Proposal: High-Performance N-Body Simulation Using CUDA

Mohammad Hassan Liaghat
Parmida Hooshang
Mohammad Reza Hadian Shirazi

December 2025

1 Introduction

The N-body problem represents a fundamental challenge in computational physics: simulating the interactions of N bodies where each body exerts forces on all others. This creates an inherently parallel problem with $O(N^2)$ computational complexity, making it an ideal candidate for GPU acceleration. We propose to implement a gravitational N-body simulation using CUDA, with primary emphasis on parallel algorithm design, memory hierarchy optimization, and performance scalability.

2 Physical Model and Interaction Dynamics

2.1 Gravitational Force Model

We will model gravitational interactions between bodies using Newton's law of universal gravitation. The force exerted on body i by body j is given by:

$$\mathbf{F}_{ij} = G\frac{m_i m_j}{(\|\mathbf{r}_{ij}\|^2 + \epsilon^2)^{3/2}}\mathbf{r}_{ij}$$

where $\mathbf{r}_{ij} = \mathbf{r}_j - \mathbf{r}_i$ is the displacement vector, G is the gravitational constant, m_i and m_j are the masses, and ϵ is a softening parameter that prevents numerical singularities when bodies approach each other closely. The softening parameter will be set to approximately $\epsilon \approx 10^{-3}$ AU (astronomical units) to maintain numerical stability.

The total acceleration on body i is computed by summing contributions from all other bodies:

$$\mathbf{a}_i = \sum_{j \neq i}^{N} \frac{\mathbf{F}_{ij}}{m_i}$$

This pairwise summation represents the computational load of our simulation and will be the primary target for GPU optimization.

2.2 Temporal Integration

We will employ the fourth-order Runge-Kutta (RK4) method for temporal integration. RK4 provides a balance between accuracy and computational cost, achieving fourth-order convergence with four force evaluations per time step.

For a system of ordinary differential equations $\frac{d\mathbf{y}}{dt} = f(\mathbf{y}, t)$, where $\mathbf{y}$ represents the state vector (positions and velocities), RK4 computes the next state as:

$$\begin{aligned}
\mathbf{k}_1 &= f(\mathbf{y}_n, t_n) \\
\mathbf{k}_2 &= f(\mathbf{y}_n + \frac{\Delta t}{2}\mathbf{k}_1, t_n + \frac{\Delta t}{2}) \\
\mathbf{k}_3 &= f(\mathbf{y}_n + \frac{\Delta t}{2}\mathbf{k}_2, t_n + \frac{\Delta t}{2}) \\
\mathbf{k}_4 &= f(\mathbf{y}_n + \Delta t\mathbf{k}_3, t_n + \Delta t) \\
\mathbf{y}_{n+1} &= \mathbf{y}_n + \frac{\Delta t}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4)
\end{aligned}$$

This method requires computing forces (and thus pairwise interactions) four times per time step, but maintains bounded energy drift over long simulations. The time step Δt will be chosen based on the shortest orbital period in the system, typically $\Delta t \approx 0.01$ orbital periods of the innermost body.

3 Simulation Scale and System Configuration

3.1 Target System Size

We plan to evaluate our implementation across multiple scales:

- **Small scale:** $N = 1,000$ bodies (development and validation)
- **Medium scale:** $N = 10,000$ bodies (primary performance analysis)
- **Large scale:** $N = 50,000$ bodies (scalability demonstration)

These scales are chosen to fit within the 6GB VRAM constraint of our target GPU (NVIDIA RTX 3050) while allowing meaningful performance analysis. For $N = 50,000$ bodies with double-buffered position/velocity data (24 bytes per body for position and velocity in single precision), we require approximately 2.4 GB for state storage, leaving sufficient memory for intermediate computations and buffers.

3.2 System Composition

Our solar system model will include:

- One central massive body (Sun analog, $M_\odot \approx 2 \times 10^{30}$ kg)
- Multiple planetary bodies with masses ranging from 10^{-6} to 10^{-3} solar masses
- A population of smaller bodies (asteroids, comets) with negligible individual gravitational influence

This hierarchical mass distribution reflects realistic astrophysical systems where a central body dominates the gravitational potential. The presence of many small bodies increases N for performance testing while maintaining physical relevance.

3.3 Validation Strategy

To verify physical correctness, we will:

1. Compare planetary orbits against known solutions for simplified N-body problems (e.g., two-body orbits, restricted three-body problem)
2. Use NASA JPL Horizons API to obtain ephemeris data for actual solar system bodies and compare simulated trajectories over short time spans
3. Monitor conservation of total energy and angular momentum as indicators of numerical accuracy
4. Verify that stable orbital configurations remain stable over extended simulations (thousands of time steps)

The NASA JPL Horizons system provides precise position and velocity data for solar system objects, allowing quantitative validation of our gravitational model when applied to realistic initial conditions.

4 CUDA Parallelization Strategy

4.1 Computational Structure

The N-body force calculation exhibits natural data parallelism: each body's acceleration can be computed independently, but requires reading the positions of all other bodies. This creates a read-heavy computational pattern with high arithmetic intensity, well-suited for GPU acceleration.

Our parallelization approach maps one thread per body for force computation. Each thread:

1. Reads its own position and mass from global memory
2. Iterates over all other bodies, computing pairwise force contributions
3. Accumulates the total acceleration in thread-local registers
4. Writes the result back to global memory

4.2 Memory Hierarchy Exploitation

The RTX 3050's memory hierarchy will be exploited as follows:

Global Memory: Stores primary data arrays (positions, velocities, masses, accelerations). We will use structure-of-arrays (SoA) layout to ensure coalesced memory access. Position and velocity data will be stored as `float4` types, allowing vectorized loads and efficient memory transactions.

Shared Memory: Each thread block will use shared memory as a software-managed cache for position data. By loading tiles of N_{tile} bodies into shared memory (typically 128-256 bodies per tile), we reduce global memory accesses by a factor of N_{tile}. With 48 KB of shared memory per SM and 256-thread blocks, we can cache approximately 256 body positions per tile.

Constant Memory: Physical constants (G, ϵ) and simulation parameters will be stored in constant memory, which provides broadcast capabilities and reduces register pressure.

Registers: Force accumulation will occur entirely in registers, avoiding unnecessary memory traffic for intermediate values.

4.3 Kernel Architecture

We will implement three primary computational kernels:

4.3.1 Tiled Force Computation Kernel

This kernel computes accelerations for all bodies using a tiled shared memory approach. The algorithm divides the N bodies into tiles, loads each tile into shared memory, and computes forces between the current body and all bodies in the tile. This pattern is repeated for all tiles, with synchronization barriers ensuring correct tile loading.

The tiling strategy reduces global memory reads from $O(N^2)$ to $O(N^2/N_{\text{tile}})$, providing substantial bandwidth savings. Block sizes will be tuned to maximize occupancy while fitting shared memory requirements.

4.3.2 Integration Kernel

A separate kernel updates positions and velocities using the computed accelerations. For RK4 integration, this kernel will be invoked four times per time step with different intermediate states. Position updates follow:

$$\mathbf{r}_{n+1} = \mathbf{r}_n + \Delta t\, \mathbf{v}_n + \frac{(\Delta t)^2}{2}\mathbf{a}_n$$

with appropriate modifications for each RK4 stage. This kernel is memory-bound and benefits from coalesced access patterns.

4.3.3 Reduction Kernel

For validation and monitoring, we will implement reduction kernels to compute total energy and angular momentum. These kernels use tree-based reduction in shared memory to efficiently aggregate contributions from all bodies.

4.4 Optimization Techniques

- **Memory Coalescing:** Structure-of-arrays data layout ensures that consecutive threads access consecutive memory addresses, maximizing memory bandwidth utilization.
- **Double Buffering:** Separate input and output buffers prevent read-after-write hazards and enable pipelining of computation and memory operations.
- **Loop Unrolling:** Inner loops in force computation will be partially unrolled to increase instruction-level parallelism and reduce branch overhead.
- **Fused Multiply-Add:** Gravitational force calculations will use `fma()` intrinsics to leverage hardware FMA units, reducing instruction count and improving precision.

5 Expected Performance Characteristics

5.1 Computational Complexity

The direct N-body algorithm has $O(N^2)$ complexity per time step. For $N = 10,000$ bodies, each time step requires $\sim 10^8$ pairwise force calculations. With RK4 integration requiring four force evaluations per step, we expect $\sim 4 \times 10^8$ pairwise interactions per time step.

At peak performance, the RTX 3050's 2560 CUDA cores operating at $\sim$ 1.5 GHz provide theoretical throughput of ~ 4 TFLOPS (single precision). Gravitational force calculation requires approximately 20 floating-point operations per pair, suggesting theoretical performance of $\sim 2\times10^{11}$ interactions per second, or ~ 0.5 ms per time step for $N = 10,000$.

In practice, memory bandwidth limitations will likely dominate. The RTX 3050's memory bandwidth ($\sim$ 192 GB/s) limits the rate at which position data can be read. Without tiling, each time step reads $N^2 \times 16$ bytes (assuming `float4` positions), which for $N = 10,000$ would require 1.6 TB of data transfer—far exceeding available bandwidth.

5.2 Anticipated Bottlenecks

- **Memory Bandwidth:** For large N, global memory bandwidth becomes the limiting factor. Tiling strategies reduce this bottleneck but cannot eliminate it entirely.
- **Shared Memory Capacity:** Larger tiles improve bandwidth efficiency but are constrained by the 48 KB shared memory limit per SM.
- **Register Pressure:** Complex force calculations and RK4 intermediate states may increase register usage, potentially reducing thread occupancy.
- **Synchronization Overhead:** Frequent `__syncthreads()` calls in the tiled kernel introduce synchronization costs that become more significant at small tile sizes.

5.3 Mitigation Strategies

We will address these bottlenecks through:

- Careful tuning of tile sizes to balance shared memory usage and bandwidth reduction
- Profiling-guided optimization using NVIDIA Nsight Compute to identify memory access patterns and occupancy issues
- Kernel fusion to reduce intermediate memory traffic
- Exploration of mixed-precision arithmetic (FP32 for computation, FP16 for storage) if precision requirements permit

6 Performance Evaluation Methodology

6.1 Metrics

We will measure and report:

- **Elapsed time per simulation step** as a function of N
- **Interaction throughput:** pairwise interactions computed per second
- **Memory bandwidth utilization:** achieved bandwidth relative to hardware peak
- **Computational throughput:** floating-point operations per second
- **Scaling behavior:** empirical scaling exponent relative to theoretical $O(N^2)$

6.2 Profiling and Analysis

NVIDIA Nsight Compute will be used to analyze:

- Memory throughput and cache hit rates
- Warp occupancy and stall reasons
- Instruction mix and execution efficiency
- Shared memory bank conflicts

These metrics will guide iterative optimization and justify design decisions in the final report.

6.3 Physical Validation

We will verify correctness through:

- Energy conservation checks (relative energy drift $< 10^{-4}$ over 1000 time steps)
- Angular momentum conservation for isolated systems
- Comparison of planetary trajectories with JPL Horizons ephemeris data
- Visual inspection of orbital stability and comet deflection events

7 Implementation Platform

7.1 Hardware

- **GPU:** NVIDIA GeForce RTX 3050 Laptop (6GB VRAM, 2560 CUDA cores, Ampere architecture)
- **CPU:** Intel Core i5-13420H (8 cores, 12 threads)
- **RAM:** 16GB DDR4 3200 MT/s (dual channel: 2×8GB)

7.2 Software

- **CUDA Toolkit:** Version 12.x
- **Compiler:** NVCC with optimization flags
- **Development Environment:** Visual Studio 2022 / Linux with GCC
- **Profiling Tools:** NVIDIA Nsight Compute, Nsight Systems
- **Visualization:** OpenGL for rendering (post-processing of simulation results)

8 Team Organization

The complexity and scope of this project justify a three-member team with complementary expertise:

- **CUDA Optimization:** Responsible for kernel implementation and memory hierarchy optimization.
- **Numerical Validation and Profiling:** Implements integration schemes, validates physical correctness, and interfaces with JPL Horizons API + performance profiling
- **Visualization and Analysis:** Develops visualization pipeline and conducts performance analysis

Each member contributes specialized knowledge while collaborating on overall system design and debugging.