Project Proposal: GPU-Accelerated N-Body Simulation

Topic: High-Performance Solar System Dynamics using CUDA

Target Hardware: NVIDIA RTX 4070 Mobile

1. Project Overview

This project aims to implement a high-performance N-Body simulation "from scratch" using CUDA C++. The simulation will model a gravitational system resembling a simplified solar system (a dominant central mass, orbiting planets, and a chaotic comet). The primary objective is not just physical simulation, but the analysis of GPU architectural efficiency. I will implement, optimize, and benchmark a brute-force $O(N^2)$ interaction model, moving from a naïve global memory approach to an optimized shared-memory tiled approach.

2. Simulation Scale and Physics Model

The Scale (N)

To effectively saturate the $compute$ capabilities of the RTX 4070, I will $test$ the system across a logarithmic scale of bodies:

- **Validation Scale:** $N = 1{,}024$ (To verify orbital stability and correctness).
- **Performance Scale:** $N = 4{,}096$ to $65{,}536$ bodies.
    - *Rationale:* A brute-force calculation for $N = 65{,}536$ requires over 4 billion interaction checks per time step. This is sufficient to expose the difference between memory-bound and compute-bound execution paths on the GPU.

Interaction Model

The simulation will rely on **Newton's Law of Universal Gravitation**. For every pair of bodies i and j, the force exerted on body i by body j is:

$$\overrightarrow{F_{ij}} = \frac{G \cdot m_i \cdot m_j}{|\overrightarrow{r_{ij}}|^3} \cdot \overrightarrow{r_{ij}}$$

where $\overrightarrow{r_{ij}} = \overrightarrow{p_j} - \overrightarrow{p_i}$

To prevent numerical instability (explosive forces) when bodies come extremely close (e.g., a comet passing near a planet), I will apply a softening factor (ϵ) to the distance calculation:

$$|\overrightarrow{r_{ij}}| \approx \sqrt{dist^2 + \epsilon^2}$$

Time Integration

I will use the **Semi-Implicit Euler** or **Velocity Verlet** method for time integration. These are symplectic integrators, meaning they conserve energy better than standard Euler integration, which is critical for maintaining stable planetary orbits over long simulation durations.

3. CUDA Parallelization Strategy

Mapping to Hardware

The problem is inherently data-parallel. I will map the workload such that **one CUDA thread computes the total force for one body**.

- **Grid Configuration:** A 1D grid where the total number of threads $\geq N$.
- **Thread Responsibility:** Thread i loads the position/mass of Body i, loops through all other Bodies j $(0\ to\ N-1)$, accumulates the force vectors, and updates Body i's velocity and position.

Memory Optimization Strategy (The "Tiled" Approach)

A naïve implementation reads body data from slow Global Memory for every interaction, resulting in redundant memory traffic. I plan to implement the **Tiled N-Body algorithm**:

1. **Shared Memory Tiling:** Threads in a block will cooperatively load a "tile" (subset) of bodies into fast on-chip Shared Memory.
2. **Synchronization:** Using $\_\_syncthreads()$, the block will wait until the tile is loaded.
3. **Compute:** Threads will compute interactions against the cached tile in Shared Memory, rather than Global Memory.
4. **Repeat:** The block moves to the next tile until all N bodies have been processed.

This strategy is expected to drastically reduce global memory bandwidth pressure and improve arithmetic intensity.

Data Layout

I will experiment with **Structure of Arrays (SoA)** (storing all X positions together, all Y positions together) versus **Array of Structures (AoS)** ($struct\ Body\ \{\ float\ x, y, z;\ \}$). The

RTX 4070 typically favors coalesced memory accesses, which SoA often provides, though AoS is more intuitive for physics.

4. Expected Performance Bottlenecks

1. Memory Bandwidth (Global Memory):

In the naïve implementation, the ratio of memory operations to math operations is high. The GPU may spend more time fetching position data than calculating gravity. This will likely be the primary bottleneck before optimization.

2. Register Pressure:

If the kernel uses too many registers per thread (to store force accumulators, velocity, position, and mass), the GPU occupancy (active warps per SM) may drop, reducing the ability to hide memory latency.

3. Instruction Latency (Special Function Units):

The gravitational formula involves $rsqrt()$ (reciprocal square root). While CUDA has a fast intrinsic hardware instruction for this ($\_rsqrtf$), heavy use of square roots can bottleneck the math pipeline if not managed correctly.

5. Evaluation Plan

Correctness Verification

- **Visual Check:** Output position data to . csv files and use a Python script (Matplotlib) to render the orbits. I expect to see the "Comet" follow a highly elliptical path, potentially being deflected if it passes near a massive "Jupiter-like" body.
- **Energy Conservation:** I will log the total system energy (Kinetic + Potential) over time. While slight drift is expected, sudden spikes will indicate numerical errors or bugs.

Performance & Scalability Analysis

I will measure and report:

1. **Interactions per Second:** The primary metric for N-Body performance.
2. **Execution Time per Frame:** Measured using cudaEventRecord for high precision.
3. **Scaling Curve:** I will plot execution time vs. N. I expect the time to grow quadratically ($O(N^2)$), but the *efficiency* (GFLOPS) should increase as N grows until the GPU is fully saturated.

4. **Comparison:** I will present a side-by-side comparison of the **Naïve Global Memory Kernel** vs. the **Shared Memory Tiled Kernel** to demonstrate the speedup gained from architectural optimization.