# High-Performance CUDA Implementation of N-Body Simulation with Barnes-Hut Algorithm

Hsin-Hung Wu

May 5, 2023

## 1 Introduction

### 1.1 Problem description

N-body simulation simulates a dynamical system of particles, usually under the influence of physical forces. In our N-body problem, we will be predicting the individual movements of a group of stars and planets interacting with each other through gravitational force. We will implement and compare both the direct-sum algorithm, which takes $O(N^2)$, and the Barnes-Hut algorithm, which takes $O(N \log N)$. The direct-sum algorithm is a brute force algorithm that explicitly simulates the interaction of an object with every other object in the system. The Barnes-Hut algorithm is an approximation algorithm that first divides the bodies into groups and stores them in a quad-tree. Nearby bodies are treated individually, but distant bodies are approximated as a single large body [1]. Barnes-Hut algorithm consists mainly of hierarchical quad-tree construction and force calculation. In tree construction, each body is inserted into the tree recursively and the center of mass for each node is updated accordingly. During force calculation on each body, the net force is summed up through the traversal of the quad-tree. After the force calculation, the algorithm calculates and updates the new positions and velocities for each body. The tree will then be discarded and a new tree will be generated for each of the following iterations.

### 1.2 Suitability for GPU acceleration

The N-body problem consists of expensive computation in force calculations among all pairs of bodies, which is highly parallelizable as each pair of bodies can be computed independently of each other and so is the net force summation on each body. Direct-sum algorithm will require $O(N^2)$ computations and can be parallelized by allocating N threads where each thread computes the force acting on each of the N bodies. Barnes-Hut algorithm will require $O(N \log N)$ computations and can be parallelized in bounding box computation, quad-tree construction, and force update phases. The bounding box can be computed with parallel reduction. The construction of the quad-tree can be done in a top-down hierarchical manner where each node in the same level can be constructed in parallel. The force of all bodies can be computed by having each body traverse the quadtree in parallel.

### 1.3 Intellectual Challenges

N-body is highly applied in astrophysics, to understand the evolution of the universe's large-scale structures such as galaxy filaments, galaxy halos, and the dynamic evolution of star clusters [2], or in protein folding, turbulent fluid flow simulation, global illumination computation, and etc [3]. For the direct-sum method, it is pretty straightforward to parallelize. We need to pick the optimal block size, to optimize shared memory access, data reuse, and locality. For Barnes-Hut algorithm, there are more challenges mainly in the quad-tree construction phase. We must minimize communication and synchronization during construction. In addition, constructions of quad-tree among blocks have nonuniform workloads since the bodies are not uniformly distributed which causes irregular data access patterns. We will need to load balance to ensure the blocks have a similar workload. Lastly, in the force update phase, we must minimize global access when each thread traverses the quad-tree.

# 2 Methods

## 2.1 Data and Parameters

For our n-body simulation, the parameters of the bodies, which include mass, radius, position, velocity, and acceleration, follow an astronomical scale. In order to display real astronomical positions on a fixed-size window, we scale the positions down. For the quadtree node, the parameters contain centerMass, totalMass, isLeaf, two vectors that identify the bounding box, and a start and end indices that represent the subset of bodies the node contains. We provide four simulations which are spiral galaxy, random initialization, galaxy collision, and our solar system.

## 2.2 Direct-Sum Algorithm

For the Direct-Sum algorithm, we first considered computing all pairs in parallel, but it would require $O(N^2)$, which would be limited by memory bandwidth. We then decided to serialize some of the computation by letting each thread compute all N interactions for one body. We used tiling to optimize data reuse, utilize shared memory, and coalesce memory accesses. It allows us to do $p^2$ computations with $2p$ body descriptions as shown in Figure 1 [3].

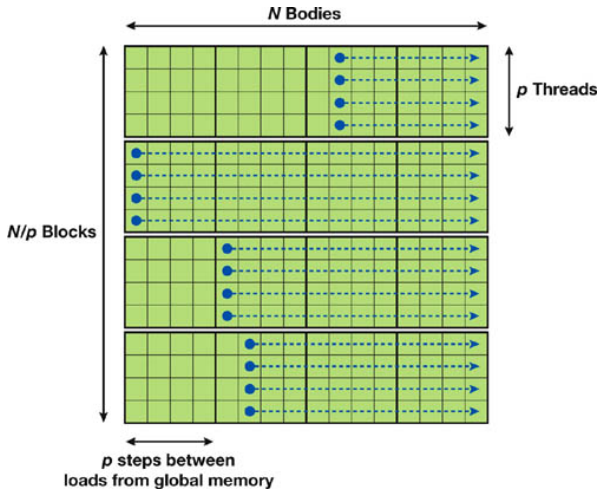



Figure 1: Tiled Direct-Sum Implementation [3]

## 2.3 Barnes-Hut Algorithm

For the Barnes-Hut algorithm, Algorithm 1 gives us a high-level overview, which contains the four kernels that are used.

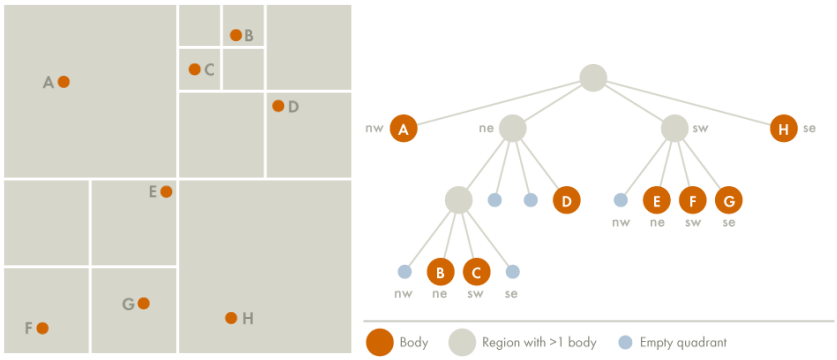
**Algorithm 1** Barnes-Hut Algorithm Overview

**for** each timestep **do**
1. **reset memory**
2. **compute bounding box**
3. **construct quadtree**
4. **compute force**

**end for**

Kernel 1 resets the memory of the quadtree to its default values. Kernel 2 computes the bounding box that contains all the N bodies and the bound will become the root node of the quadtree. Kernel 3 constructs the quadtree in a top-down hierarchical manner using dynamic parallelism where each block represents a node and holds all the bodies within its bound. The node will then spawn four more blocks, one for each of its quadrants, until it either reaches the leaf node or contains one body. Kernel 4 uses the constructed quadtree to compute the force acting on each of the N bodies. Figure

2a shows region partitions with bounding boxes and Figure 2b shows a constructed quadtree with the given bodies.




(a) region partitions (b) quad-tree

Figure 2: Quadtree and Bounding Box[4]

# 3 Implementation

## 3.1 Global Optimizations

The core data structure for the Barnes-Hut algorithm is the quad-tree as it is what the body uses to update its force. Quadtree is typically built with pointers on the heap and the child nodes are dynamically allocated as the tree grows, which introduces performance overheads. We will instead implement the quadtree using a fixed-size array and traverse the tree with indexing. We picked the array size based on our 2d window size, body radius/pixel, and the balance between efficiency and accuracy [5].

## 3.2 Kernel 1

For the first kernel, we will reset each quadtree node to its default value in parallel where each thread will be resetting one node.

## 3.3 Kernel 2

For the second kernel, we will compute the bounding box of all bodies for the root node of the quadtree as the root node encompasses the whole 2d space. We first split the N bodies into multiple blocks where each block will find the bound for a subset of the bodies. Each body is read globally once in a fully coalesced manner when loaded into shared memory. The bound will then be computed using parallel reduction. Lastly, each block updates the root node's bound with min and max and global atomic operations to prevent race conditions when accessing the root node in parallel.

## 3.4 Kernel 3 [6]

For the third kernel, we will compute the quadtree in a top-down hierarchical manner to avoid locks and synchronizations. One node is assigned to one thread block. Initially, at depth 0, one thread block is assigned to the entire two-dimensional space, which contains all bodies. It divides the space into four quadrants and launches one thread block for each quadrant. These child blocks will again subdivide their quadrants if they contain more than one body or have not reached the leaf nodes.

First, the block will compute its center of mass with the bodies it contains. We compute the center of mass for each node while we construct the quadtree to avoid an addition kernel just for updating the center of mass. When computing the center of mass, we use parallel reduction and warp unrolling for speed up. If the block contains more than one body and is not at the leaf nodes, then we will proceed with the process of launching child blocks. The number of bodies for each quadrant

is counted in parallel with atomicAdd to update the count for each quadrant. A four-element parallel scan operation is used to compute the offsets to the locations where the bodies will be stored. Then, the bodies are reordered in parallel, so that those bodies in the same quadrant are grouped together and placed into their section of the body storage. Finally, we assign the new start and end indices for each child node as each node uses two indices to keep track of the bodies it contains. The block then launches a child kernel with four thread blocks, one for each of the four new quadrants.

Figure 3 shows us an overview of kernel 3 including how it launches new blocks with dynamic parallelism.

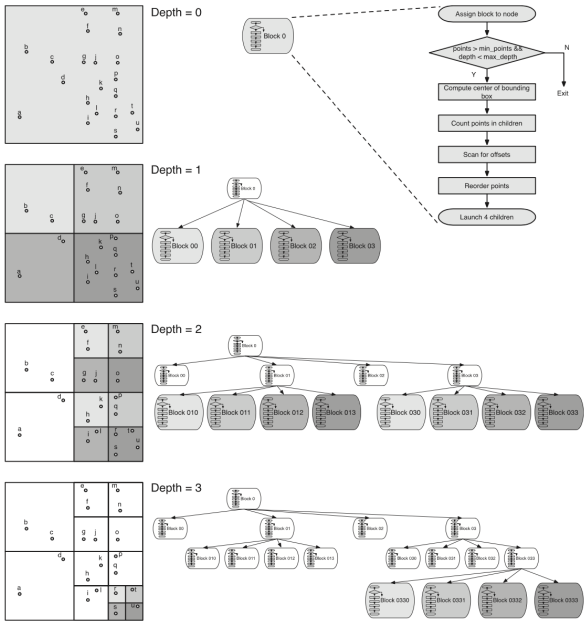



Figure 3: Dynamic Parallelism [6]

Figure 4 shows us how the bodies are reordered and regrouped using two buffers. At depth 0, the bodies are stored in buffer 0 and buffer 1 will be the reorder buffer. Each buffer will be used alternatively as the reorder buffer as the tree traverses.

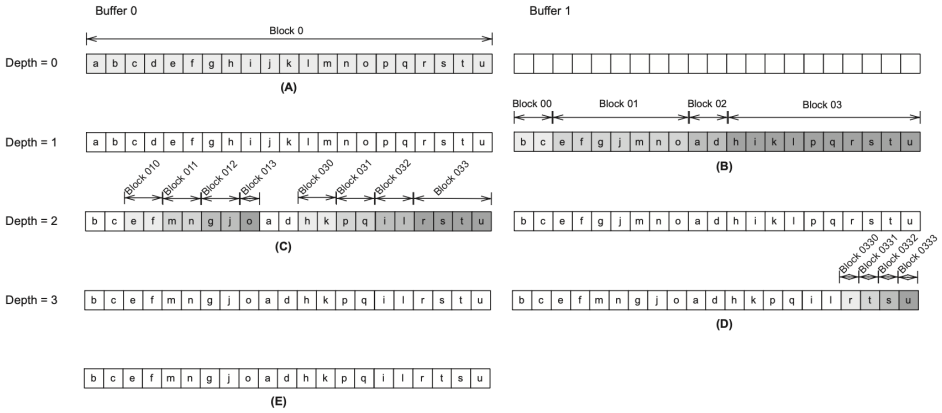



Figure 4: Body Reorder [6]

## 3.5 Kernel 4

For the fourth kernel, we will compute the force acting on each body with the previously constructed quadtree. We originally considered using a queue to traverse the quadtree iteratively in a breath-first-search manner to avoid the overhead that comes with recursions. However, the queue that needs to be

stored on the stack has to be as large as the number of leaves in the quadtree, which is not feasible. We then decided to traverse the quadtree recursively in a depth-first search manner where the recursion depth will be only as much as the max depth of the quadtree.

# 4 Evaluations

Figure 5 shows the run time for all implementations as the number of bodies increases. As we can see, the GPU implementations are more efficient than the CPU by an order of magnitude and the Barnes-Hut algorithm is the most efficient algorithm of the two. Direct Sum GPU runs faster than Barnes-Hut GPU when $N < 10000$, but its run time starts growing exponentially as the number of bodies increases beyond that. Barnes-Hut GPU scales really well as the run time grows linearly with 1509.54 milliseconds of run time per iteration for 5,000,000 bodies.

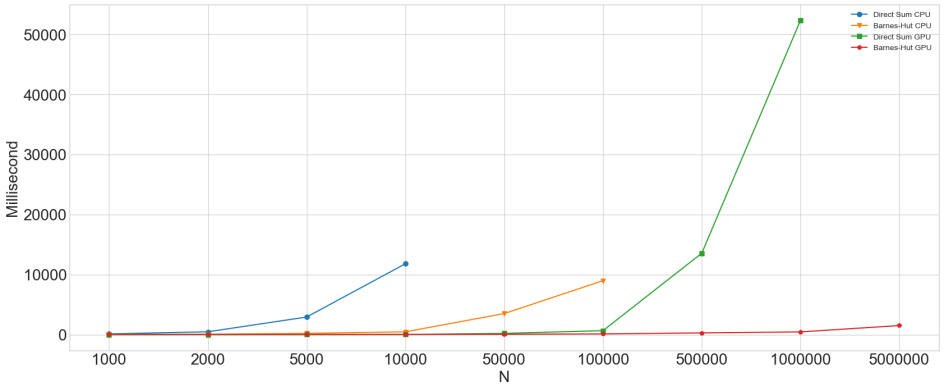



Figure 5: Runtime per iteration in millisecond

Figure 6 shows how the run time of the compute force kernel changes as the block size changes and as we can see, the run time increases as the block size increases. In the compute force kernel, we traverse the quadtree recursively which introduces additional overhead and potentially impacts the performance as the block size increases. Increasing the block size means that each block requires more of these resources and so if the resources are insufficient to accommodate larger block sizes, the GPU may need to schedule and execute blocks in a less efficient manner, leading to increased runtime. We picked 32 as the block size for the compute force kernel as it gives us the most efficient result.

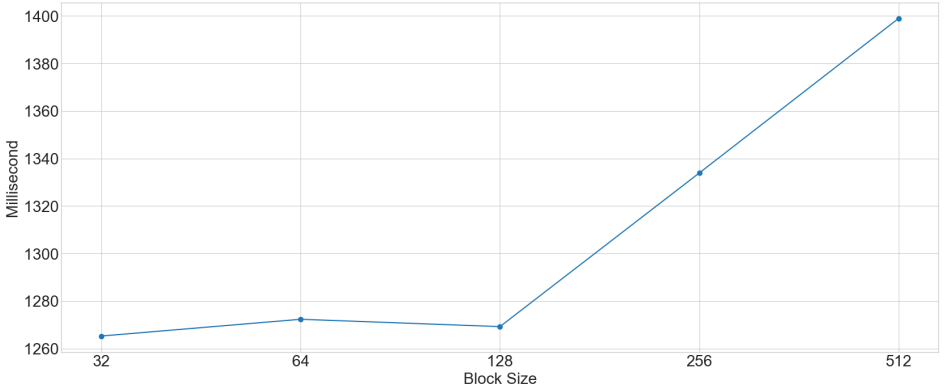



Figure 6: Runtime for different block sizes in milliseconds

| Barnes-Hut GPU Kernel Runtime for one iteration with 5,000,000 bodies | | | | |
|---|---|---|---|---|
| | **Kernel 1** | **Kernel 2** | **Kernel 3** | **Kernel 4** |
| CPU Runtime (ms) | 0 | 321 | 17025 | 776512 |
| GPU Runtime (ms) | 0.159 | 3.16 | 240.9 | 1265.32 |
| CPU/GPU | 0 | 101.58 | 70.67 | 613.68 |

From the table above, we can see the run time for each kernel and the CPU/GPU ratio which measures the speed up. As we can see, kernel 4, the compute force kernel, is the slowest and takes around 84% of the total run time. However, the most speed-up also comes from kernel 4 which is more than 600 times faster than the CPU version.

# 5 Conclusions

The CUDA implementation makes n-body simulation feasible as we are able to simulate 5 million or more bodies in real time. A lot of optimization and redesigning is done when switching from the CPU version to the GPU version. One major redesign is to use a fixed-size array instead of pointers for the quadtree. It reduces the overhead of pointers and heap and eliminates the need for locks and synchronization, which gives us drastic speed-ups. Different techniques such as the use of shared memory, memory coalescing, warp unrolling, tiling, parallel reduction, dynamic parallelism, computation-to-communication ratio and etc, are used to reduce global memory accesses and for achieving high performance. This simulation allows us to visualize and study the formation of our galaxy and gain deeper insight into our universe.

# 6 Future Directions

One major future improvement would be to implement the simulation in 3D instead of 2D as 2D only gives us an idea, but not the real picture of our universe. A 3D implementation of the Barnes-Hut algorithm would then require the use of an octtree instead of a quadtree. Overall, there are improvements to be made regarding kernel implementation, hyper-parameter tuning, and algorithm design such as using the Fast multipole method.

# 7 Source Code

The source code is here.

# References


[1] Wikipedia. Barnes–Hut simulation — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Barnes%E2%80%93Hut%20simulation&oldid=1092207998, 2023. [Online; accessed 26-February-2023].

[2] Wikipedia. N-body simulation — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=N-body%20simulation&oldid=1115952336, 2023. [Online; accessed 26-February-2023].

[3] Lars Nyland, Mark Harris, and Jan Prins. Chapter 31. fast n-body simulation with cuda. https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-31-fast-n-body-simulation-cuda.

[4] Tom Ventimiglia and Kevin Wayne. The barnes-hut algorithm. http://arborjs.org/docs/barnes-hut.

[5] Martin Burtscher and Keshav Pingali. An efficient cuda implementation of the tree-based barnes hut n-body algorithm. *GPU Computing Gems Emerald Edition*, 12 2011.

[6] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Applications of GPU Computing Series. Morgan Kaufmann Publishers, Burlington, MA, 2010.