Parallel Programming Assignment 5 Report

b12901069 段奕鳴

December 2, 2025

1 Implementation

1.1 Parallelism Strategy

In high-level, three levels of parallelism are adopted:

- Host-level: P1 and P2 run simultaneously via std::thread
- GPU-level: P3 tasks distributed across both GPUs (in round-robin manner for load-balance)
- Kernel-level:
  - Small N ($\leq$ 256): Single-block fused interval kernel - runs 1000 steps in one launch, all states in register/shared memory
  - Large N ($>$ 256): One-block-per-body kernel - each block cooperatively computes forces on a single body using parallel reduction

1.2 Optimizations

Hybrid Kernel Strategy

- N $\leq$ 256: run_interval_kernel_single_block
  - Runs up to 1000 steps per kernel launch instead of 1 step
  - All data in shared memory (MAX_N = 1024 elements)
  - No global memory traffic during time loop
  - Early exit on collision detection
  - Reduction in kernel launches
- N $>$ 256: step_kernel_one_block_per_body
  - Grid dimension: N blocks (one per body)
  - Block dimension: 256 threads per block
  - Each thread computes partial forces from N/256 bodies
  - Parallel reduction in shared memory to sum all forces
  - Optimized for high number of celestial body counts with cooperative thread work

Memory Optimization

- Constant memory for simulation parameters (cached, broadcast efficiently)
- Pre-computed sin table stored in global memory, avoiding expensive sin() calls
- Optional constant memory for sin table when n_steps is small enough
- Double buffering for position/velocity arrays
- Register caching of particle states in all kernels

Kernel Fusion

- Mass modulation, force calculation, velocity update, and position update fused into single kernel
- Collision detection integrated into main simulation kernel
- Eliminates intermediate global memory writes
- Reduces kernel launch overhead significantly

Atomic Operations

- Custom atomicMinDouble for tracking minimum distance
- atomicCAS for first-hit timestep detection
- Minimal thread contention by having only thread 0 perform atomic updates

1.3 Management of 2-GPU Resources

P1&P2: Dedicated Assignment

```
Simulator sim0(0, ...);  // GPU0 handles P2 (with checkpoints)
Simulator sim1(1, ...);  // GPU1 handles P1 (lightweight)

thread t1([&]() { sim1.run_main(..., true);  });  // P1 on GPU1
thread t2([&]() { sim0.run_main(..., false); });  // P2 on GPU0
```

- GPU0: Heavy workload (P2 + save checkpoints every 1000 steps)
- GPU1: Light workload (P1, no checkpoints, devices disabled)
- No peer-to-peer transfers
- Peer access enabled but not actively used

P3: Dynamic Load Balancing

```
for (int id : device_ids) {
    int step = destroy_steps[id];
    if (step != -1 && step <= params.n_steps) {
        int ckpt = (step / CKPT_INTERVAL);
        if (tasks0.size() <= tasks1.size())
            tasks0.push({id, step, ckpt});
        else
            tasks1.push({id, step, ckpt});
    }
}
```

- Both GPUs receive identical checkpoint data (independent copies)
- Tasks assigned by count in round-robin fashion
- Each GPU runs independent batch kernel (run_p3_batch_kernel)
- One thread block per task, 1024 threads per block
- Each block simulates one scenario from checkpoint to end
- Early termination when collision detected in shared memory flag

Checkpoint System

- Checkpoints saved every CKPT_INTERVAL = 1000 steps
- Flattened storage format: [qx_all, qy_all, qz_all, vx_all, vy_all, vz_all]
- Total checkpoints: ⌈n_steps / 1000⌉ + 1 (including initial state)
- Enables efficient restart from nearest checkpoint for P3 scenarios

2 Problem: If there are 4 GPUs instead of 2, what would you do to maximize the performance?

For smaller number of celestial bodies, the task parallelism would be more efficient than data parallelism of splitting particles across GPUs.

For problem 3, the simulation requires checking potential "destruction steps" to see if the planet is saved. The search space can be divided into 4 partitions on 4 available GPUs. This provides nearly linear scaling for this searching. Instead of streaming data between GPUs, the baseline checkpoint data could be broadcast to the VRAM of all 4 GPUs at the start.

3 Problem: If there are 5 gravity devices, is it necessary to simulate 5 n-body simulations independently for this problem?

No, full independent simulations from initial are unnecessary. The state of the universe before the time of first device is destroyed is identical.

This can be optimized using memorization and checkpointing. Run a single simulation from step 0 to the end assuming no devices are destroyed. Save the full system state into a buffer for every K steps.

To simulate the 5 separate gravity devices, we do not start from t=0. Instead, for a device destruction at step $t_{destroy}$:

- Find the nearest checkpoint $t_{ckpt} \leq t_{destroy}$
- Load the state
- Simulate forward to $t_{destroy}$, apply the mass change (destruction), and continue checking for collisions

4 Suggestions and Feedback

The N-body problem is well-known, and this assignment gave me a chance to tackle it hands-on. Although we were not required to implement the Barnes–Hut algorithm or other approximation methods, the lectures and readings still helped me gain a deeper understanding of how this problem has been addressed in previous work.

In addition, previous assignments and experience focused mainly on NVIDIA GPUs and CUDA, so this was my first time working with AMD GPUs. I believe this was a great opportunity to avoid limiting myself to a single hardware platform and instead become more familiar with diverse hardware environments. However, I noticed that some ROCm profiling tools were not functioning as expected on the server. I am not sure what caused the issue, so most of the time I had to perform optimizations without detailed kernel profiling information.