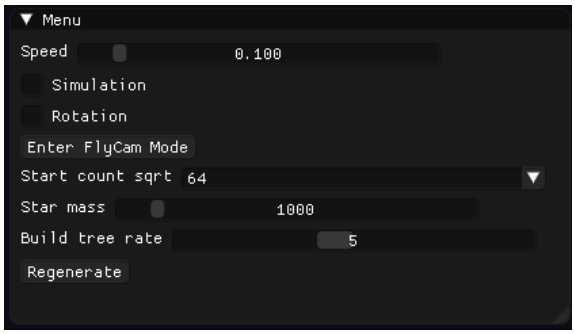

Fig. 5. Interface of the program.

For more details, see the repository of our project: https://github.com/beheni/N-BodySimulation.

B. Optimization steps

All tests were run on the computer with following parameters:

- CPU: Intel i5-10210U,
- GPU: NVIDIA GeForce MX350.

TABLE I
Comparison of different approaches

Approach	Number of particles	FPS
Brut-force on CPU	1000	30
Brut-force on GPU	16 384	50
Barnes-Hut algorithm	262 144	15
Barnes-Hut algorithm with rebuilding tree every five frames	262 144	50
Barnes-Hut algorithm with rebuilding tree every five frames	1 048 576	15

VI. Conclusion

Barnes-Hut Algorithm is a practical approach to modeling gravitational interaction on large scales. It allows for improving computational efficiency while still providing reasonably accurate results. We achieved 50 FPS for 262 144 particles, but the FPS for 1 048 576 is still low (15 FPS).

REFERENCES

[1] T. Hamada and K. Nitadori, "190 tflops astrophysical n-body simulation on a cluster of gpus," in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010, pp. 1–9.

[2] C. to Wikimedia projects, "Particle mesh," Jul 2022. [Online]. Available: https://en.wikipedia.org/wiki/Particle_mesh

[3] R. Reusser, "2d (non-physical) n-body gravity with poisson's equation," Mar 2023. [Online]. Available: https://observablehq.com/@rreusser/2d-n-body-gravity-with-poissons-equation

[4] [Online]. Available: https://www.cs.princeton.edu/courses/archive/fall04/cos126/assignments/barnes-hut.html

[5] C. to Wikimedia projects, "Z-order curve," May 2023. [Online]. Available: https://en.wikipedia.org/wiki/Z-order_curve

[6] T. Karras, "Maximizing parallelism in the construction of BVHs, octrees, and k-d trees," Jun 2012. [Online]. Available: https://developer.nvidia.com/blog/parallelforall/wpcontent/uploads/2012/11/karras2012hpg_paper.pdf

[7] "Implementing bitonic merge sort in vulkan compute." [Online]. Available: https://poniesandlight.co.uk/reflect/bitonic_merge_sort/