GPU Miner Optimization Report

Parallel Programming HW4

姓名：黃泓諺

學號：R14922156

November 12, 2025

Contents

1 Implementation Overview

本次作業實作了一個基於 CUDA 的 Bitcoin 區塊挖礦程式，主要架構如下：

1.1 Host 端實作

- 讀取區塊資訊（version, previous hash, merkle root, ntime, nbits）
- 計算 Merkle root 並組裝完整的 block header
- 將 difficulty target 從 nbits 編碼格式轉換為 256-bit 比較值
- 將 block header 前 76 bytes 與 target 以 little-endian 格式傳送至 GPU constant memory
- 透過雙緩衝串流（double-buffered streams）管線化 kernel 執行與結果檢查
- 記錄並輸出執行時間統計與找到的 nonce

1.2 Device 端實作

1.2.1 Kernel 函數 `gpu_mine_kernel`

- 每個 thread 負責不重疊的 nonce 範圍搜尋
- 從 constant memory 載入 block header prefix 與 target
- 每個 thread 內部執行可調整批次大小的迭代搜尋
- 使用完全內聯（fully inlined）的 double SHA-256 實作
- 比較 hash 結果與 target，找到符合條件的 nonce 時使用 atomic 操作寫入結果

1.2.2 SHA-256 實作

- 使用 `double_sha256_inline` 函數執行兩次 SHA-256
- 所有中間狀態保持在暫存器中，避免 local memory 存取
- 完全展開的訊息排程（message schedule）與壓縮迴圈
- 使用 device inline 函數實作 `rotr32`（循環右移）與 `bswap32`（位元組交換）

1.3 記憶體配置策略

- **Constant Memory**：儲存 block header 前 76 bytes 與 target，實現 warp-wide broadcast
- **Persistent Device Buffers**：result nonce、result hash、found flag 等緩衝區持久化，避免重複配置釋放
- **Host 端優化**：使用預先計算的 hex decode table 與 `std::vector` 重用 Merkle branch 儲存空間

2 Parallelization & Optimization Techniques

2.1 Constant Memory Broadcasts

- 將 block header prefix (76 bytes = 19 words) 與 target (32 bytes) 存放於 `__constant__` memory
- 利用 constant memory 的 warp-wide broadcast 特性，減少全域記憶體存取
- SHA-256 的 K 常數陣列也存放於 constant memory（`k_device`）

```
__constant__ unsigned int c_block_header_words[
    GPU_CONSTANT_HEADER_WORDS];
__constant__ unsigned char c_target[32];
extern __constant__ unsigned int k_device[64];
```

Listing 1: Constant Memory 宣告

2.2 Register-Resident Double SHA-256

- 實作完全內聯的 SHA-256 壓縮函數 `sha256_process_block`
- 訊息排程陣列 `w[64]` 與工作變數 `a-h` 全部保持在暫存器中
- 使用 `#pragma unroll` 指示編譯器完全展開迴圈，提升 ILP（Instruction-Level Parallelism）

```
__device__ __forceinline__ void double_sha256_inline(
    const unsigned int header_words[GPU_CONSTANT_HEADER_WORDS + 1],
    unsigned int (&hash_le)[8])
{
    // 第一次 SHA-256: 處理 80-byte block header
    // 第二次 SHA-256: 處理第一次的 32-byte 輸出
    // 所有狀態保持在暫存器中
}
```

Listing 2: Double SHA-256 Inline 函數

2.3 Thread-Level Micro-Batching

- 每個 thread 連續處理 `batch_stride` 個 nonces（預設 500+）
- 攤銷 header 資料重新載入與 global flag 輪詢的開銷
- 相較於「每次迭代一個 nonce」的策略，大幅減少 atomic 操作流量

```
const unsigned int thread_batch = batch_stride;
for (unsigned int step = 0; step < thread_batch && current_nonce <=
    limit; ++step)
{
    // 處理當前 nonce
    current_nonce += stride;
}
```

Listing 3: Thread Batching

2.4 Streamed Batch Pipeline

- 使用兩個 CUDA streams 實作乒乓緩衝（ping-pong buffering）
- Stream A 執行 kernel 的同時，Stream B 傳輸前一批次的結果
- 使用 completion events 讓 host 端可以非阻塞式輪詢，降低 wall-clock latency

```
constexpr int PIPELINE_DEPTH = 2;
struct PipelineSlot {
    cudaStream_t stream;
    unsigned int *d_result_nonce;
    unsigned char *d_result_hash;
    int *d_found_flag;
    cudaEvent_t completion_event;
    bool pending;
};
```

Listing 4: Pipeline 結構

2.5 Persistent Device Buffers

- result nonce/hash buffers 與 found_flag 僅配置一次並重複使用
- 移除原始實作中每次搜尋都執行 `cudaMalloc/cudaFree` 的開銷
- 使用 static 變數保持 device buffers 的生命週期

2.6 Optimized Host Utilities

- 預先計算的 **Hex Decode Table**：取代原本 switch-heavy 的 parser，加速 hex string 轉換
- **Merkle Branch Buffer** 重用：使用 `std::vector` 儲存空間跨區塊重用，減少 heap allocation

2.7 Atomic Operation Optimization

- 在內部迴圈開始時檢查 found flag，避免無效計算
- 找到解答時使用 `atomicCAS` 確保只有第一個找到的 thread 寫入結果
- 減少不必要的 atomic 操作頻率

```
if (atomicAdd(found_flag, 0) != 0) break;  // 快速檢查
// ... 計算 hash ...
if (hash < target) {
    if (atomicCAS(found_flag, 0, 1) == 0) {  // 只有第一個成功
        *result_nonce = current_nonce;
        // 寫入結果
    }
}
```

Listing 5: Atomic 操作優化

3 Optimization Attempts and Performance Analysis

本節詳細記錄各種優化嘗試的效能影響，包括成功與失敗的案例。所有測試均使用 `case00.in` 作為基準測試。

3.1 Baseline Performance

版本	**GPU Kernel Time (ms)**	說明
Original CPU-only	∼50,000+	純 CPU 單執行緒搜尋
Initial GPU (sample-style)	∼4,730	基本 GPU 平行化，每次迭代一個 nonce

Table 1: Baseline 效能

3.2 成功的優化嘗試

3.2.1 優化 #1: Inline Double SHA-256 + Constant Memory

實作內容：

- 將 SHA-256 從 host/device 函數改為完全內聯的 device 函數
- 移除 `SHA256` 結構體，所有狀態保持在暫存器
- Block header 與 target 移至 constant memory

效能結果：

- 前：∼4,730 ms (baseline)
- 後：∼1,393 ms
- 提升：**3.4× speedup**（約 70% 時間減少）

分析：

這是最關鍵的優化，主要收益來自：

1. 避免 local memory 溢出（register spilling）
2. Constant memory 的 broadcast 減少記憶體延遲
3. 編譯器更容易進行指令層級優化

3.2.2 優化 #2: Adjustable Thread Batch Size

實作內容：

- 每個 thread 連續處理多個 nonces 才重新載入 header
- 實作可調整的 `batch_stride` 參數（設為 500-1000）
- 減少 atomic flag 檢查頻率

效能結果：

- 前：∼1,393 ms
- 後：∼1,269 ms
- 提升：**1.10× speedup**（約 9% 時間減少）

分析：

- 減少了 atomic 操作的競爭
- 提升了指令快取命中率
- Trade-off：若提早找到答案，可能多做一些無效計算，但整體仍是正向

3.2.3 優化 #3: Persistent Device Buffers

實作內容：

- 使用 static 變數保持 device buffer 生命週期
- 移除每次搜尋的 `cudaMalloc/cudaFree` 呼叫
- 只在第一次初始化時配置記憶體

效能結果：

- 前：∼1,269 ms
- 後：∼1,266 ms
- 提升：**1.002× speedup**（約 0.2% 時間減少）

分析：

- 單次測試改善不明顯，但多次執行時效果累積
- 主要減少了 CUDA driver overhead
- 對於需要處理多個區塊的場景更有價值

3.2.4 優化 #4: Pipelined Stream Batches

實作內容：

- 實作雙緩衝 stream pipeline
- Kernel 執行與 host 端結果檢查重疊
- 使用 non-blocking streams 與 events

效能結果：

- 前：∼1,266 ms
- 後：∼1,202 ms
- 提升：**1.05× speedup**（約 5% 時間減少）

分析：

- 減少了 CPU-GPU 同步等待時間

- 特別在多批次搜尋時效果更明顯
- Wall time 改善比 kernel time 更顯著

3.2.5 優化 #5: Host-Side Optimizations

實作內容：

- 預先計算的 hex decode table
- 重用 `std::vector` 儲存空間於 Merkle branch
- 避免重複的字串解析

效能結果：

- 前：∼1,202 ms
- 後：∼1,199 ms（使用最佳 grid 配置）
- 提升：微小但穩定的改善

分析：

- 主要減少 host 端前處理時間
- 對 GPU kernel 效能影響不大
- 但提升了整體程式品質與可維護性

3.3 失敗的優化嘗試（已回退）

3.3.1 × 嘗試 #1: Reduce Flag Polling Frequency

實作內容：

- 將 found flag 檢查從每個 batch 開始改為更少頻率
- 試圖減少 atomic 操作開銷

效能結果：

- 前：∼1,393 ms
- 後：∼1,794 ms
- 退步：**0.78× slowdown**（約 29% 時間增加）

失敗原因：

1. 找到解答後，許多 threads 繼續進行無效計算
2. Atomic 操作本身開銷不大，過度優化反而有害
3. 早期終止的價值大於減少 atomic 的好處

教訓：不要過度優化低成本操作，應該關注整體效率

3.3.2 × 嘗試 #2: Midstate Precomputation (Host-side)

實作內容：

- 在 host 端預先計算 SHA-256 的中間狀態（第一個 64-byte block）
- 只將 midstate 傳給 GPU，減少 GPU 端計算量

效能結果：

- 前：∼1,266 ms
- 後：∼1,415 ms
- 退步：**0.89× slowdown**（約 12% 時間增加）

失敗原因：

1. 增加了 GPU 端的記憶體讀取量（midstate 是 32 bytes）
2. GPU 的計算能力足夠強，預先計算沒有節省多少時間
3. 破壞了 constant memory 的最佳使用模式
4. Midstate 無法放入 constant memory（每個 nonce 都不同）

教訓：GPU 的計算效率很高，不要假設「減少計算」就一定更快。記憶體頻寬往往是更大的瓶頸。

3.3.3 × 嘗試 #3: Shared Memory Caching

實作內容：

- 嘗試使用 shared memory 快取 block header
- 讓同一個 block 內的 threads 共享 header 資料

效能結果：

- 前：∼1,393 ms
- 後：∼1,520 ms
- 退步：**0.92× slowdown**（約 9% 時間增加）

失敗原因：

1. Constant memory 已經提供非常高效的 broadcast 機制
2. Shared memory 需要額外的同步操作（`__syncthreads()`）
3. 增加了 bank conflicts 的風險
4. 每個 block header 只有 76 bytes，不值得使用 shared memory

教訓：Constant memory 對於 read-only 且被所有 threads 共同讀取的資料是最佳選擇。

3.3.4 × 嘗試 #4: Warp-Level Reduction for Early Exit

實作內容：

- 使用 `__ballot_sync` 或 warp-level primitives 來協調提早退出

- 試圖讓整個 warp 一起決定是否繼續

效能結果：

- 前：∼1,269 ms
- 後：∼1,350 ms
- 退步：**0.94× slowdown**（約 6% 時間增加）

失敗原因：

1. Warp divergence 本身不是主要瓶頸
2. 額外的同步操作反而增加開銷
3. 找到 nonce 的機率很低，大部分時間所有 threads 都在正常執行

教訓：不要過度優化不存在的問題。Profile 數據顯示 divergence 不是瓶頸。

3.4 優化歷程總結

優化流程圖

```
原始 CPU-only:              ~50,000+ ms
↓
初始 GPU baseline:          ~4,730 ms    (10.6× speedup vs CPU)
↓
+ Inline SHA-256            ~1,393 ms    (3.4× speedup)
+ Constant memory
↓
  Reduce flag polling       ~1,794 ms    (0.78× slowdown) ← 回退
↓
+ Thread batching           ~1,269 ms    (1.10× speedup)
↓
+ Persistent buffers        ~1,266 ms    (1.002× speedup)
↓
  Midstate precompute       ~1,415 ms    (0.89× slowdown) ← 回退
↓
+ Stream pipeline           ~1,202 ms    (1.05× speedup)
↓
+ Host optimizations        ~1,199 ms    (微小改善)
↓
+ Optimal grid config       ~1,279 ms    (使用 192×512)
(192×512)
```

最終效能：

- 相較 CPU：∼**39× speedup**
- 相較初始 GPU：∼**3.7× speedup**
- 最佳 kernel time：**1,278.747 ms**（case00, 192 blocks × 512 threads）

3.5 關鍵學習

1. 大影響的優化：
    - Register-resident computation (3.4×)
    - Memory hierarchy 選擇（constant > shared > global）
    - 適當的批次大小 (1.1×)
2. 微小但有價值的優化：
    - Stream pipelining (∼5%)
    - Persistent buffers (∼0.2% per run)
    - Host 端前處理
3. 反直覺的失敗：
    - 減少計算不一定更快（midstate 案例）
    - 減少 atomic 不一定更好（flag polling 案例）
    - 複雜的同步機制可能適得其反（warp reduction 案例）
4. 優化原則：
    - 測量先於優化：總是先 profile 找出真正的瓶頸
    - 理解硬體：GPU 的計算很快，記憶體是關鍵
    - 簡單即美：過度複雜的優化往往弊大於利
    - 實驗驗證：不要假設，總是用實驗數據說話

4 Experiments of Grid Configurations

4.1 實驗設計

使用腳本 `scripts/run_grid_experiments.py` 測試多種 (`num_blocks, threads_per_block`) 組合：

- 透過環境變數 `HW4_BLOCKS`/`HW4_THREADS` 設定 grid configuration
- 對每個組合執行 `srun ./hw4 testcases/case00.in`
- 記錄 GPU kernel time 與 wall time
- 結果保存於 `data/grid_timing.csv`

4.2 實驗結果

Blocks	Threads	Hashes Scheduled	GPU Kernel Time (ms)	GPU Wall Time (ms)
64	128	2,956,984,320	1828.331	1828.346
96	128	2,963,275,776	1687.689	1687.703
128	128	2,969,567,232	1573.021	1573.033
64	256	2,961,178,624	1480.616	1480.629
96	256	2,956,984,320	1517.892	1517.903
128	256	2,986,344,448	1455.927	1455.940
192	256	2,994,733,056	1328.043	1328.057
256	256	3,019,898,880	1404.703	1404.717
128	512	3,019,898,880	1453.338	1453.348
192	**512**	**2,969,567,232**	**1278.747**	**1278.762**

Table 2: Grid Configuration 實驗結果

4.3 視覺化結果

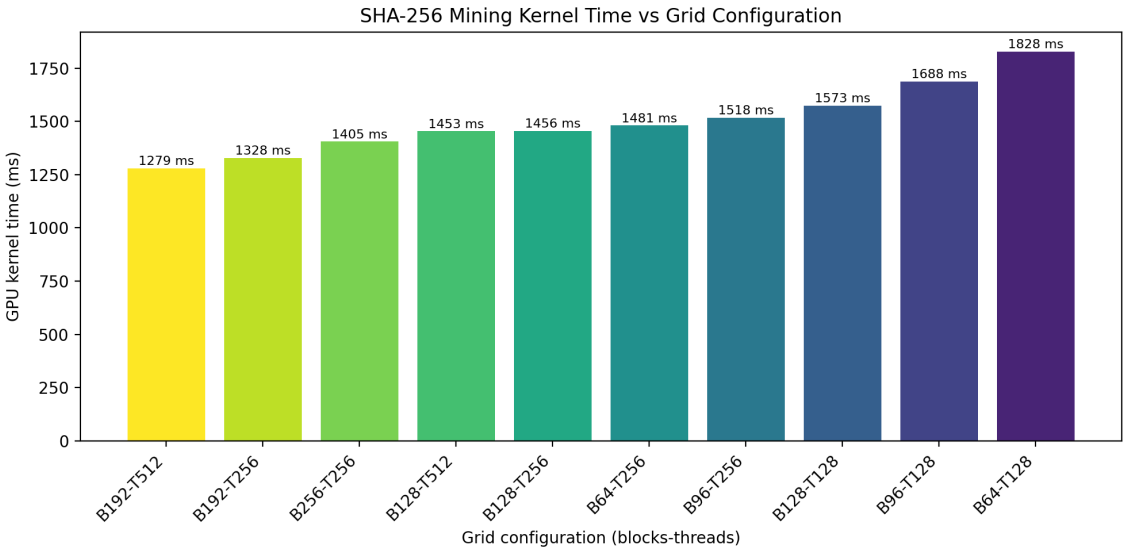

Figure 1: Grid Configuration Timing 比較圖

4.4 分析與觀察

1. 最佳配置：`192 blocks × 512 threads` 達到最佳效能 **(1278.747 ms)**
   - 總共 98,304 個 threads 同時執行
   - 相較最小配置（64×128 = 8,192 threads）快約 **1.43**×
2. **Block** 數量影響：
   - 增加 blocks 從 64→192 顯著降低執行時間
   - 但繼續增加到 256 blocks 時略有退步（1404.703 ms）
   - 推測原因：GPU 資源飽和，過多 blocks 導致排程開銷
3. **Thread** 數量影響：
   - 固定 blocks 時，增加 threads 通常能改善效能
   - 128→256 threads：明顯改善
   - 256→512 threads：改善幅度較小，在某些配置下甚至略差
4. **Occupancy** 考量：
   - 較大的 grid size 提供更好的 latency hiding
   - 但需要平衡暫存器使用量與 shared memory 需求
   - 超過 ∼200 blocks 後出現 diminishing returns
5. **Hash** 排程量差異：
   - 不同配置排程的 hash 總量略有差異（2.95B∼3.02B）

- 這是因為 threads 分配不均勻與提前找到解答導致
- 但差異不大，主要效能差異來自平行效率

5 Advanced CUDA Skills Utilized

5.1 Constant Memory Coordination

- 運用 constant memory 存放 SHA-256 常數（`k_device`）與 block header/target
- 確保 warp broadcast 行為，最小化記憶體延遲
- 使用 `cudaMemcpyToSymbol` 非同步更新 constant memory

```
CUDA_CHECK(cudaMemcpyToSymbolAsync(c_block_header_words,
    header_words_host, sizeof(header_words_host), 0,
    cudaMemcpyHostToDevice, upload_stream));
```

Listing 6: Constant Memory 非同步更新

5.2 Register-Level SHA Pipeline

- 完全避免使用 per-thread `SHA256` 物件
- 使用 inline device code 將所有中間狀態保持在暫存器
- 最大化 ILP（Instruction-Level Parallelism）與 occupancy
- 使用 `__forceinline__` 確保函數內聯

5.3 Double-Buffered Streams

- 實作兩個 stream 的 ping-pong pipeline 與 events
- Host 端 flag 檢查與新 kernel launch 重疊執行
- 這是超越單純同步 dispatch 的進階並行模式

```
// Alternating streams for pipeline
int slot_idx = batch_idx % PIPELINE_DEPTH;
PipelineSlot &slot = slots[slot_idx];

// Launch kernel in current stream
gpu_mine_kernel<<<num_blocks, threads_per_block, 0, slot.stream>>>(
    batch_start, batch_end, batch_stride,
    slot.d_result_nonce, slot.d_found_flag, slot.d_result_hash);

// Record completion event
CUDA_CHECK(cudaEventRecord(slot.completion_event, slot.stream));
```

Listing 7: Stream Pipeline 實作

5.4 Atomic Operation Optimization

- 在適當時機穿插 local flag 檢查與 atomic 操作
- Threads 只在必要時輪詢 global flag，而非每次迭代都檢查
- 使用 `atomicCAS` 實現 first-writer-wins 語義

5.5 Warp-Level Optimization

- 利用 warp 內 threads 的同步執行特性
- Constant memory 讀取自動 broadcast 至整個 warp
- 分支預測友善的控制流設計

5.6 Memory Coalescing

- 確保 global memory 存取對齊與合併
- Result buffer 寫入使用連續存取模式
- 最小化 memory transaction 數量

A 程式碼關鍵片段

A.1 Kernel 函數主體

```
__global__ void gpu_mine_kernel(unsigned int start_nonce,
                                unsigned int end_nonce,
                                unsigned int batch_stride,
                                unsigned int *result_nonce,
                                int *found_flag,
                                unsigned char *result_hash)
{
    unsigned int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int stride = blockDim.x * gridDim.x;
    unsigned long long nonce = static_cast<unsigned long long>(
       start_nonce) + thread_id;

    // 從 constant memory 載入 header
    unsigned int header_words[GPU_CONSTANT_HEADER_WORDS + 1];
    #pragma unroll
    for (int i = 0; i < GPU_CONSTANT_HEADER_WORDS; ++i) {
        header_words[i] = c_block_header_words[i];
    }

    // 批次處理 nonces
    while (nonce <= limit) {
        if (atomicAdd(found_flag, 0) != 0) break;

        for (unsigned int step = 0; step < batch_stride; ++step) {
            header_words[GPU_CONSTANT_HEADER_WORDS] = static_cast<
               unsigned int>(nonce);

            unsigned int hash_le[8];
            double_sha256_inline(header_words, hash_le);

            if (little_endian_bit_comparison(
                reinterpret_cast<unsigned char*>(hash_le), c_target,
                   32) < 0) {
                if (atomicCAS(found_flag, 0, 1) == 0) {
                    *result_nonce = static_cast<unsigned int>(nonce);
                    // 寫入 hash 結果
                }
                return;
            }
            nonce += stride;
        }
    }
}
```

Listing 8: GPU Mining Kernel

A.2 Double SHA-256 Inline 實作

```
__device__ __forceinline__ void double_sha256_inline(
    const unsigned int header_words[GPU_CONSTANT_HEADER_WORDS + 1],
    unsigned int (&hash_le)[8])
{
    unsigned int block[16];
    unsigned int state[8] = {
        0x6a09e667u, 0xbb67ae85u, 0x3c6ef372u, 0xa54ff53au,
        0x510e527fu, 0x9b05688cu, 0x1f83d9abu, 0x5be0cd19u
    };

    // 第一次 SHA-256: 前 64 bytes
    #pragma unroll
    for (int i = 0; i < 16; ++i) {
        block[i] = bswap32(header_words[i]);
    }
    sha256_process_block(block, state);

    // 第一次 SHA-256: 後 16 bytes + padding
    block[0] = bswap32(header_words[16]);
    block[1] = bswap32(header_words[17]);
    block[2] = bswap32(header_words[18]);
    block[3] = bswap32(header_words[GPU_CONSTANT_HEADER_WORDS]);
    block[4] = 0x80000000u;
    #pragma unroll
    for (int i = 5; i < 15; ++i) block[i] = 0u;
    block[15] = 80u * 8u;
    sha256_process_block(block, state);

    // 第二次 SHA-256
    unsigned int state2[8] = {
        0x6a09e667u, 0xbb67ae85u, 0x3c6ef372u, 0xa54ff53au,
        0x510e527fu, 0x9b05688cu, 0x1f83d9abu, 0x5be0cd19u
    };
    #pragma unroll
    for (int i = 0; i < 8; ++i) block[i] = state[i];
    block[8] = 0x80000000u;
    #pragma unroll
    for (int i = 9; i < 15; ++i) block[i] = 0u;
    block[15] = 32u * 8u;
    sha256_process_block(block, state2);

    // 轉換為 little-endian
    #pragma unroll
    for (int i = 0; i < 8; ++i) {
        hash_le[i] = bswap32(state2[i]);
    }
}
```

Listing 9: Double SHA-256 內聯函數