N-Body Simulation: CPU Barnes-Hut Method vs GPU Tiling Method

Virinchi Lalwani

April 2025

1 Executive Summary

This report documents the implementation of a 2D N-body simulation that explores various algorithms and hardware acceleration techniques to efficiently simulate gravitational interactions between multiple bodies. The project successfully demonstrates the performance differences between CPU-based Barnes-Hut algorithm and GPU-accelerated Direct Sum method using CUDA and shared memory tiling optimization. Results show significant performance gains with the GPU implementation, particularly as the number of bodies increases.

2 Project Description

2.1 Overview

The N-body simulation models gravitational interactions between multiple particles in a 2D space. Each body exerts gravitational force on all other bodies according to Newton's law of universal gravitation. As the number of bodies increases, the computational complexity grows substantially, making efficient algorithms and hardware acceleration crucial for interactive simulations.

2.2 Objectives

- Implement a Barnes-Hut algorithm for CPU-based N-body simulation
- Implement a Direct Sum algorithm with shared memory tiling for GPU acceleration
- Create an interactive visualization using Pygame
- Compare performance between algorithms as the number of bodies scales
- Provide a user-friendly interface for experimentation

2.3 Technologies Used

- Python for high-level program structure
- Pygame for visualization and user interaction
- Numba and CUDA for GPU acceleration
- NumPy for efficient array operations

3 Research and Design

3.1 Algorithms

3.1.1 Direct Sum (Brute Froce)

The direct sum approach calculates forces between every pair of bodies, resulting in O(N^2) computational complexity. While computationally expensive, this algorithm is straightforward to implement and can be efficiently parallelized on GPUs.

```
For each body i:
    For each body j (j not equal i):
        Calculate force between i and j
        Add force to body i's total force
```

3.1.2 Barnes-Hut Algorithm

The Barnes-Hut algorithm reduces computational complexity to O(N log N) by approximating forces from clusters of distant bodies. It uses a quadtree data structure to recursively subdivide space. Here's how the Barnes-Hut algorithm works:

```
For each body:
    Build_Octtree()
    Calculate_Total_CenterofMass()
    Compute_Force()
    Update velocity and position()
```

The theta parameter controls the approximation accuracy - smaller values increase accuracy but reduce performance.

3.2 GPU Acceleration with CUDA

The project leverages NVIDIA's CUDA platform via Numba to accelerate computation. Since each body's force calculation is independent, the problem is well-suited for parallelization. The advantages of fast shared memory can be utilized to accelarate our computations.

3.3 Shared Memory Tiling

The tiling method is an optimization technique used in the direct sum (all-pairs) N-body simulation to improve memory efficiency and overall performance on GPUs.

- **Divide Work into Tiles:** The entire set of particles is divided into smaller "tiles" or blocks. Each tile consists of a subset of particles that can be loaded into the GPU's shared memory. Here's how the tiling method for GPU accelaration works:
- **Load Data into Shared Memory:** Instead of having every thread access global memory repeatedly (which is slow), a block of threads first loads a tile (e.g., positions and masses of a group of particles) from global memory into shared memory. Since shared memory is much faster, this minimizes redundant global memory accesses.
- **Compute Interactions Within the Tile:** Once the data is in shared memory, each thread in the block computes the interactions (gravitational forces) between its assigned particle and all particles in the current tile. After computing these interactions, the results (such as partial force contributions) are accumulated.
- **Iterate Over All Tiles:** The process is repeated: the kernel loads the next tile into shared memory and each thread computes interactions with that tile's data. This continues until all particles have been processed.
- **Combine Partial Results:** After processing all tiles, each thread ends up with the full accumulated force for its corresponding particle, which can then be used to update velocities and positions.

This approach significantly reduces global memory accesses, which are a major performance bottleneck in GPU computing.

4 Implementation

4.1 Code Structure

The simulation is built using the following key components:

1. Main Simulation: Controls the simulation loop, rendering, and user interaction
2. Body Class: Represents individual particles with physical properties
3. Rectangle Class: Defines spatial boundaries for the quadtree
4. QuadTree Class: Handles the Quadtree creation
5. CUDA Kernel: Performs GPU-accelerated direct-sum calculations with optimization

4.2 Structure Details

4.2.1 Body Class

The Body class represents individual particles in the simulation with the following properties:

```python
class Body:
def __init__(self, position, mass, velocity):
    self.x = float(position[0])
    self.y = float(position[1])
    self.position = [self.x, self.y]
    self.vx = float(velocity[0])
    self.vy = float(velocity[1])
    self.ax = 0.0
    self.ay = 0.0
    self.mass = float(mass)
    self.color = "blue" if mass < 100 else "yellow" if mass < 140
        else "red"
    self.radius = 1.5
```

Listing 1: Body class definition

Key methods include:

- update_velocity(): Updates velocity based on acceleration and time step
- update_position(): Updates position based on velocity and time step
- calculate_force(): Calculates gravitational force using the Barnes-Hut approximation

4.2.2 Rectangle Class

The Rectangle class defines rectangular boundaries used in the quadtree:

```python
class Rectangle:
def __init__(self, x, y, w, h):
    self.x = x
    self.y = y
    self.w = w
    self.h = h
```

Listing 2: Body class definition

It contains one method:

- • contains(self, body): Checks if the body in withing this rectangle

4.2.3 QuadTree Class

The QuadTree class handels the methods for the Barnes-Hut implementation and maintains the quadTree structure for making force calculations.

```python
    class QuadTree:
    def __init__(self, boundary, screen):
        self.boundary = boundary
        self.capacity = 1
        self.bodies = []
        self.divided = False
        self.screen = screen
        self.total_mass = 0
        self.center_of_mass = [0, 0]
        self.northwest = None
        self.northeast = None
        self.southwest = None
        self.southeast = None
```

Listing 3: Body class definition

Key methods include:

- subdivide(): Divides the current quadrant into four subquadrants
- insert(body): Inserts a body into the appropriate quadrant
- draw(screen): Visualizes the quadtree structure on screen

4.2.4 Main Simulation Loop

The main simulation loop handles:

- Initialization of bodies and simulation parameters
- User interaction (adding particles, toggling between CPU/GPU)
- Switching between calculation methods
- Rendering and performance tracking

4.2.5 CUDA Kernel

The tiling method is implemented in the direct_sum_kernel function:

```python
    @cuda.jit
def direct_sum_kernel(positions, masses, accelerations, softening, G,
    num_bodies):
    # Shared memory for the tile of positions and masses
    shared_positions = cuda.shared.array(shape=(TILE_SIZE, 2),
        dtype=nb.float32)
    shared_masses = cuda.shared.array(shape=TILE_SIZE, dtype=nb.float32)

    # Get thread and block indices
    tx = cuda.threadIdx.x
    bx = cuda.blockIdx.x

    # Global thread index
    i = bx * BLOCK_SIZE + tx

    # Skip if this thread doesn't correspond to a body
    if i >= num_bodies:
        return

    # Local copies of body i's position and accumulator for acceleration
    pos_i_x = positions[i][0]
    pos_i_y = positions[i][1]
    acc_x = 0.0
    acc_y = 0.0

    # Loop over tiles
    for tile_start in range(0, num_bodies, TILE_SIZE):
        tile_end = min(tile_start + TILE_SIZE, num_bodies)

        # Load tile data into shared memory
        if tile_start + tx < num_bodies:
            shared_positions[tx][0] = positions[tile_start + tx][0]
            shared_positions[tx][1] = positions[tile_start + tx][1]
            shared_masses[tx] = masses[tile_start + tx]

        # Synchronize to make sure all threads have loaded the data
        cuda.syncthreads()

        # Compute interactions with bodies in this tile
        for j in range(tile_end - tile_start):
            # Skip self-interaction
            if tile_start + j != i:
                # Calculate distance
                dx = shared_positions[j][0] - pos_i_x
                dy = shared_positions[j][1] - pos_i_y
                distance_squared = dx*dx + dy*dy + softening*softening
                distance = math.sqrt(distance_squared)

                # Calculate gravitational force
                if distance > 0:
                    f = G * shared_masses[j] / distance_squared
                    acc_x += f * dx / distance
                    acc_y += f * dy / distance

        # Synchronize before loading the next tile
        cuda.syncthreads()

    # Store the final acceleration
    accelerations[i][0] = acc_x
    accelerations[i][1] = acc_y
```

Listing 4: Body class definition

Key implementation details include:

- BLOCK_SIZE: defines the number of threads per block
- TILE_SIZE: controls the size of each shared memory tile
- cuda.syncthreads() ensure all threads complete current operations before proceeding

4.3 Visualization

The visualization is implemented using Pygame, providing the following:

- Interactive addition of particles through mouse clicks
- Runtime switching between GPU and CPU algorithms
- Optional quadtree visualization
- Real-time performance metrics display

5 Resluts and Analysis

5.1 Performance Benchmarks

Performance was measured in frames per second (FPS), increasing the number of bodies:

Table 1: Performance comparison between CPU and GPU implementations

Number of Bodies	**CPU (FPS)**	**GPU (FPS)**	**Speedup Factor**
100	125	190	1.52x
1,000	12	95	7.92x
2,000	5	75	15.00x
5,000	1.7	38	22.35x
10,000	0.9	13	14.44x

5.2 Performance Comparison

The performance data demonstrate several key insights:

1. **CPU vs. GPU Scaling**: As the number of simulated bodies increases, the GPU implementation maintains significantly better performance. At 10,000 bodies, the GPU implementation gives more than 14x speedum over the CPU implementation.

2. **Algorithm Efficiency**: Although the Barnes-Hut algorithm has a better theoretical complexity (O(N log N) compared to O(N^2)), the massive parallelism of the GPU overcomes this advantage as the number of simulated bodies scales.

3. **Optimal Range**: The GPU's advantage peaks around 5,000 bodies with a 22.35x speedup. For very small simulations, the overhead of GPU data transfers reduces relative performance.

5.3 Memory Considerations

The GPU implementation's memory usage scales linearly with the number of bodies. Each body requires storage for:

- Position (2 floats)
- Mass (1 float)
- Acceleration (2 floats)

This efficient memory footprint allows for the simulation of tens of thousands of bodies within typical GPU memory constraints.

5.4 Achievement of Goals

The project successfully achieved its stated objectives:

- Both Barnes-Hut (CPU) and Direct Sum with tiling (GPU) algorithms were successfully implemented
- Interactive visualization provides smooth user experience for experimentation
- Performance comparisons demonstrate the effectiveness of GPU acceleration and shared memory tiling
- The simulation maintains interactive framerates even with large body counts when using GPU acceleration
- User interface provides easy switching between algorithms and visualization options

6 User Guide

6.1 Installation and Setup

6.1.1 Requirements

- Python 3.7+

- Pygame
- NumPy
- Numba with CUDA support
- NVIDIA GPU with CUDA capability

6.1.2 Installlation

pip install pygame numpy numba

6.1.3 Running the Simulation

python nbody_simulation.py

6.2 Controls

- **Left Click**: Add a new particle at the cursor position
- **G Key**: Toggle between GPU (Direct Sum) and CPU (Barnes-Hut) algorithms
- **Q Key**: Toggle quadtree visualization (when using the CPU algorithm).

6.3 Interface Elements

- **Algorithm Indicator**: Shows current algorithm (green for GPU, orange for CPU)
- **Bodies Counter**: Displays number of particles in simulation
- **FPS Display**: Shows current performance in frames per second
- **Control Instructions**: Listed in the top-left corner

7 References

References

[1] J. Barnes and P. Hut, "A hierarchical O(N log N) force-calculation algorithm," *Nature*, vol. 324, no. 6096, pp. 446–449, Dec. 1986.

[2] NVIDIA Corporation, "CUDA C Programming Guide," 2021. Available: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`

[3] L. Nyland, M. Harris, and J. Prins, "Fast N-Body Simulation with CUDA," in *GPU Gems 3*, H. Nguyen, Ed. Addison-Wesley, 2007, ch. 31.

[4] Hsin-Hung Chen, ”N-body simulation,” 2016. [Online]. Available: `https://hsin-hung.github.io/N-body-simulation/report.pdf`

[5] Pygame Documentation. [Online]. Available: `https://www.pygame.org/docs/`

[6] Numba Documentation. [Online]. Available: `https://numba.pydata.org/numba-doc/latest/`