Parallel Programming Assignment 5 Report

b12901069 段奕鳴

December 1, 2025

1 Implementation

1.1 Parallelism Strategy

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

- Host-level: P1 and P2 run simulaneously via std::thread
- GPU-level: P3 task 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): Multi-block tiled kernel with 256-thread shared memory tiles

1.2 Optimizations

Hybrid Kernel Strategy

- N $\leq$ 256: run_interval_kernel_single_block
  - Runs 1000 steps per kernel launch instead of 1 step
  - All data in shared memory (24KB for 3 position arrays)
  - No global memory traffic during time loop
  - Early exit on collision detection
- N $>$ 256: step_kernel_large_fused
  - Tiled force calculation (256-element tiles)
  - Fused mass calculation + force + update in one kernel
  - Reduces kernel launch overhead from 200k to 200

Memory Optimization

- Constant memory for parameters (cached)
- Pre-computed sin table, avoid sin() calls
- Async memcpy with streams, overlap compute/transfer

Kernel Fusion

- Original: 4 kernels per step ($mass \rightarrow force \rightarrow update \rightarrow check$)
- Fusion: 1 kernel per step, eliminates 3 kernel launches overhead per step

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)
- GPU1: Light workload (P1, no checkpoints)
- No peer-to-peer transfers needed (independent problems)

P3: Dynamic load balancing

```
for (int id : device_ids) {
    if (tasks0.size() <= tasks1.size())
        tasks0.push({id, destroy_step, ckpt_idx});
    else
        tasks1.push({id, destroy_step, ckpt_idx});
}
```

- Both GPUs receive identical checkpoint data (broadcast on each)
- Tasks assigned by count (simpler than workload-aware)
- Each GPU runs independent batch kernel

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.