N-body Simulation: Implementation Report

學號: R14922156
姓名: 黃泓諺

2025 年 11 月 30 日

Contents

1 Implementation

1.1 Parallelism Strategy

本專案採用 **Dynamic Work Dispatch with Multi-GPU** 策略，核心設計包含三個主要階段的平行化：

1.1.1 Phase 1 & 2: Multi-GPU Parallel Execution

```
// P1 (無裝置) 和 P2 (有裝置) 在不同 GPU 上同時執行
std::thread t1([&]() {
    cudaSetDevice(0);
    p1_min_dist = simulate_phase1(dev0, ...);
});

std::thread t2([&]() {
    cudaSetDevice(1);
    p2_result = simulate_phase2(dev1, ...);
});

t1.join();
t2.join();
```

Listing 1: P1 和 P2 平行執行

設計理由:

- P1 和 P2 是完全獨立的模擬任務
- 使用兩張 GPU 達成真正的並行計算
- 避免 GPU 資源閒置

1.1.2 Phase 3: Dynamic Task Queue with Load Balancing

```
std::atomic<int> task_queue(0);

void worker_thread(int gpu_id) {
    cudaSetDevice(gpu_id);
    while (true) {
        int device_id = task_queue.fetch_add(1, std::
            memory_order_relaxed);
        if (device_id >= n_device) break;

        // 為此裝置模擬飛彈追蹤
        simulate_device_trajectory(gpu_id, device_id, ...);
    }
}

// 啟動兩個 worker threads
std::thread worker0(worker_thread, 0);
std::thread worker1(worker_thread, 1);
```

Listing 2: 動態任務分配

Dynamic Dispatch 的優勢:

方法	負載分配	GPU 利用率	效能
靜態分割	固定前 2 個裝置給 GPU0	不均衡	7.29s
動態分配	GPU 自動搶任務	高度平衡	**5.39s (↑35%)**

Table 1: 靜態 vs 動態分配比較

關鍵改善：P3 階段從 5.20s 降至 3.20s（↑**38%** 效能）

1.2 Optimization Techniques

根據 benchmark 測試結果，以下是關鍵優化技術及其影響：

1.2.1 Critical Optimizations (High Impact)

1. Shared Memory Reduction (29.18x speedup)

```
__global__ void compute_force_kernel(...) {
    __shared__ double s_ax[BLOCK_SIZE];
    __shared__ double s_ay[BLOCK_SIZE];

    // 每個 thread 計算部分加速度
    double ax = 0, ay = 0;
    for (int j = 0; j < n; j++) {
        // 計算引力...
        ax += fx;
        ay += fy;
    }

    // Shared memory parallel reduction
    s_ax[tid] = ax;
    s_ay[tid] = ay;
    __syncthreads();

    for (int offset = blockDim.x >> 1; offset > 0; offset >>= 1) {
        if (tid < offset) {
            s_ax[tid] += s_ax[tid + offset];
            s_ay[tid] += s_ay[tid + offset];
        }
        __syncthreads();
    }
}
```

Listing 3: Shared Memory 平行歸約

為何關鍵:

- 將 $O(N)$ 的序列歸約變為 $O(\log N)$ 的平行歸約
- Shared memory 延遲 (∼5 cycles) 遠低於 global memory (∼400 cycles)
- 移除此優化導致執行時間從 7.55s 暴增至 220.37s

2. Batching (13-18x speedup)

```
constexpr int BATCH_SIZE = 200;

for (int batch = 0; batch < n_steps; batch += BATCH_SIZE) {
    int batch_size = std::min(BATCH_SIZE, n_steps - batch);

    // Kernel 在 GPU 連續執行 200 步
    compute_batch_kernel<<<blocks, threads, 0, stream>>>(
        batch_size, dt, ...
    );

    // 批次結束才同步一次
    cudaStreamSynchronize(stream);
}
```

Listing 4: 批次處理

效能影響:

- 減少 CPU-GPU 同步次數：200,000 → 1,000 次
- 每次 `cudaStreamSynchronize()` 有 ∼10-50 μs 固定開銷
- `BATCH_SIZE=1` 導致執行時間增加 **13.48x**
- 每步同步導致執行時間增加 **17.98x**

1.2.2 Medium Impact Optimizations

3. Fast Math Intrinsics (`rsqrt()` - 1.09x speedup)

```
// 優化版本: 單一硬體指令
double inv = rsqrt(dist);

// 未優化: 兩次運算
double inv = 1.0 / sqrt(dist);
```

Listing 5: rsqrt 內建函數

- GPU 提供專用硬體指令 `rsqrt()`
- 在 N-body 模擬中，每對天體計算引力都需要此運算
- 累積效果可觀（節省 0.72 秒）

4. Memory Coalescing & Alignment

```
// 優化版本：告訴編譯器指標不會重疊
__global__ void kernel(
```

```
    const double* __restrict__ qx_in,
    double* __restrict__ qx_out,
    const double* __restrict__ qy_in,
    double* __restrict__ qy_out
)
```

Listing 6: 使用 ___restrict___ 提示

- 告訴編譯器指標不會重疊
- 允許更激進的記憶體存取優化
- 影響約 1.01x（在此 workload 下）

1.2.3 Configuration Optimizations

5. Block Size Tuning

```
constexpr int BLOCK_SIZE = 256;  // 經實驗驗證的最佳值
```

Listing 7: Block Size 設定

測試結果:

- Block Size = 32: 3.38s（但輸出錯誤 ×）
- Block Size = 256: 7.55s（正確 ✓）

選擇 256 因為：

- 充分利用 SM (Streaming Multiprocessor)
- 足夠的 warps 隱藏延遲
- 避免 shared memory bank conflicts

1.3 Two-GPU Resource Management

1.3.1 Resource Allocation Strategy

GPU 資源分配架構

Phase 1 & 2 (Parallel)
- GPU 0: P1 Simulation + CUDA Stream 0 + Pinned Memory
- GPU 1: P2 Simulation + CUDA Stream 0 + Pinned Memory

⇓

Phase 3 (Dynamic Dispatch)
- GPU 0 Worker: Task Queue + CUDA Stream 0 + Device 0,2,4... (dynamic)
- GPU 1 Worker: Task Queue + CUDA Stream 0 + Device 1,3,5... (dynamic)

1.3.2 Memory Management

```
struct GPUContext {
    int device_id;
    cudaStream_t stream;
```

```
    // Device memory (重複使用同一塊記憶體)
    double *d_qx, *d_qy, *d_vx, *d_vy, *d_m;
    double *d_qx_out, *d_qy_out, *d_vx_out, *d_vy_out;

    // Pinned host memory (加速傳輸)
    double *h_qx, *h_qy;
};

// 初始化兩個 GPU contexts
GPUContext gpu_ctx[2];
for (int i = 0; i < 2; i++) {
    cudaSetDevice(i);
    cudaStreamCreate(&gpu_ctx[i].stream);
    cudaMalloc(&gpu_ctx[i].d_qx, size);
    cudaMallocHost(&gpu_ctx[i].h_qx, size);  // Pinned
}
```

Listing 8: GPU Context 結構

關鍵設計:

1. **Workspace Reuse**: 每個 GPU 的 device memory 在所有模擬中重複使用
2. **Pinned Memory**: 使用 `cudaMallocHost` 加速 CPU-GPU 傳輸
3. **Independent Streams**: 每個 GPU 有獨立的 CUDA stream，避免同步干擾

1.3.3 Load Balancing Mechanism

```
// 原子操作實現無鎖任務隊列
std::atomic<int> task_queue_head(0);

int get_next_task() {
    return task_queue_head.fetch_add(1, std::memory_order_relaxed);
}
```

Listing 9: 無鎖任務隊列

為何有效:

- P3 階段各裝置追蹤的飛彈數量不同
- 動態分配讓快速完成的 GPU 自動接手更多任務
- 避免靜態分割造成的 GPU 閒置時間

實測負載分配:

靜態分割:
- GPU 0: Device 0,1 (70% 利用率)
- GPU 1: Device 2,3 (100% 利用率)

動態分配:
- GPU 0: Device 0,2,4 (98% 利用率)
- GPU 1: Device 1,3 (97% 利用率)

2 Scaling to 4 GPUs

若系統擁有 4 張 GPU，可採用以下策略最大化效能：

2.1 Proposed Architecture

4-GPU 架構

Phase 1 & 2: 繼續使用 **2 GPUs** (不變)
- GPU 0: P1 (無裝置質量)
- GPU 1: P2 (有裝置質量)

Phase 3: 擴展至 **4 GPUs** (主要改進)
- GPU 0: Worker Thread 0
- GPU 1: Worker Thread 1
- GPU 2: Worker Thread 2 ← 新增
- GPU 3: Worker Thread 3 ← 新增

2.2 Implementation Strategy

```
constexpr int N_GPUS = 4;
std::atomic<int> task_queue(0);

void worker_thread(int gpu_id) {
    cudaSetDevice(gpu_id);
    GPUContext ctx = init_gpu_context(gpu_id);

    while (true) {
        int device_id = task_queue.fetch_add(1, std::
           memory_order_relaxed);
        if (device_id >= n_device) break;

        simulate_device_trajectory(ctx, device_id, ...);
    }

    cleanup_gpu_context(ctx);
}

// Phase 3: 啟動 4 個 workers
std::vector<std::thread> workers;
for (int i = 0; i < N_GPUS; i++) {
```

```
    workers.emplace_back(worker_thread, i);
}

for (auto& t : workers) t.join();
```

Listing 10: 4-GPU 實作

2.3 Expected Performance Gain

理論分析:

假設 P3 階段為主要瓶頸（當前約 3.2s），且任務可完美並行：

GPU 數量	預期 P3 時間	總時間	Speedup
2 GPUs	3.20s	5.39s	1.00x (基準)
4 GPUs	∼1.60s	∼3.79s	**1.42x**

Table 2: 4-GPU 效能預估

實際考量:

- 完美線性擴展不太可能（Amdahl's Law）
- 可能的瓶頸：
  1. PCIe 頻寬競爭（4 個 GPU 同時傳輸）
  2. CPU 端的原子操作競爭
  3. 記憶體頻寬限制

優化建議:

```
// 減少原子操作競爭：批次取任務
int get_batch_tasks(int batch_size = 4) {
    return task_queue.fetch_add(batch_size, std::memory_order_relaxed
        );
}
```

Listing 11: 批次取任務減少原子操作競爭

2.4 Alternative: Hierarchical GPU Assignment

若 P1+P2 也成為瓶頸，可考慮：

Phase 1: 2 GPUs 平行計算
- GPU 0: P1 前半 (天體 0 ∼ n/2)
- GPU 1: P1 後半 (天體 n/2 ∼ n)

Phase 2: 2 GPUs 平行計算
- GPU 2: P2 前半
- GPU 3: P2 後半

Phase 3: 全部 **4 GPUs**

但此方法需要額外的 reduction 步驟，可能不劃算。

3 Multiple Gravity Devices Analysis

3.1 Problem Statement

若有 **5** 個引力裝置，是否需要獨立模擬 5 次 N-body simulation？

答案：否，不需要

3.2 Mathematical Justification

引力場的線性疊加原理 (Superposition Principle)：

$$\vec{F}_{\text{total}} = \vec{F}_1 + \vec{F}_2 + \vec{F}_3 + \vec{F}_4 + \vec{F}_5 \tag{1}$$

對於 N-body simulation：

$$\vec{a}_i = \sum_{j \neq i} \frac{G m_j (\vec{r}_j - \vec{r}_i)}{|\vec{r}_j - \vec{r}_i|^3} \tag{2}$$

其中 m_j 可以包含裝置質量：

$$m_j = \begin{cases} m_{\text{planet},j} + \sum_{k=1}^{5} m_{\text{device},k} & \text{if planet } j \text{ 上有裝置} \\ m_{\text{planet},j} & \text{otherwise} \end{cases} \tag{3}$$

3.3 Efficient Implementation

3.3.1 方法一：Bitmask 表示裝置組合（當前實作）

```
// 使用 bitmask 表示裝置組合 (2^n_device 種可能)
for (int mask = 0; mask < (1 << n_device); mask++) {
    // 設定質量
    for (int i = 0; i < n; i++) {
        double mass = mass_planet[i];

        // 檢查每個裝置是否在此組合中
        for (int d = 0; d < n_device; d++) {
            if ((mask & (1 << d)) && device_on_planet[d] == i) {
                mass += device_mass[d];
            }
        }

        m[i] = mass;
    }
```

```
16
17      // 執行一次 N-body simulation
18      run_simulation(m, ...);
19  }
```

Listing 12: Bitmask 裝置組合

複雜度: $O(2^{n_{\text{device}}} \times T \times n^2)$

對於 5 個裝置：$2^5 = 32$ 次模擬

3.3.2 方法二：**On-the-fly** 質量調整（更高效）

```
1   // 初始化：所有裝置都開啟
2   double m[n];
3   for (int i = 0; i < n; i++) {
4       m[i] = mass_planet[i];
5       for (int d = 0; d < n_device; d++) {
6           if (device_on_planet[d] == i) {
7               m[i] += device_mass[d];
8           }
9       }
10  }
11
12  // 主迴圈：動態調整質量
13  for (int mask = 0; mask < (1 << n_device); mask++) {
14      // 只更新改變的裝置
15      int changed = mask ^ prev_mask;
16      for (int d = 0; d < n_device; d++) {
17          if (changed & (1 << d)) {
18              int planet_id = device_on_planet[d];
19              if (mask & (1 << d)) {
20                  m[planet_id] += device_mass[d];  // 開啟
21              } else {
22                  m[planet_id] -= device_mass[d];  // 關閉
23              }
24          }
25      }
26
27      run_simulation(m, ...);
28      prev_mask = mask;
29  }
```

Listing 13: 動態質量調整

3.4 Why Not Independent Simulations?

方法	模擬次數	記憶體使用	優點	缺點
獨立模擬 **5** 次	5	5x	簡單	浪費計算
Bitmask 組合	$2^5 = 32$	1x	涵蓋所有情況	組合爆炸

Table 3: 不同模擬方法比較

關鍵洞察:

- P1 只需要一次模擬（無裝置）
- P2 只需要一次模擬（全部裝置開啟）
- P3 需要 $2^{n_{\text{device}}}$ 次模擬（所有組合）

若獨立模擬 5 次：

```
Simulation 1: Device 0 on
Simulation 2: Device 1 on
Simulation 3: Device 2 on
Simulation 4: Device 3 on
Simulation 5: Device 4 on
```

Listing 14: 獨立模擬的問題

問題

無法回答「當 Device 0 和 Device 2 同時開啟時」的情況！
因為引力是非線性的組合（距離的三次方），不能簡單地將兩次模擬結果相加。

3.5 Optimization for Many Devices

若裝置數量很多（如 10 個），$2^{10} = 1024$ 次模擬：

3.5.1 策略一：Monte Carlo Sampling

```cpp
// 隨機採樣重要的組合
std::vector<int> important_masks;
for (int i = 0; i < n_samples; i++) {
    int mask = generate_random_mask();
    important_masks.push_back(mask);
}
```

Listing 15: 隨機採樣

3.5.2 策略二：Importance Sampling

```cpp
// 只模擬可能影響結果的組合
// 例如：只考慮飛彈會經過的裝置組合
```

Listing 16: 重要性採樣

3.5.3 策略三：Early Termination

```
// 若某組合下飛彈已確定會擊中，提前終止模擬
if (missile_distance < hit_threshold) {
    mark_hit_and_skip();
}
```

Listing 17: 提前終止

4 Performance Summary

4.1 Optimization Impact Ranking

優化技術	Speedup	影響
Shared Memory Reduction	29.18x	⋆⋆⋆⋆⋆
Batching (每步同步)	17.98x	⋆⋆⋆⋆⋆
Batching (BATCH_SIZE=1)	13.48x	⋆⋆⋆⋆⋆
Dynamic Dispatch (Strategy C)	1.35x	⋆⋆⋆⋆
rsqrt()	1.09x	⋆⋆⋆
Multi-GPU	1.08x	⋆⋆⋆
__restrict__	1.01x	⋆
Async / Pinned / Precompute sin	∼1.00x	⋆

Table 4: 優化技術影響力排名

4.2 Final Performance

階段	時間 (s)	百分比
P1+P2	2.08	38.6%
P3	3.20	59.4%
其他	0.11	2.0%
總計	**5.39**	100%

Table 5: 最終效能分析

相比原始 **Baseline (7.29s)**:

- Speedup: **1.35x**
- 時間節省: **1.90s**
- 主要改善: P3 階段的動態負載均衡

4.3 GPU Utilization

GPU 使用率 (NVIDIA-SMI 觀察)

GPU 使用率:
- GPU 0: 95-98%
- GPU 1: 95-98%

記憶體使用:
- GPU 0: 2.1 GB / 24 GB (9%)
- GPU 1: 2.1 GB / 24 GB (9%)

5 Conclusion

本專案成功實現了高效能的 N-body 模擬系統，關鍵成果包括：

1. **Core Optimizations**:
   - Shared Memory Reduction: **29x** 加速
   - Batching: **13-18x** 加速
   - Dynamic Work Dispatch: **1.35x** 額外加速
2. **Multi-GPU Strategy**:
   - Phase 1&2: 靜態分配（獨立任務）
   - Phase 3: 動態任務隊列（負載均衡）
   - 整體 GPU 利用率 $> 95\%$
3. **Scalability**:
   - 2 GPUs $\rightarrow$ 4 GPUs: 預期 **1.4x** 額外加速
   - 策略可擴展至更多 GPU
4. **Mathematical Correctness**:
   - 利用疊加原理，無需獨立模擬每個裝置
   - $2^{n_{\text{device}}}$ 組合涵蓋所有可能性

未來改進方向:

- CUDA Graph 減少 kernel launch overhead
- Unified Memory 簡化記憶體管理
- Multi-Stream 進一步重疊計算與傳輸