

# HW3 All-Pairs Shortest Path

---

HacMD link (<https://hackmd.io/@elliehung/Hk79vnhzJl>).

CS542200 Parallel Programming	HW3: All-Pairs Shortest Path	112065528	洪巧雲
-------------------------------	------------------------------	-----------	-----

## Hw3-1 CPU version

---

- 這次的作業所選擇實作的演算法是 Floyd-Warshall 演算法來解決All-Pairs Shortest Path 問題。
  - 利用 Blocked 策略，將整個距離矩陣切成大小為  $B \times B$  的區塊，分 3 個 Phase 處理。
- CPU 版本的優化主要是在 code 上使用 OpenMP 平行化與 SIMD 達成加速。

## Optimization

- OpenMP 平行化
  - 使用 `#pragma omp parallel for schedule(dynamic)`
- SIMD
  - 使用 `__m128i SIMD_l`、`__m128i SIMD_kj`、`__m128i SIMD_ik` 搭配 `_mm_add_epi32`、`_mm_min_epi32` 等進行批量運算，減少指令數量與提升 CPU pipeline 利用率。

```

// #pragma omp parallel for schedule(static)
#pragma omp parallel for schedule(dynamic)
for (int b_i = block_start_x; b_i < block_end_x; ++b_i) {
    for (int b_j = block_start_y; b_j < block_end_y; ++b_j) {
        int k_start = Round * B;
        int k_end = (Round + 1) * B;
        if (k_end > n) k_end = n;

        int block_internal_start_x = b_i * B;
        int block_internal_end_x = (b_i + 1) * B;
        int block_internal_start_y = b_j * B;
        int block_internal_end_y = (b_j + 1) * B;

        if (block_internal_end_x > n) block_internal_end_x = n;
        if (block_internal_end_y > n) block_internal_end_y = n;

        for (int k = k_start; k < k_end; ++k) {
            for (int i = block_internal_start_x; i < block_internal_end_x; ++i) {
                int ik = i * n + k;
                __m128i SIMD_ik = __mm_set1_epi32(Dist[ik]);

                // 使用 SSE 處理 j 的迴圈，每次處理4個元素
                int j;
                for (j = block_internal_start_y; j + 3 < block_internal_end_y; j += 4) {
                    // 載入 Dist[i][j..j+3]
                    __m128i SIMD_l = __mm_loadu_si128((__m128i*)&Dist[i*n + j]);
                    // 載入 Dist[k][j..j+3]
                    __m128i SIMD_kj = __mm_loadu_si128((__m128i*)&Dist[k*n + j]);
                    // SIMD_ik + SIMD_kj
                    __m128i SIMD_r = __mm_add_epi32(SIMD_ik, SIMD_kj);
                    // 取 min
                    __m128i SIMD_min = __mm_min_epi32(SIMD_l, SIMD_r);
                    // 將結果寫回
                    __mm_storeu_si128((__m128i*)&Dist[i*n + j], SIMD_min);
                }

                // 處理不滿4個的殘餘部分
                for (; j < block_internal_end_y; j++) {
                    int ij = i*n + j;
                    int kj = k*n + j;
                    int new_val = Dist[ik] + Dist[kj];
                    if (new_val < Dist[ij]) {
                        Dist[ij] = new_val;
                    }
                }
            }
        }
    }
}

```

## Hw3-2 Single-GPU version

### 如何分配 CUDA kernel 的資料大小

- thread block: (32,32) · 代表的是每個 block 中的 x 方向有 32 個 threads · y 方向有 32 個 threads。總共會有 1024 條 threads。
- fw\_phase1:

```
// Phase 1: Process pivot block
fw_phase1<<<dim3(1, 1), dim3(32, 32)>>>(device_matrix, padded_n, r);
```

(1, 1) 代表只使用一個 thread block

- fw\_Phase2:

```
// Phase 2: Process pivot row and column
fw_phase2<<<dim3(numBlocks, 2), dim3(32, 32)>>>(device_matrix, padded_n, r, numBlocks);
```

dim3(numBlocks, 2) : 用來定義 fw\_phase2 階段 Kernel 所需要的 thread blocks 數量  
 , 這裡設置為 numBlocks \* 2 。

x 軸 = numBlocks : blockIdx.x -> 第幾個 Block 。

y 軸 = 2 : 代表 pivot block 的 x 軸以及 pivot block 的 y 軸 , if (blockIdx.y == 0)  
 則是在處理 pivot row , if (blockIdx.y == 1) 則是在處理 pivot column 。

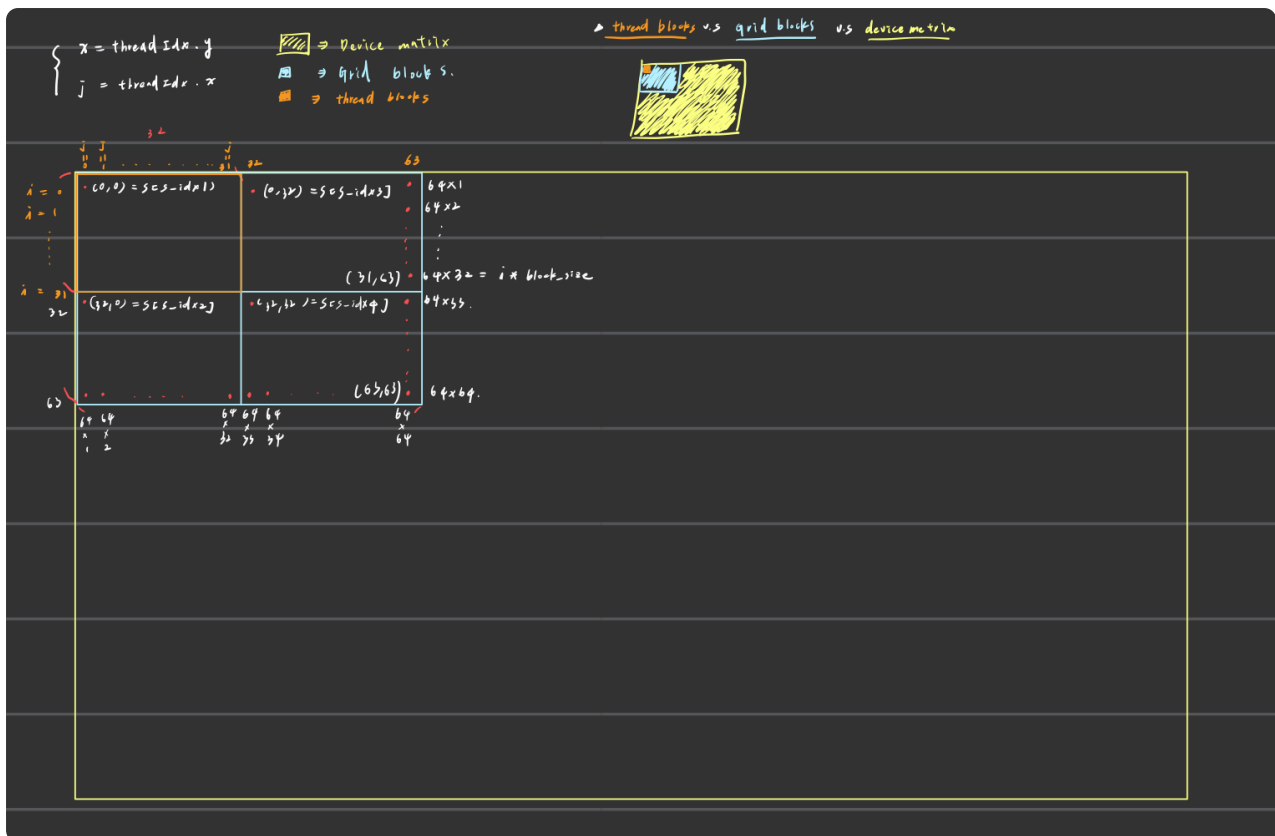
- fw\_phase3:

```
// Phase 3: Process remaining blocks
fw_phase3<<<dim3(numBlocks, numBlocks), dim3(32, 32)>>>(device_matrix, padded_n, r, numBlocks);
```

dim3(numBlocks, numBlocks) : 用來定義 fw\_phase3 階段 Kernel 所需要的 thread  
 blocks 數量 , 這裡設置為 (numBlocks) \* (numBlocks)

◦ CUDA 就是透過以上方法去 activate kernel 。

- 以下為圖示:



$\triangle$  thread blocks v.s grid blocks v.s device matrix



由圖中可以看到黃色的部分是 Device Matrix

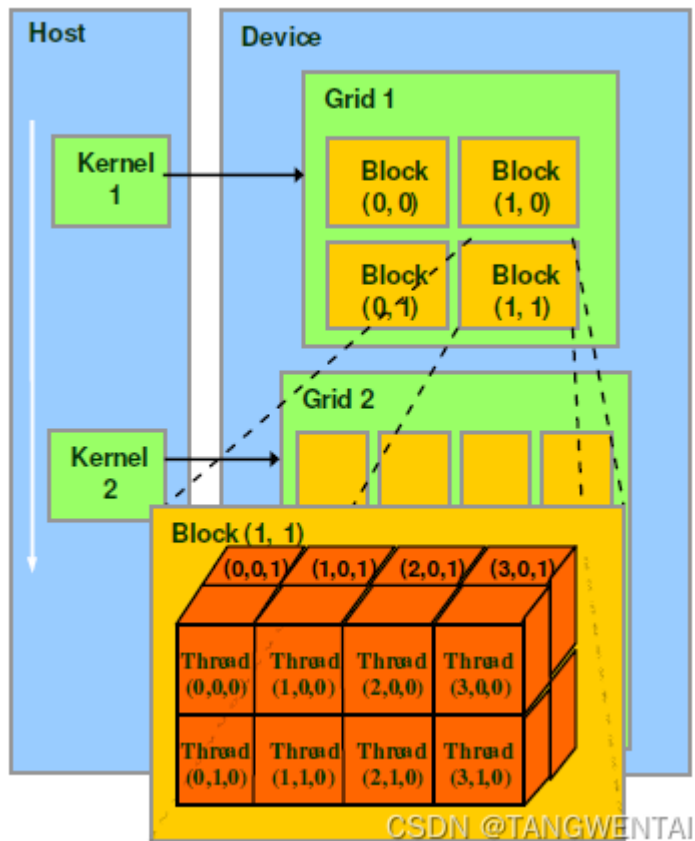
· 也就是一開始存 vertex、edges 的大矩陣。size 為  $64 \times 64$  的藍色矩形則是 Grid block:  $\text{block\_size} = 64$ ，這個大小會跟每個 Kernel 中所宣告的 shared memory 大小相同，因為我們會需要以 Grid Block 為單位去實作 Floyd-Warshall 的演算法：

```
__global__ void fw_phase1(int *dist, int padded_n, int r)
{
    __shared__ int s[64][64];
```

最後是橘色最小的矩形，這部分則是 thread block，其大小為  $32 \times 32$ 。

目前定義了三個不同的矩形以及它們各自的 size，接下來會解釋它們各自的關係與用意。

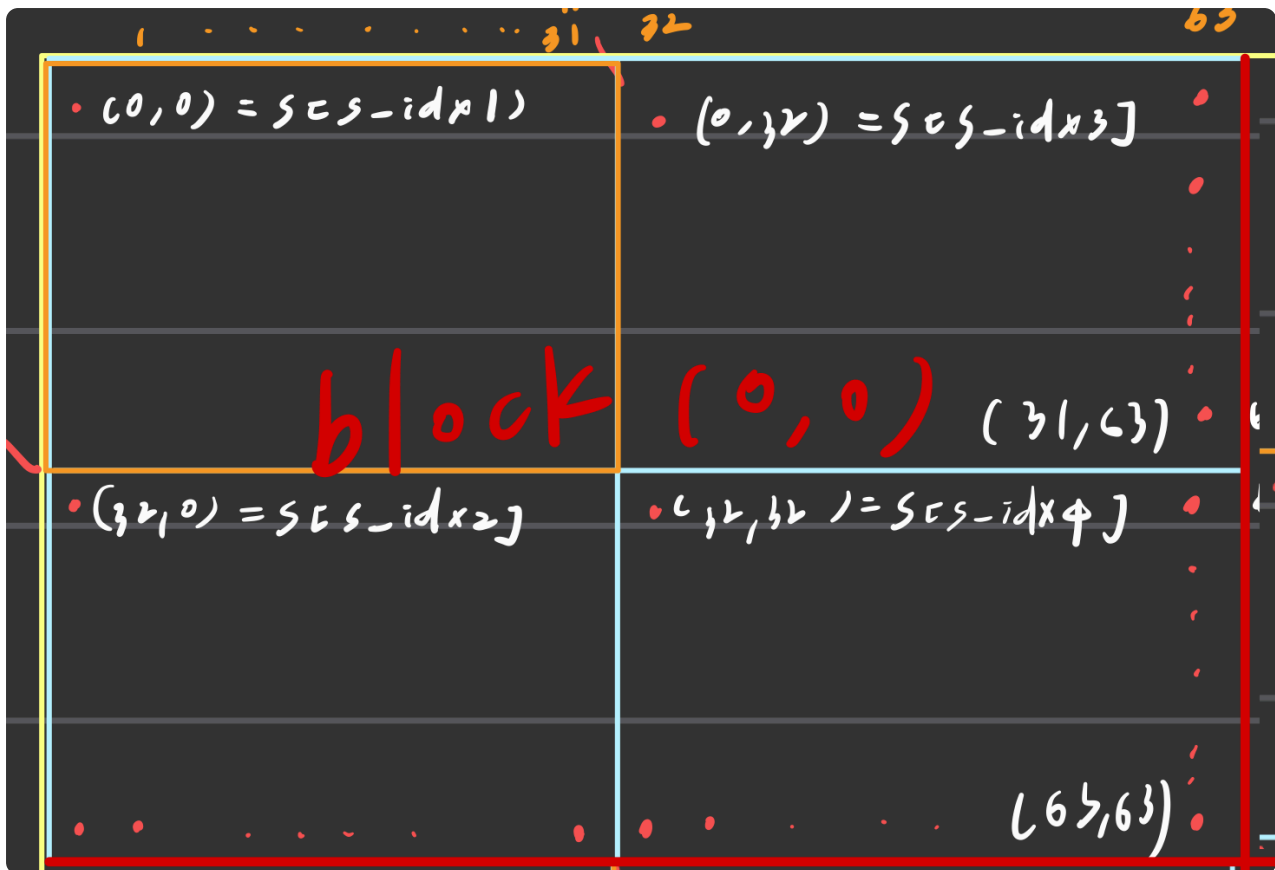
下圖解釋了 Thread Block 與 Grid 之間的關係:



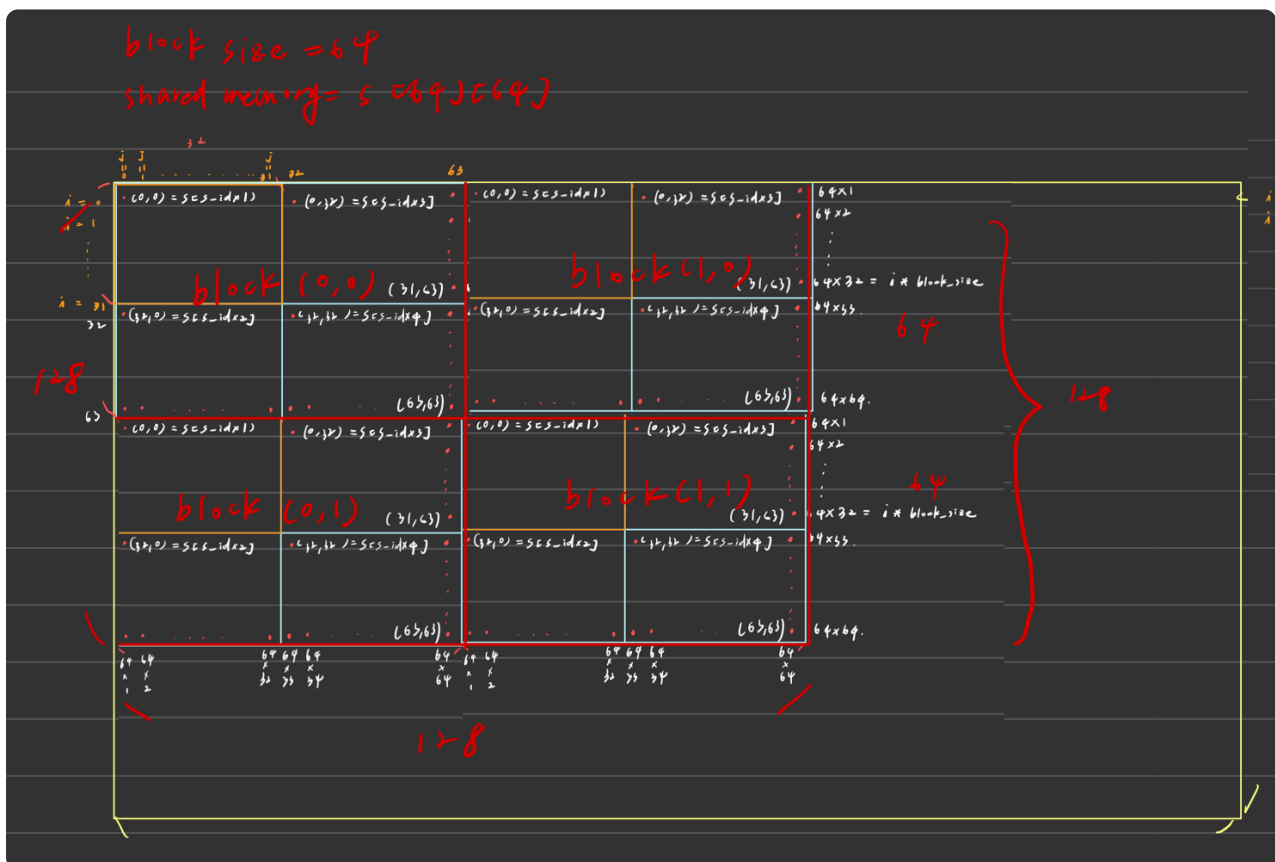
這張圖片則是展示了 Host 跟 Device 之間 Kernel 與 Grid、Block、Thread 之間的關係，可以清楚的看到 Device 的部分有 Grid，而每個 Grid 中存放了四個 Block，每個 Block 中則是開了多個 Threads。在我原本設定的程式中，一開始我是使用 `block_size = 32`，這時 Threads 一樣是開 `32 x 32`，但是在 Kernel 中，程式會長的跟 `block_size = 64` 不一樣，原因是 `block_size = 32` 的話，它跟 Thread Block 大小會相同，代表說一條 thread 一個 thread 處理 1 筆資料，如果要達到啟動 Kernel 後可以一個 thread 處理多筆資料的話，則可以把 Block Size 開大一點，像是開到 64 的話，它可以一次處理 4 筆資料，也就是如下所示一次更新 Block 中的四個 index 元素：

```
#pragma unroll 64
for (int k = 0; k < 64; ++k) {
    idx_1 = min(idx_1, s_row[i][k] + s_col[k][j]);
    idx_2 = min(idx_2, s_row[I_32][k] + s_col[k][j]);
    idx_3 = min(idx_3, s_row[i][k] + s_col[k][J_32]);
    idx_4 = min(idx_4, s_row[I_32][k] + s_col[k][J_32]);
}
```

也就是以下這張示意圖中的：(0, 0)、(0, 32)、(32, 0)、(32, 32) 這四個位置的 index:



下圖的  $\text{block}(0,0)$ 、 $\text{block}(1,0)$ 、 $\text{block}(0,1)$ 、 $\text{block}(1,1)$ ：



分別代表了下圖的 grid 中的 4 個 Block:

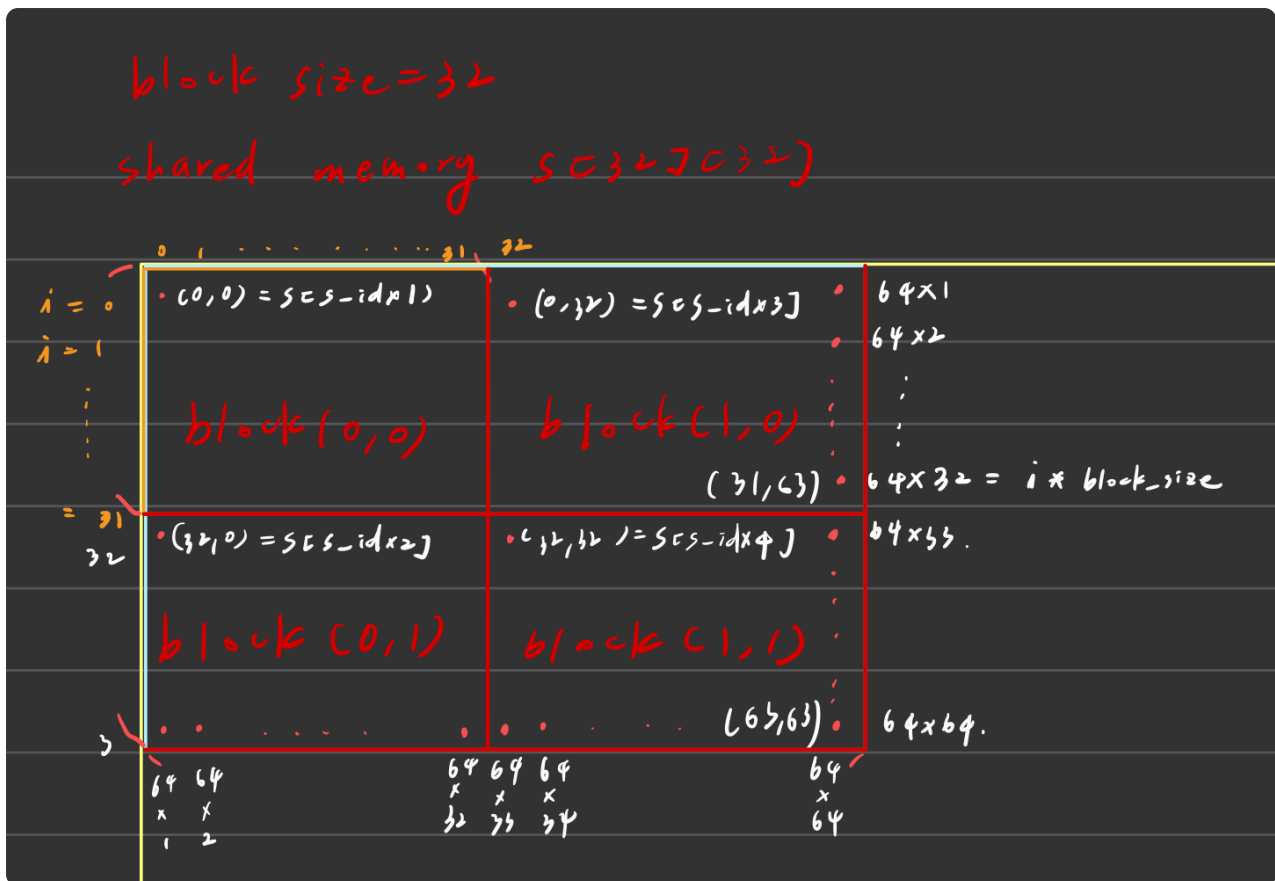


接下來是每個 Block 中會有 Thread Blocks，也對應到圖中的每個 Block 中的 Thread:

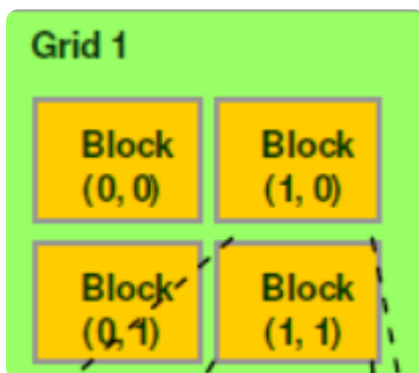


如果 Block Size 是開 32 的話，程式碼如下，就會是更新一個元素:

```
// Compute shortest paths for pivot row and column
#pragma unroll 32
for (int k = 0; k < BLOCK_SIZE; ++k) {
    if (b_j == 0) {
        // Column-wise update
        int updated_value = s_block[ty][k] + s_pivot[k][tx];
        s_block[ty][tx] = min(s_block[ty][tx], updated_value);
    } else {
        // Row-wise update
        int updated_value = s_pivot[ty][k] + s_block[k][tx];
        s_block[ty][tx] = min(s_block[ty][tx], updated_value);
    }
    __syncthreads();
}
```



上圖一樣對應到下圖：



## Block Size 的選擇

Block Size	每條 Thread 的工作量
32	一個 thread 處理 1 筆資料 $(32 \times 32) / (32 \times 32) = 1$
64	一個 thread 處理 4 筆資料 $(64 \times 64) / (32 \times 32) = 4$

之所以 Thread Block 只開  $32 \times 32$  原因是 CUDA 最多只能開 1024 條 threads，雖然 Block size 可以開到 128、256、512...，但是我們的 GPU (1080) 的 memory 只有 48 KB，所以最大的 block size 只能開到 110，但是要跟 thread align 故只能開到最接近 110 的 32 倍數，又因為 Phase2 & Phase3 需要 2 個 shared memory ( $2 \times 64 \times 64 = 8,192$ )



$2 * 4096 * 4 \text{ byte} = 32,768 \rightarrow 32 \text{ KB}$

假設 block size 開到 96，在 Phase2 & Phase3 所需要的記憶體如下：

$2 * 96 * 96 * 4 \text{ byte} = 73,728 \rightarrow 73 \text{ KB}$

這樣就超過硬體的限制了，故這份作業 Block Size 最多只能開到 64。

## Grid Blocks 的數量選擇

首先是 Grid Block Size 我選擇了 64，那就需要去計算出最接近 Input vertex number  $n$  的 64 的倍數去宣告成 `host_matrix` 與 `device_matrix` 的大小：

- `host_matrix`：

```
padded_n = ((n + 64 - 1) / 64) * 64;
size_t size = padded_n * padded_n * sizeof(int);
*host_matrix = (int *)malloc(size);
```

- `device_matrix`：

```
// Allocate device memory
int *device_matrix;
cudaMalloc(&device_matrix, padded_n * padded_n * sizeof(int));
```

接著是計算總共會切幾個 Grid Blocks：

```
int numBlocks = padded_n / 64;
```

這個 `numBlocks` 也是總共 Phase1、Phase2、Phase3 需要做幾次才能計算完成的回合數。

## Hw3-3 Multi-GPU version

- hw3-3 中使用了 OpenMP 開兩條 threads，分別分配給兩個 GPU

```
#pragma omp parallel num_threads(2)
```

```
cudaSetDevice(threadId)
```

根據不同 thread 去設定所要使用的 GPU: `threadId=0` 使用 GPU0，`threadId=1` 使用 GPU1。

- Hw3-3 的 block size 與 Hw3-2 一樣使用的是 64，然後在 Phase3 時才會分多個 GPU 去做平行化，其中會需要計算幾個不同的變數：

gridPhase3Size：這是指在 Phase3 負責處理的 Blocks 數量：

```
int gridPhase3Size = halfBlocks + (threadId == 1 ? extraBlocks : 0);
```

start：為此 GPU 的起始處理的 index。

```
int start = halfBlocks * threadId;
```

接下來與 hw3-2 不同的地方在於，前面提到在 Phase3 會需要改成多 GPU 執行運算，所以在 `__global__ void fw_phase3(int *dist, int padded_n, int r, int numBlocks, int start)` 的 function 中會需要增加一個傳入的參數: `start`，這個參數是為了讓不同 GPU 負責 Phase3 的計算處理的區塊不會重疊，都位於其負責的範圍內。

- 分配資料的方法為將 data 分一半，前半分給 GPU0 後半分給 GPU1，若有餘數則是分給 GPU1。在 `fw_phase1`、`fw_phase2` 這兩個計算的階段，Phase2 會需要 Phase1 的資料才能做計算，但是 Phase3 是去計算非 Pivot Block、Pivot Row、Pivot Column 的部分，它整個計算都是獨立的，不會涉及到後續有 data dependency 的問題，故在 Phase3 做 Multi-GPU 的分配非常適合。

```
// 計算每個線程負責的起始區塊和區塊數量
int halfBlocks = numBlocks / 2;
int extraBlocks = numBlocks % 2;

// 計算起始區塊
int start = halfBlocks * threadId;

// 計算負責處理的區塊數量
int gridPhase3Size = halfBlocks + (threadId == 1 ? extraBlocks : 0);
```

## Communication between Multi-GPU

- Host 與 Device 之間的數據傳輸
  - Step 1 初始化複製資料: 在 OpenMP thread 各自分配好自己的 GPU 後 `cudaSetDevice(threadId);`，會需要將各自負責處理的資料區塊從 `host_matrix` 複製到對應的設備記憶體 `device_matrix`：

```
cudaMemcpy(device_matrix[threadId] + start * BLOCK_SIZE * padded_n,
            host_matrix + start * BLOCK_SIZE * padded_n,
            gridPhase3Size * BLOCK_SIZE * padded_n * sizeof(int),
            cudaMemcpyHostToDevice);
```

- Step 2 GPU 之間的 Data 交換: 在每個計算的階段中，包含了 `fw_phase1`、`fw_phase2` 以及 `fw_phase3`，因為上面已經分好原始 data 會切

一半，一半給 GPU0 一半給 GPU1，所以說會需要做資料的交換，才能確保計算的 result 是正確的。這裡使用的一樣是 `cudaMemcpy`：

下圖是在判斷是否為這個 GPU 負責到的 data 範圍，是的話會將記憶體由 device 複製到 host:

```
if (blockId >= start && blockId < start + gridPhase3Size) {
    cudaMemcpy(host_matrix + blockId * BLOCK_SIZE * padded_n,
               device_matrix[threadId] + blockId * BLOCK_SIZE * padded_n,
               BLOCK_SIZE * padded_n * sizeof(int),
               cudaMemcpyDeviceToHost);
}
```

這部分的資料是為了讓上一個 round 所計算完的 Phase1、Phase2、Phase3 階段的結果更新到 Host 上，才能後續計算出正確的結果。

下圖則是在判斷是否不為這個 GPU 負責到的 data 範圍，這樣的話會將記憶體會由 host 複製到 device:

```
if (blockId < start || blockId >= start + gridPhase3Size) {
    cudaMemcpy(device_matrix[threadId] + blockId * BLOCK_SIZE * padded_n,
               host_matrix + blockId * BLOCK_SIZE * padded_n,
               BLOCK_SIZE * padded_n * sizeof(int),
               cudaMemcpyHostToDevice);
}
```

這部分是為了讓這一回合進行計算前把其他 GPU 所產生的計算結果更新到自己的 device 上 (from host)，所以才需要 Memcpy 到 Device。

在上述 data 的傳輸，GPU 與 GPU 之間直接的溝通是透過 Host 做溝通。

- Step 3 傳回最終 result:

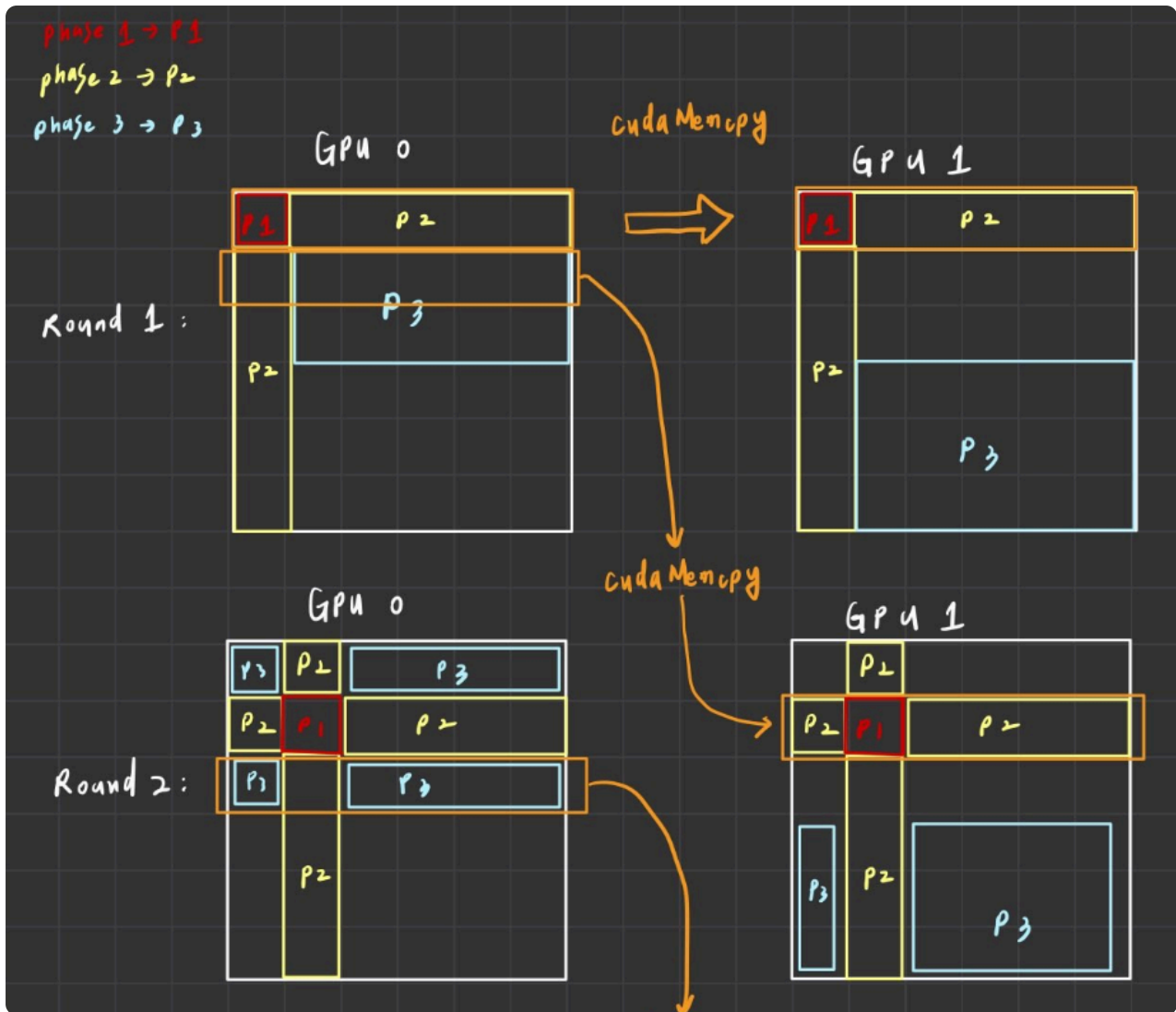
```
cudaMemcpy(host_matrix + start * BLOCK_SIZE * padded_n,
           device_matrix[threadId] + start * BLOCK_SIZE * padded_n,
           gridPhase3Size * BLOCK_SIZE * padded_n * sizeof(int),
           cudaMemcpyDeviceToHost);
```

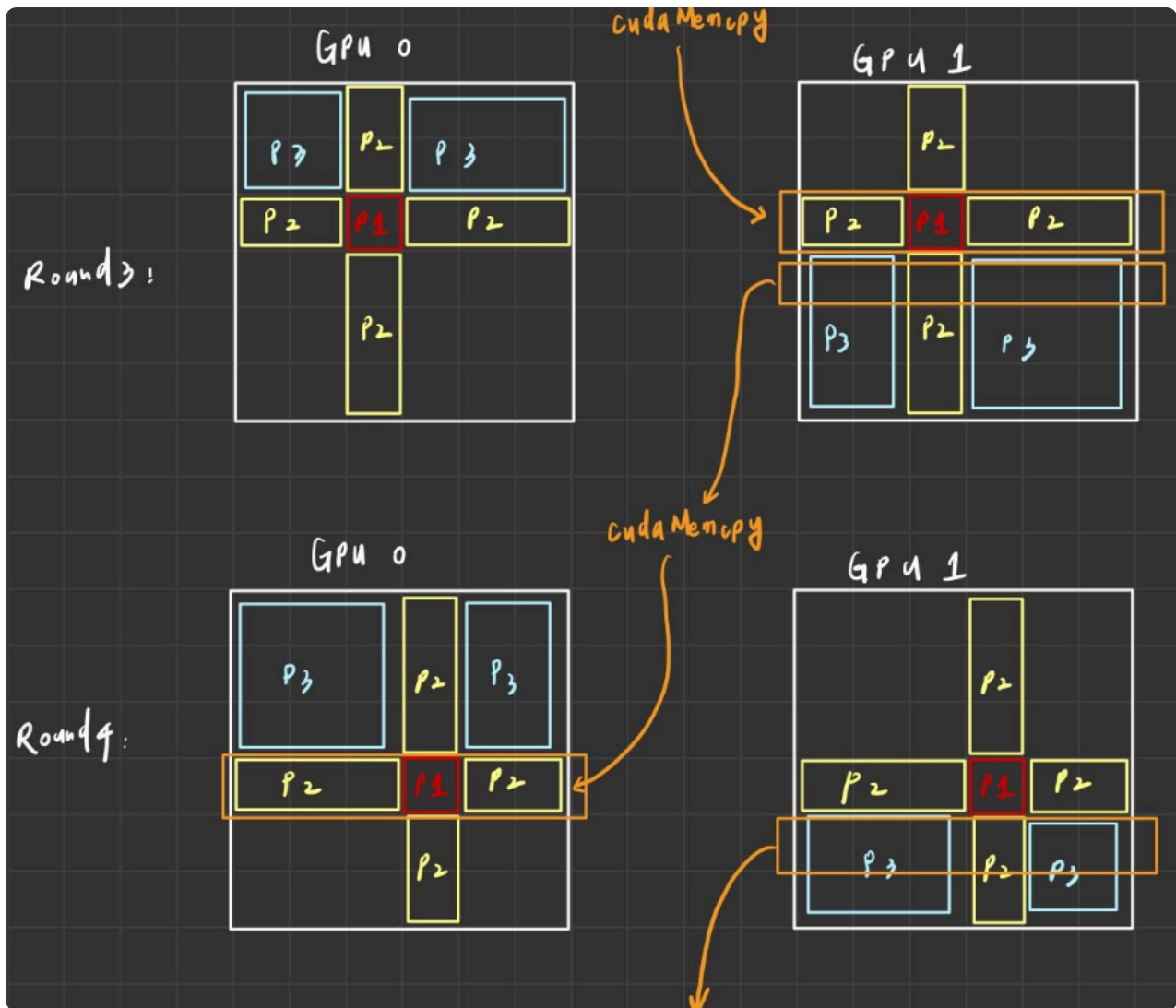
- Threads 之間的 Synchronization:

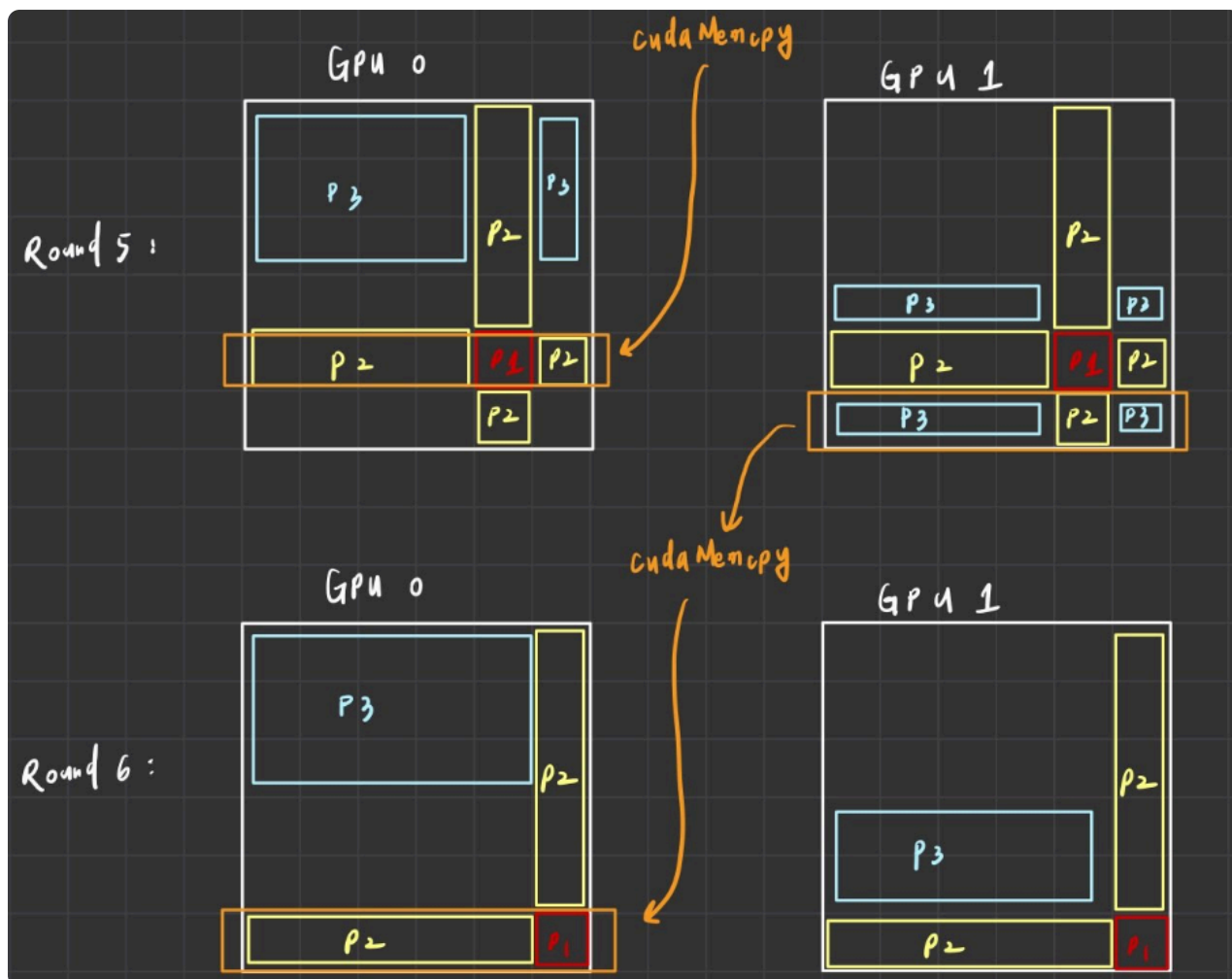
- 使用了 `#pragma omp barrier` 去做所有 threads 的同步，這會確保所有 threads 在同一步驟完成後才進入下一個步驟。
- 確保所有 OpenMP 線程在此點完成資料從設備到主機的複製後，才繼續進行下一步操作。

## Implementation in figures

以下使用圖解解釋我的 code 是如何透過 cudaMemcpy 做資料的更新:







可以看到在 Round1 ~ Round3 中，因為主要在 Phase1 & Phase2 更新的資料位於 Block 的上半部，故主要做 cudaMemcpy 的方向為 GPU0 -> GPU1，在 Round4 ~ Round6 的部分 Pivot 換到下半部了，也就是 GPU1 在負責的區域，故主要做 cudaMemcpy 的方向換成 GPU1 -> GPU0。這也是為甚麼在進入任何一個 Phase 之前會需要先 call cudaMemcpy 將資料先做更新。

## Profiling Results (hw3-2)

### Blocking Factor Result

Testcases: c20.1

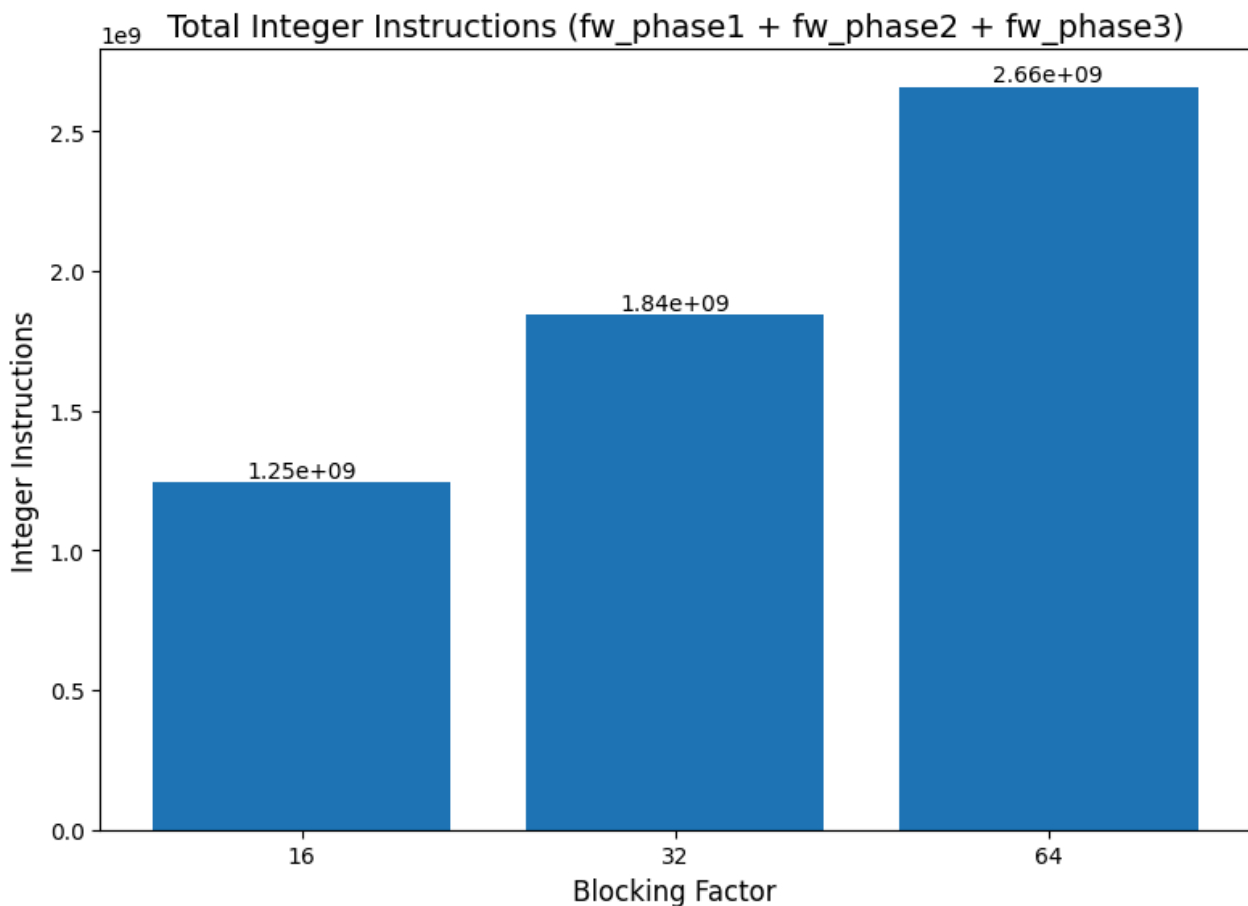
在 Blocking Factor 的實驗中，我使用的測資為 c20.1，並且使用 `srun -p nvidia -N1 -n1 --gres=gpu:1 nvprof -m $NVPROF_METRIC ./hw3-2-$i $IN $OUT` 其中：

`NVPROF_METRIC=inst_integer,shared_load_throughput,shared_store_throughput,gld_throughput,gst_throughput` 以及 `for i in {16,32,64}`。下圖為跑出來的結果：

Invocations	Metric Name	Metric Description	Min	Max	Avg
Device "NVIDIA GeForce GTX 1080 (0)"					
Kernel: fw_phase3(int*, int, int, int)					
79	inst_integer	Integer Instructions	2660538368	2660538368	2660538368
79	shared_load_throughput	Shared Memory Load Throughput	3107.2GB/s	3174.1GB/s	3145.8GB/s
79	shared_store_throughput	Shared Memory Store Throughput	129.47GB/s	132.25GB/s	131.08GB/s
79	gld_throughput	Global Load Throughput	194.20GB/s	198.38GB/s	196.62GB/s
79	gst_throughput	Global Store Throughput	64.734GB/s	66.127GB/s	65.538GB/s
Kernel: fw_phase2(int*, int, int, int)					
79	inst_integer	Integer Instructions	68212736	68212736	68212736
79	shared_load_throughput	Shared Memory Load Throughput	2626.0GB/s	2884.1GB/s	2725.1GB/s
79	shared_store_throughput	Shared Memory Store Throughput	109.42GB/s	120.17GB/s	113.54GB/s
79	gld_throughput	Global Load Throughput	164.13GB/s	180.26GB/s	170.32GB/s
79	gst_throughput	Global Store Throughput	54.709GB/s	60.086GB/s	56.772GB/s
Kernel: fw_phase1(int*, int, int)					
79	inst_integer	Integer Instructions	541696	541696	541696
79	shared_load_throughput	Shared Memory Load Throughput	122.06GB/s	125.38GB/s	123.49GB/s
79	shared_store_throughput	Shared Memory Store Throughput	41.107GB/s	42.227GB/s	41.591GB/s
79	gld_throughput	Global Load Throughput	647.59MB/s	665.23MB/s	655.22MB/s
79	gst_throughput	Global Store Throughput	647.59MB/s	665.23MB/s	655.22MB/s

- Total Integer Instructions:

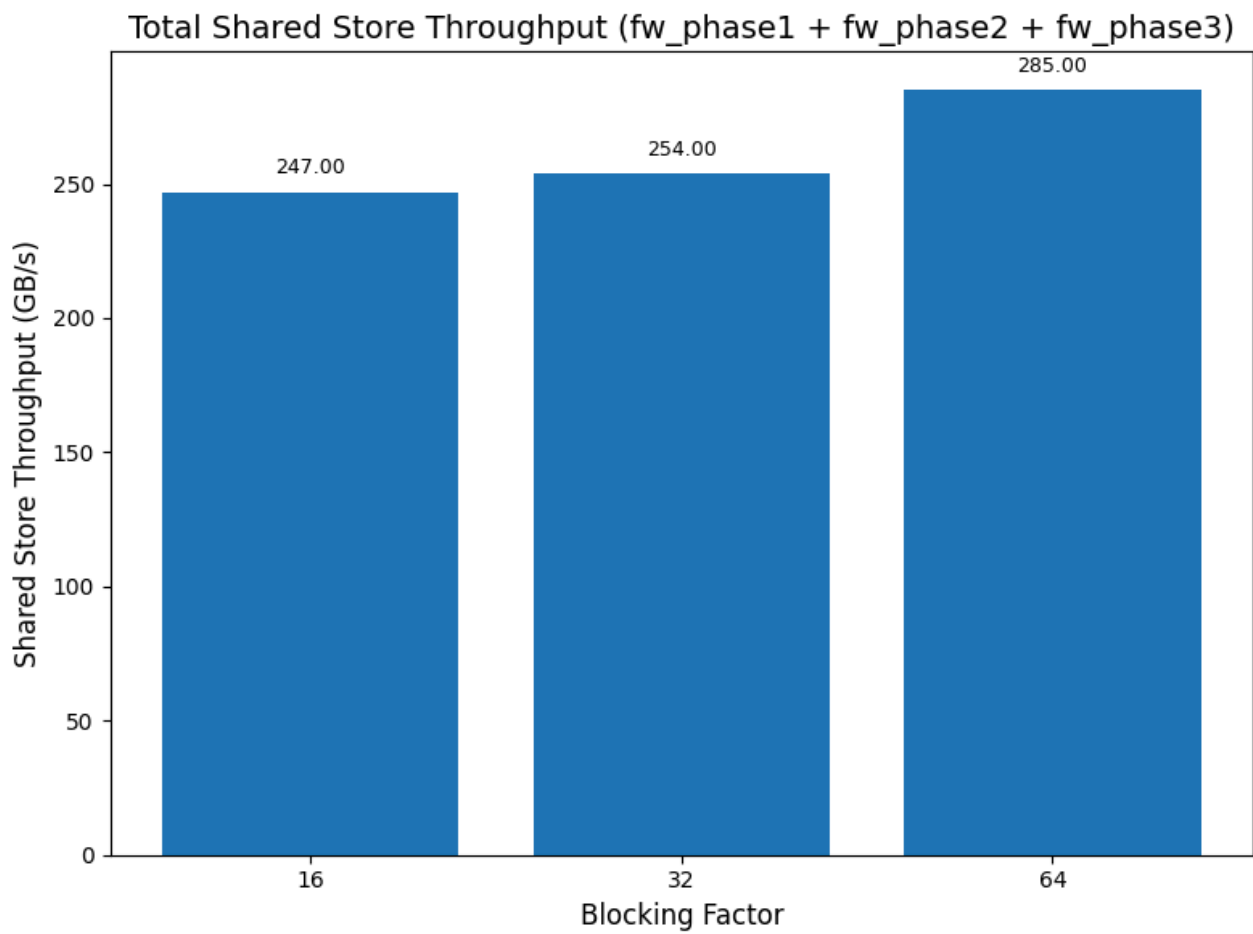
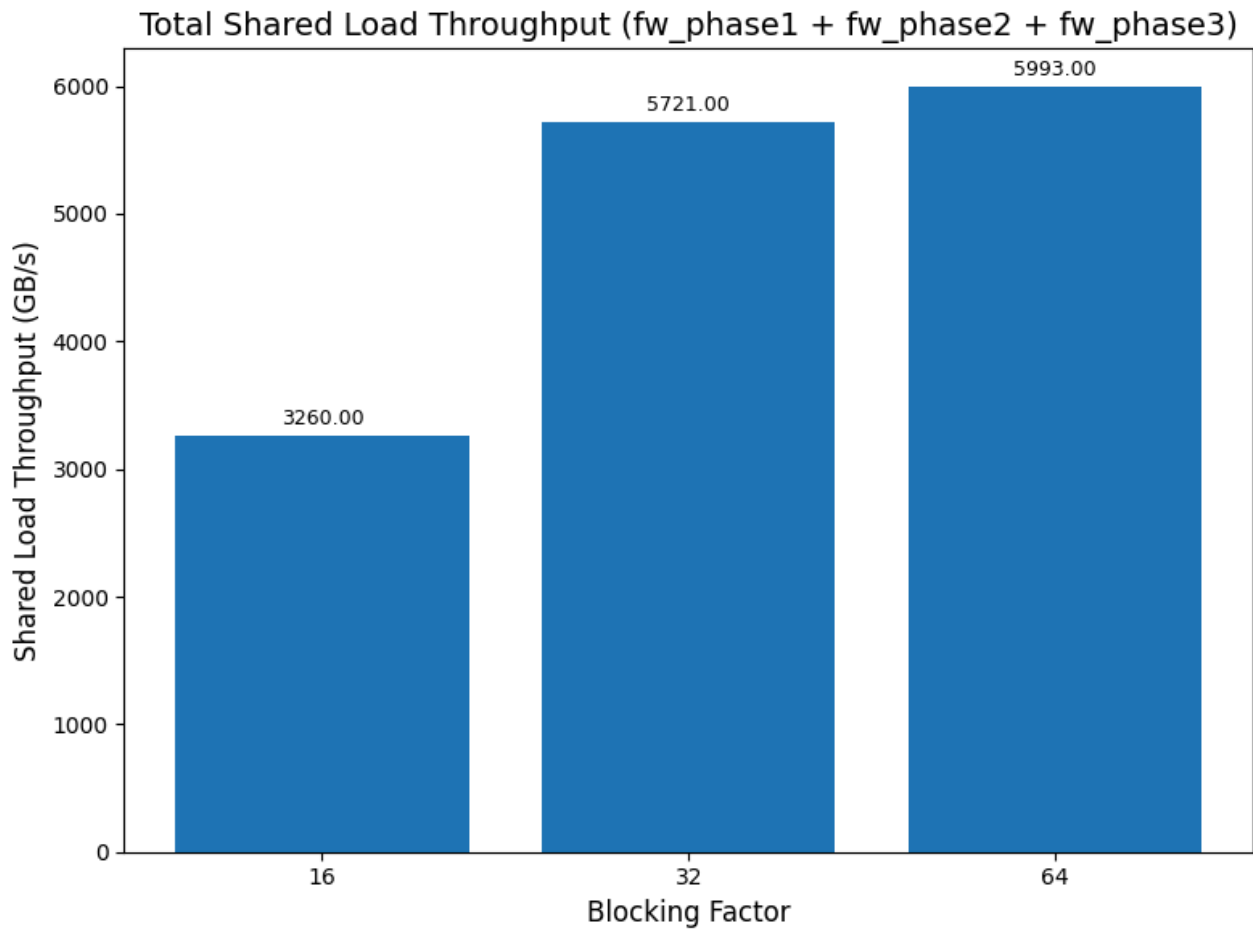
整數運算指令的執行次數隨著 Block Size 越大，執行次數越多，會出現這樣的實驗結果是因為在 block size 變大後，每個 block 所需要處理的指令變多，這就會導致整數運算指令會隨之增加。增加 Block Size 會使得總通信量減少，但需要付出的代價就是會造成每個 Block 的內部計算變大。



- Shared Memory Load/Store:

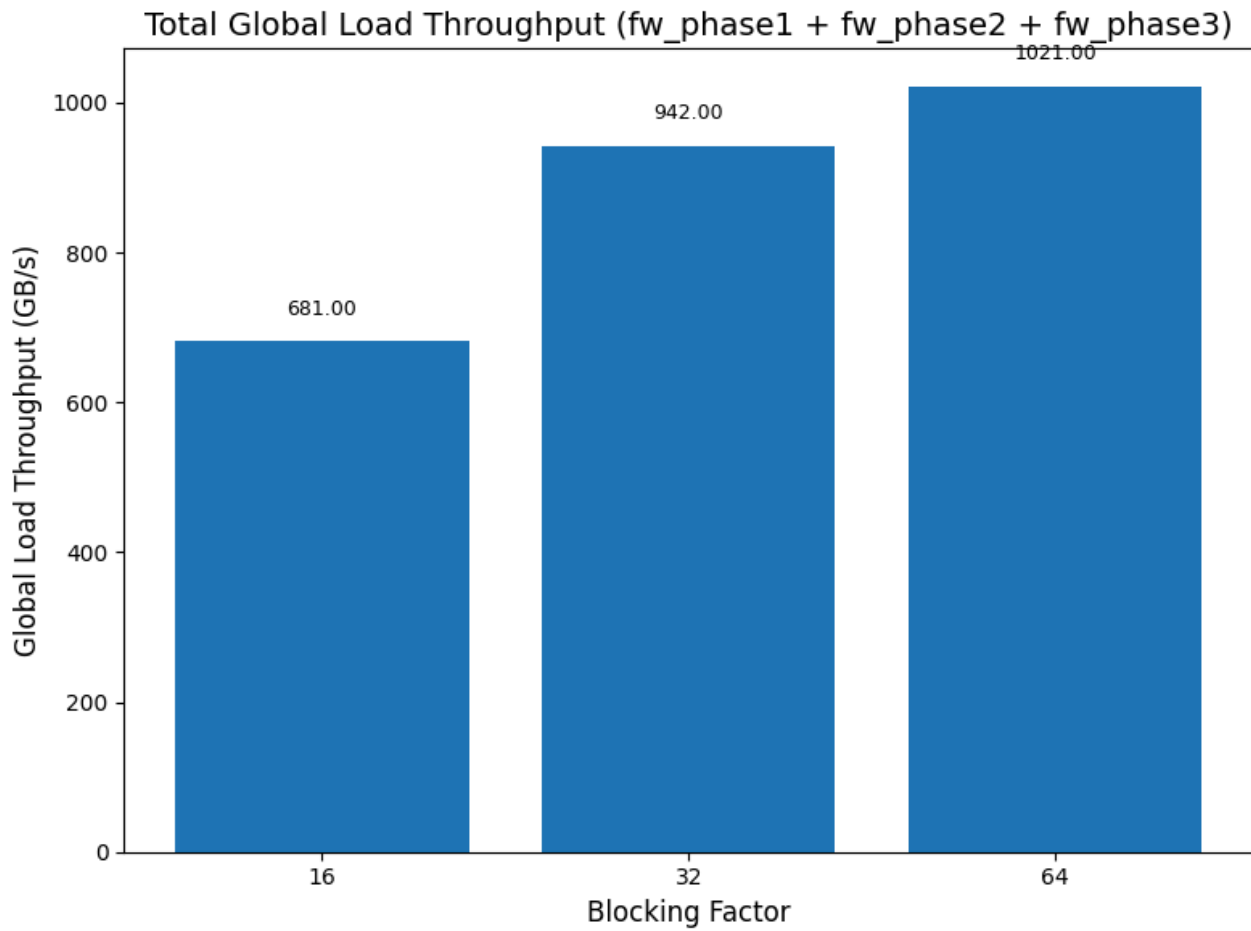
下面兩張圖展現了共享記憶體の Throughput 會隨著 Block Size 的增加也會隨之增加。原因是 Block 越大，在單一 Block 中所需要處理的數據量也會變多，Throughput 越大也代表了程式碼充分利用了 GPU 的資源，這會間接導致執行時間的優化。

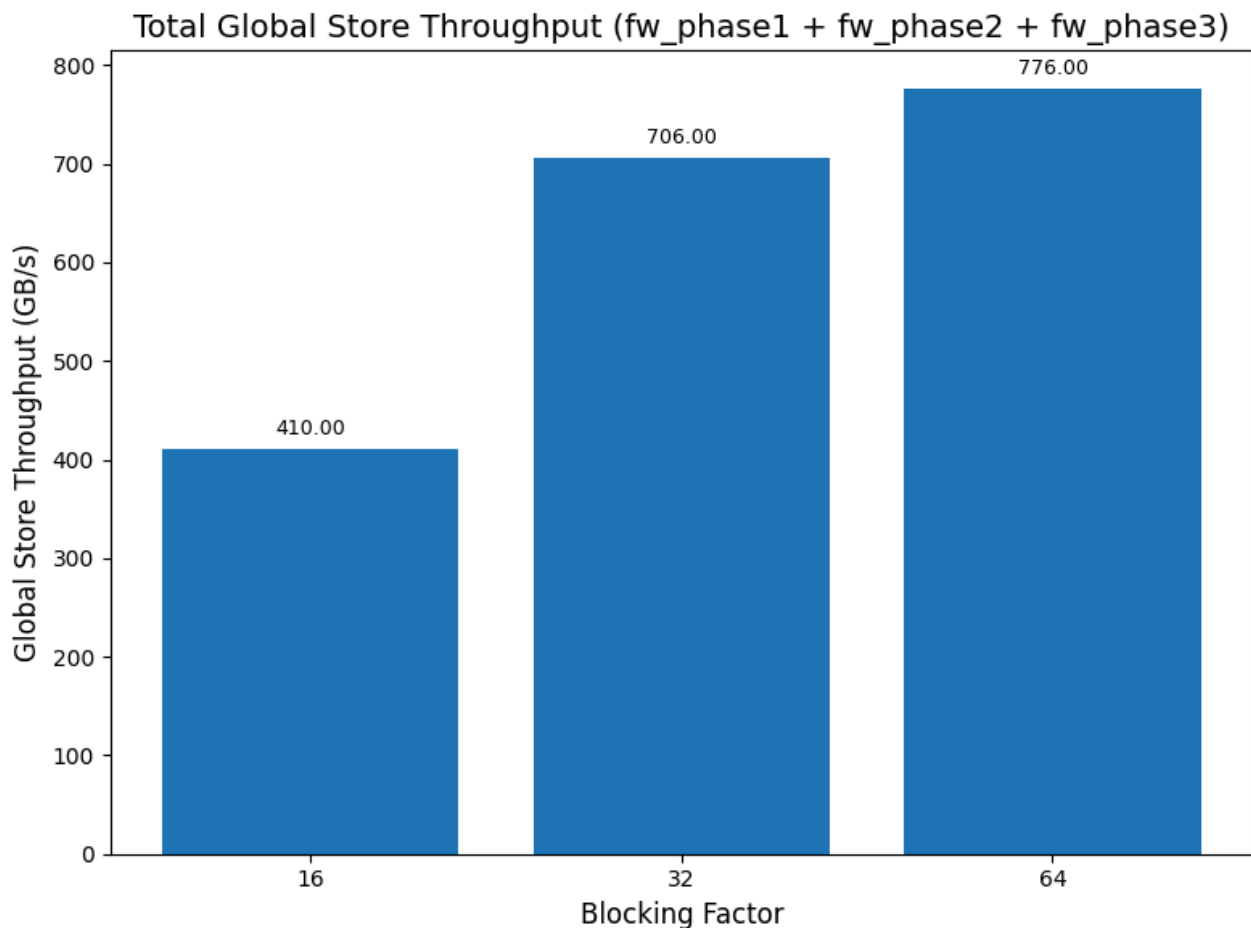




- Global Memory Load/Store:

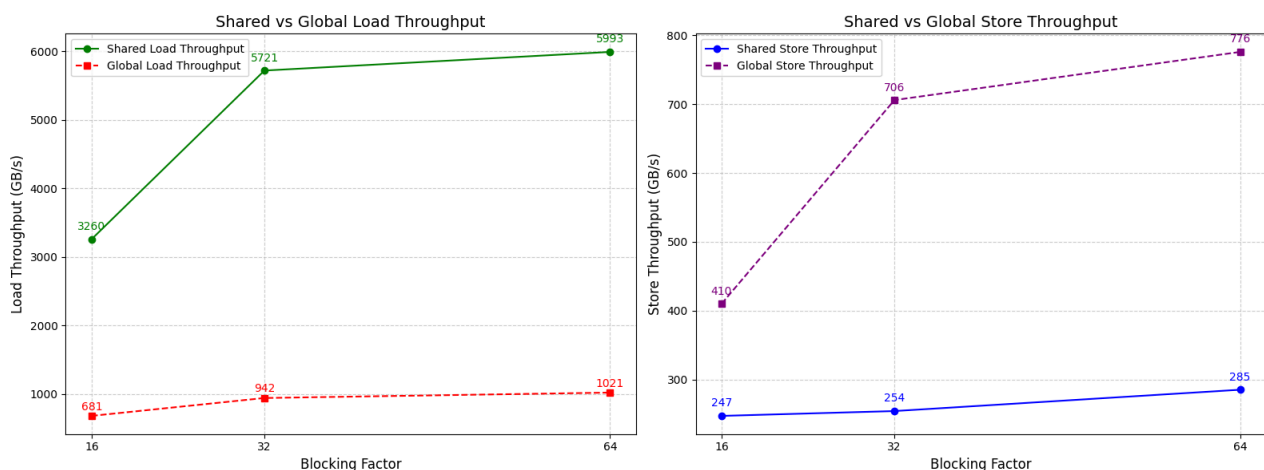
Global Memory 跟 Shared Memory 的狀況差不多，隨著 Blocking Size 的成長，Throughput 也會隨之增加。





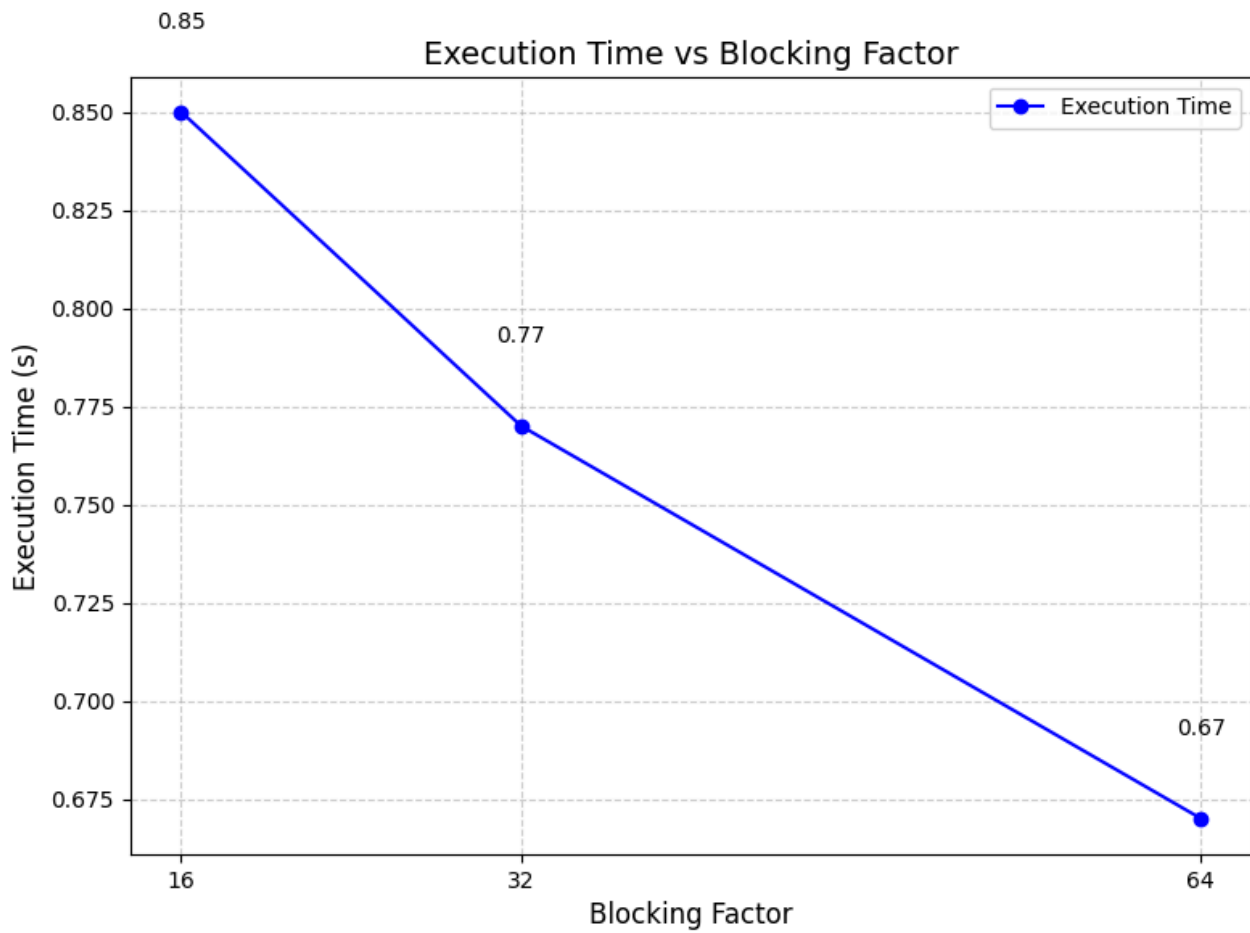
- Shared Memory and Global Memory Comparison:

在把所有數據綜合比較後，以下圖表展現出無論是 Load 還是 Store 操作，Shared memory 的 Throughput 都高於 Global memory 的 Throughput，原因是 Shared memory 相較於 Global memory 更適合用於頻繁讀寫的情況。



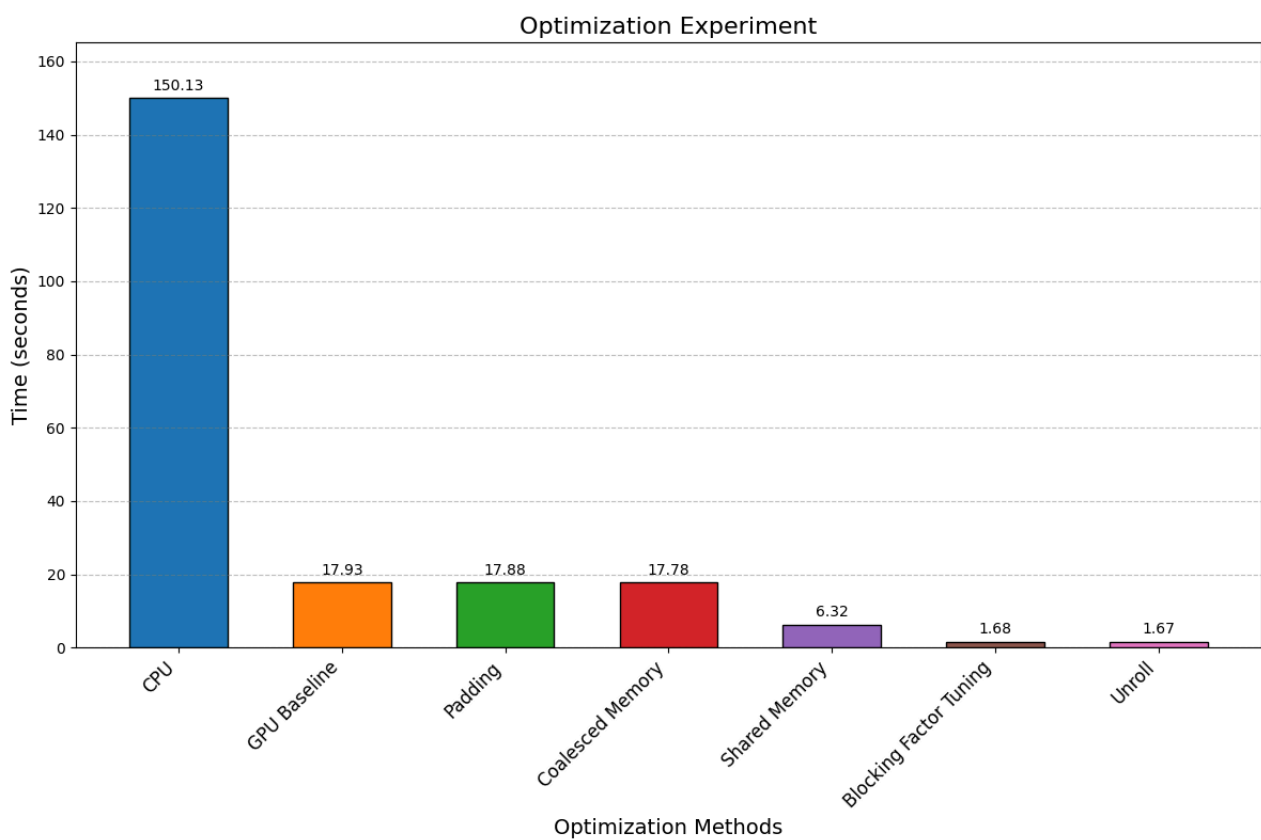
- Total execution time:

最後是整體 Blocking Factor 與執行時間的折線圖，可以發現在開到 64 時執行時間比 16 還要少非常多。



## Optimization

Testcases: p11k1



- Padding

- 將 `padded_n` 取到最接近 64 的整數倍，然後在初始化矩陣時，將超過原本  $n$  的部分都設成 `INF`。

```
int remainder = n % BLOCK_SIZE;
if(remainder != 0){
    n += (BLOCK_SIZE - remainder);
}
```

- 這樣的優化可以讓程式在執行 kernel function 的時候，讓每個 block 都可以處理相同大小的區域，避免需要使用 branch 判斷。
- 使 block/thread 對齊。

- Coalesced Memory

- 讓同一個 warp (32 threads) 可以同時存取「連續的 address」，提升 Global Memory bandwidth 的使用效率。
- 這樣做還可以減少 bank conflict。

```
int global_i = by * BLOCK_SIZE + i;
int global_j = bx * BLOCK_SIZE + j;

for (int k = 0; k < BLOCK_SIZE; k++) {
    __syncthreads();
    int old_val = dist[global_i * padded_n + global_j];
    int new_val = dist[global_i * padded_n + (B + k)] + dist[(B + k)*padded_n + global_j];
    if (new_val < old_val) {
        dist[global_i * padded_n + global_j] = new_val;
    }
}
```

如上圖所示，我實作 coalesced memory 的方式為透過計算 `global_i` 與 `global_j` 這兩個變數，確保 `threads` 中同時存取連續的記憶體位址。

```
__global__ void fw_phase3(int *dist, int padded_n, int r, int numBlocks)
{
    int bx = blockIdx.x;
    int by = blockIdx.y;
    if (bx == r || by == r) return; // 不處理 pivot row/column 所在的 block

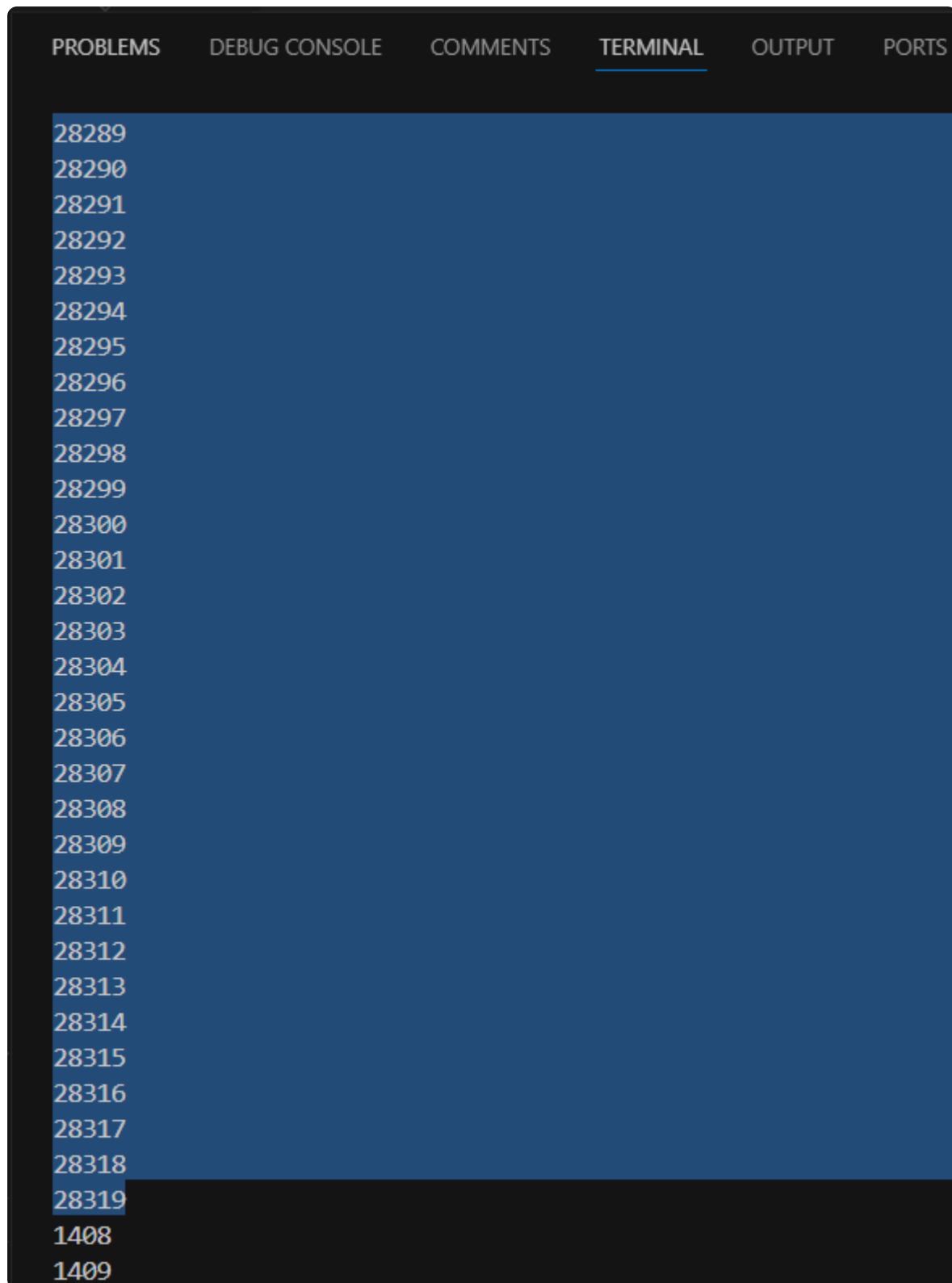
    int i = threadIdx.y;
    int j = threadIdx.x;
    int B = r * BLOCK_SIZE;

    int global_i = by * BLOCK_SIZE + i;
    int global_j = bx * BLOCK_SIZE + j;

    for (int k = 0; k < BLOCK_SIZE; k++) {
        __syncthreads();
        int old_val = dist[global_i * padded_n + global_j];
        int new_val = dist[global_i * padded_n + (B + k)] + dist[(B + k)*padded_n + global_j];
        if (new_val < old_val) {
            dist[global_i * padded_n + global_j] = new_val;
            printf("%d\n", global_i * padded_n + global_j);
        }
    }
}
```

在這個部份我使用 `printf("%d\n", GLOBAL_I_padded + global_j)` 以確保 同一個

Warp 取用的 Global Memory 是連續的:



```
PROBLEMS  DEBUG CONSOLE  COMMENTS  TERMINAL  OUTPUT  PORTS

28289
28290
28291
28292
28293
28294
28295
28296
28297
28298
28299
28300
28301
28302
28303
28304
28305
28306
28307
28308
28309
28310
28311
28312
28313
28314
28315
28316
28317
28318
28319
1408
1409
```

印出的結果如上所示，每 32 一個 warp。

- 這樣做之所以可以優化的關鍵為 Warp 中的 Threads 同時存取連續位址，可以減少記憶體存取延緩。

- Shared Memory

- 在各個 Phase 的 Kernel 中，宣告並使用 Shared Memory，如下所示：

```
__global__ void fw_phase3(int *dist, int padded_n, int r, int numBlocks)
{
    int bx = blockIdx.x;
    int by = blockIdx.y;
    if (bx == r || by == r) return;

    __shared__ int s_row[BLOCK_SIZE][BLOCK_SIZE], s_col[BLOCK_SIZE][BLOCK_SIZE];

    int i = threadIdx.y;
    int j = threadIdx.x;

    int B = r * BLOCK_SIZE;
    int global_i = by * BLOCK_SIZE + i;
    int global_j = bx * BLOCK_SIZE + j;
```

- 這樣可以減少 Global Memory 存取次數：透過 Shared Memory 快取頻繁使用的資料，減少對 Global Memory 的存取。

## • Blocking Factor Tunning

- 這個部分的實作我是把 BLOCK\_SIZE 從 32 增加到 64，並相應調整 Kernel 中的 Shared Memory 配置。
- 更大的 BlockSize 允許更多資料一次載入 Shared Memory，提高資料重用率。
- 這部分的優化有在上面詳細做解釋了。

## • Unroll

- 使用 Loop Unrolling，可以減少迴圈控制指令的開銷，並提升指令級並行度（ILP），從而加速運算。
- 在 Kernel 中，對主要的迴圈使用 #pragma unroll 的程式指令。

```
#pragma unroll 64
for (int k = 0; k < 64; ++k) {
    idx_1 = min(idx_1, s_row[i][k] + s_col[k][j]);
    idx_2 = min(idx_2, s_row[I_32][k] + s_col[k][j]);
    idx_3 = min(idx_3, s_row[i][k] + s_col[k][J_32]);
    idx_4 = min(idx_4, s_row[I_32][k] + s_col[k][J_32]);
}
```

- 這樣做可以讓更多的運算指令能夠在同一時間被執行，提升 GPU 的運算 Throughput。



## Time Distribution

Testcases: c15.1 c20.1 p13k1 p17k1 p21k1 p25k1  
 n : 777 5000 13000 17000 20959 27000

```
##### Time Distribution: c20.1 - N=5000 #####

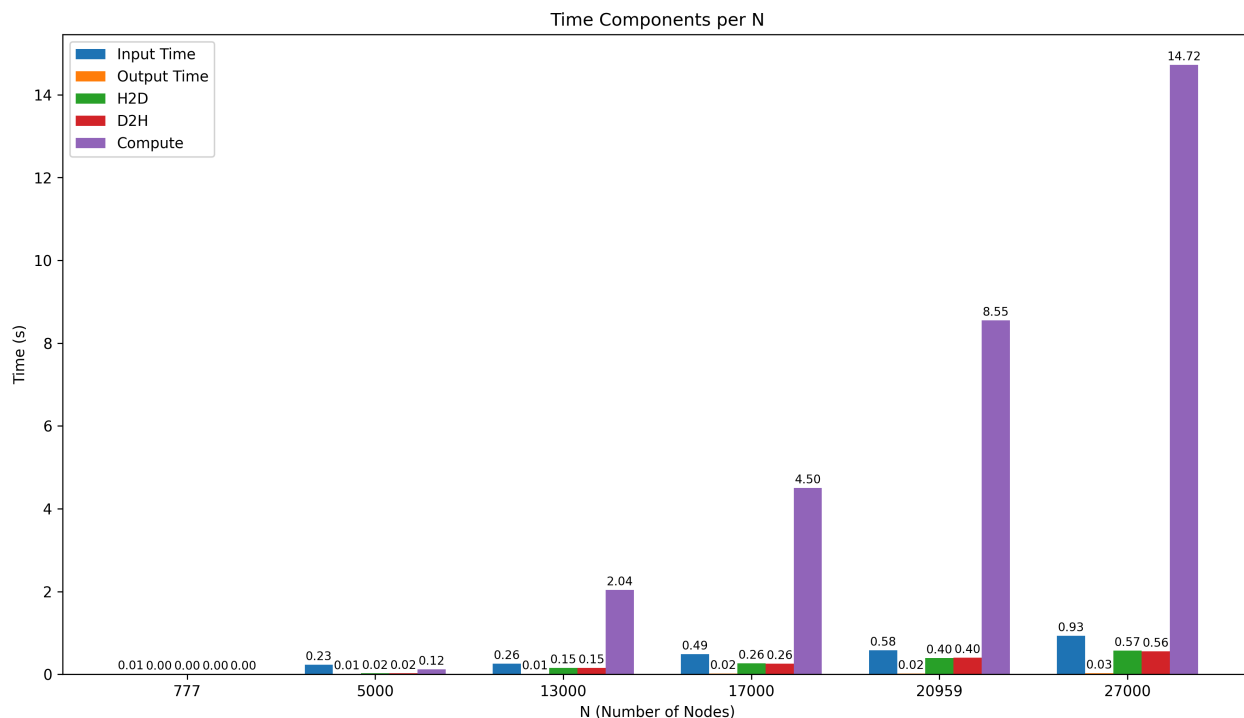
##### Time #####
Input: 0.21
Output: 0.01
Time: 0.48

##### Metric #####
==820337== NVPROF is profiling process 820337, command: ./hw3-2 /home/pp24/share/hw3-2/testcases/c20.1 /dev/null
==820337== Profiling application: ./hw3-2 /home/pp24/share/hw3-2/testcases/c20.1 /dev/null
==820337== Profiling result:
   Type  Time(%)   Time     Calls   Avg       Min       Max  Name
GPU activities:  70.27% 122.61ms    79  1.5520ms  1.5313ms  1.5708ms  fw_phase3(int*, int, int, int)
                13.30%  23.211ms    1  23.211ms  23.211ms  23.211ms  [CUDA memcpy DtoH]
                13.18%  23.005ms    1  23.005ms  23.005ms  23.005ms  [CUDA memcpy HtoD]
                2.06%   3.5948ms   79  45.503us  42.112us  48.640us  fw_phase2(int*, int, int, int)
                1.18%   2.0627ms   79  26.110us  25.600us  26.464us  fw_phase1(int*, int, int)
API calls:      68.68% 174.27ms    2  87.137ms  23.244ms  151.03ms  cudaMemcpy
                30.82%  78.207ms    1  78.207ms  78.207ms  78.207ms  cudaMalloc
                0.43%   1.0890ms  237  4.5940us  2.7430us  276.30us  cudaLaunchKernel
                0.06%   140.81us  114  1.2350us    89ns   59.959us  cuDeviceGetAttribute
                0.00%   11.865us    1  11.865us  11.865us  11.865us  cuDeviceGetName
                0.00%   8.7310us    1  8.7310us  8.7310us  8.7310us  cuDeviceGetPCIBusId
                0.00%   1.1300us    3    376ns    113ns    869ns  cuDeviceGetCount
                0.00%    676ns    1    676ns    676ns    676ns  cuDeviceTotalMem
                0.00%    515ns    2    257ns   139ns    376ns  cuDeviceGet
                0.00%    344ns    1    344ns   344ns   344ns  cuModuleGetLoadingMode
                0.00%    261ns    1    261ns   261ns   261ns  cuDeviceGetUuid

##### Time Distribution: p13k1 - N=13000 #####

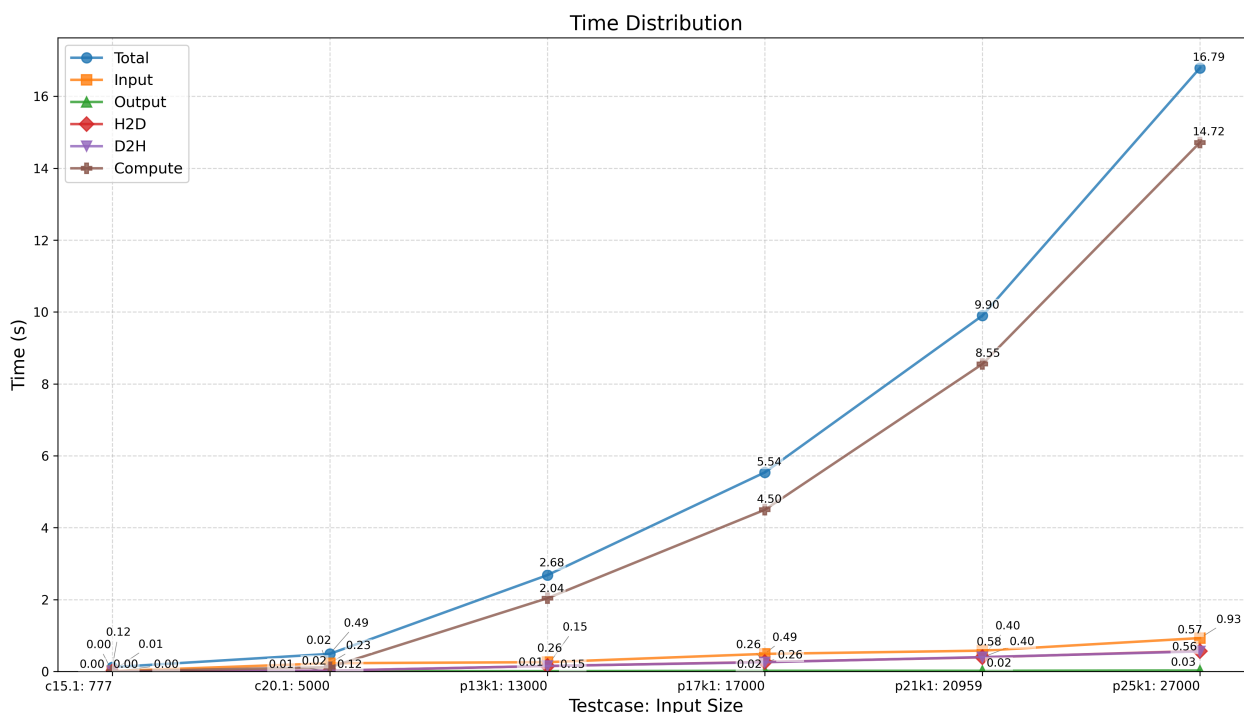
##### Time #####
Input: 0.26
Output: 0.01
Time: 2.66
```

根據不同 Input Size 以下做了 Input time 、 Output Time 、 H2D 、 D2H 、 Compute time ，這幾個不同時間的占比與分析，我的 Time Distribution 的實驗使用的測資為 c15.1 c20.1 p13k1 p17k1 p21k1 p25k1 分別對應到 n = 777, 5000, 13000, 17000, 20959, 27000 這幾個大小的測資。



由上圖可以發現隨著  $n$  size 越大，compute time 為其 bottle neck。其他的時間增長的幅度沒有 compute time 這麼多。

H2D / D2H (綠色與紅色的部分) 也相對較小，顯示資料拷貝雖然隨  $N$  擴增，但比起整體計算依然不算最顯著的瓶頸。



上圖則是時間分佈的折線圖，也可以清楚的看到 compute time 增長的幅度為主要花費的時間，因為 Floyd-Warshall 本身時間複雜度是  $O(n^3)$ ，當資料規模增長時，計算量急遽攀升。

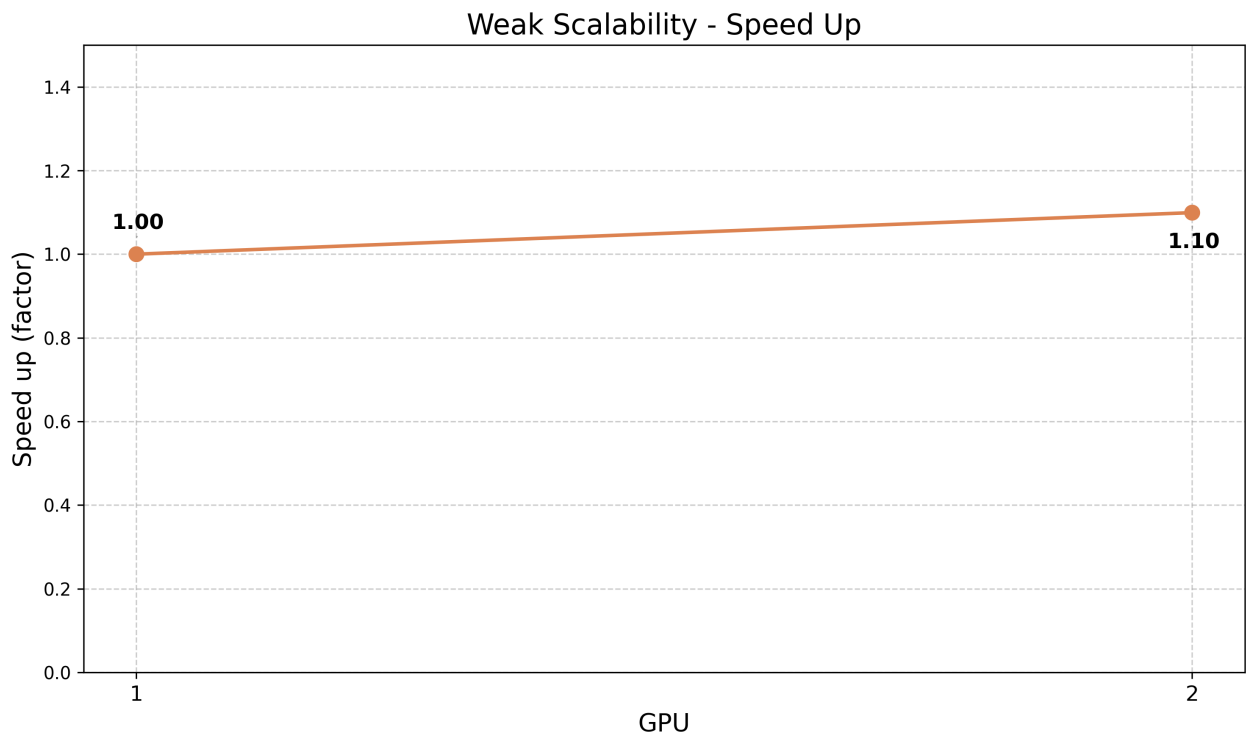
# Profiling Results (hw3-3)

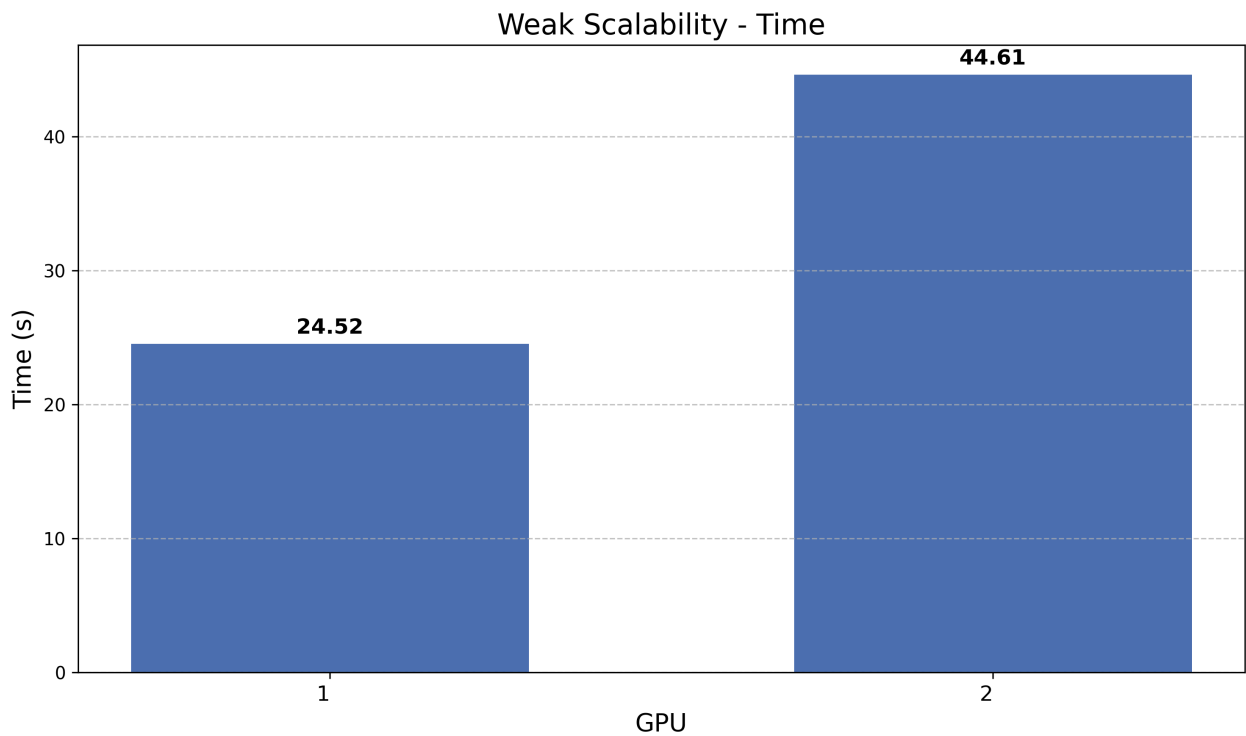
## Weak Scalability

- Testcases use:

- pk29k1 -> GPU: 1
- pk42k1 -> GPU: 2

在 Weak Scalability 的實驗中，我使用的測資為 pk29k1 以及 pk42k1，為了測試更大的資料量，2 個 GPU 的效能是否更好，我讓 single GPU 的程式跑 pk29k1 -> n 大小為 28911，讓多 GPU 版本跑 pk42k1 -> n 大小為 42000。





由實驗結果可以發現，實際上在資料量增加到將近兩倍時，2 個 GPU 版本的效能只比單 GPU 快 1.10 倍，這樣的實驗結果代表了實際上多 GPU 並不一定會導致計算效能加速呈線性增長，原因是因為多 GPU 還會涉及到 GPU 與 GPU 之間的溝通，Floyd-Warshall 每一個 Round (Phase 1, 2, 3) 均需要傳遞 pivot row/column 資料到其它 GPU，或經由 Host 做交換。通訊、同步與資料整合帶來顯著的 overhead，特別在資料量更大時，這些 overhead 也被放大。還有涉及到資料量的增長，pk42k1 ( $n=42000$ ) 的 Floyd-Warshall 計算量大約是 pk29k1 ( $n=28911$ ) 的  $(28911/42000)^3$  倍，即變使用兩張 GPU，也無法抵消整體計算量翻倍開銷。最後是因為 Floyd-Warshall 本身的演算法，具有很高的 data dependency，因而限制了多 GPU 的效能發揮。

# Experiment on AMD GPU

hw3-2.hip

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <hip/hip_runtime.h>
4  #include <sys/time.h>
5  #define US_PER_SEC 1000000
6  const int INF = ((1 << 30) - 1);
7  #define BLOCK_SIZE 64
8  using namespace std;
9  // Function prototypes
10 void read_input(const char *filename, int **host_matrix, int &n, int &m, int &padded_n);
11 void write_output(const char *filename, int *host_matrix, int n, int padded_n);
12
13 // CUDA kernels
14 __global__ void fw_phase1(int *dist, int padded_n, int r);
15 __global__ void fw_phase2(int *dist, int padded_n, int r, int numBlocks);
16 __global__ void fw_phase3(int *dist, int padded_n, int r, int numBlocks);
17
18 int main(int argc, char *argv[])
19 {
20     struct timeval start, end;
21     double time;
22     gettimeofday(&start, NULL);
23
24     int n, m, padded_n;
25     int *host_matrix = NULL;
26
27     // Read input graph
28     read_input(argv[1], &host_matrix, n, m, padded_n);
29
30     // Allocate device memory
31     int *device_matrix;
32     hipMalloc(&device_matrix, padded_n * padded_n * sizeof(int));
33     hipMemcpy(device_matrix, host_matrix, padded_n * padded_n * sizeof(int), hipMemcpyHostToDevice);
34     int numBlocks = padded_n / 64;
35
36     // dim3 threads(32, 32); // Each block has 32 x 32 threads
37     // dim3 gridPhase1(1, 1);
38     // // dim3 gridPhase2(numBlocks, 1);
39     // dim3 gridPhase2(numBlocks, 2); // blockIdx.y --> {0: pivot row} --> {1: pivot column}
40     // dim3 gridPhase3(numBlocks, numBlocks);
41
42     // Main loop
43     for (int r = 0; r < numBlocks; ++r)
44     {
45         // Phase 1: Process pivot block
46         fw_phase1<<<dim3(1, 1), dim3(32, 32)>>>(device_matrix, padded_n, r);
47         // Phase 2: Process pivot row and column
48         fw_phase2<<<dim3(numBlocks, 2), dim3(32, 32)>>>(device_matrix, padded_n, r, numBlocks);
49         // Phase 3: Process remaining blocks
50         fw_phase3<<<dim3(numBlocks, numBlocks), dim3(32, 32)>>>(device_matrix, padded_n, r, numBlocks);
51     }

```

hw3-2-nvidia

pp24s094 40 41 247.44

hw3-2-amd

## hw3-2-amd Scoreboard

### Legend:

- AC

TLE

TLE+

NA
- Each testcase will run for 10s+time limit
- NA is not accepted. It can be wrong answer or timeout
- The rank is based on Time + Penalty

User	Rank	Passed	Time	Penalty
pp24s039	1	51	220.32	
pp24s038	2	51	225.31	
pp24s092	3	51	232.90	
pp24s012	4	51	234.90	
pp24s062	5	51	237.29	
pp24s077	6	51	248.17	
pp24s089	7	51	250.34	
pp24s011	8	51	250.71	
pp24s085	9	51	253.09	
pp24s094	10	51	253.13	

hw3-3-nvidia

pp24s094	18	7	94.21
----------	----	---	-------

hw3-3-amd

## hw3-3-amd Scoreboard

### Legend:

- AC

TLE

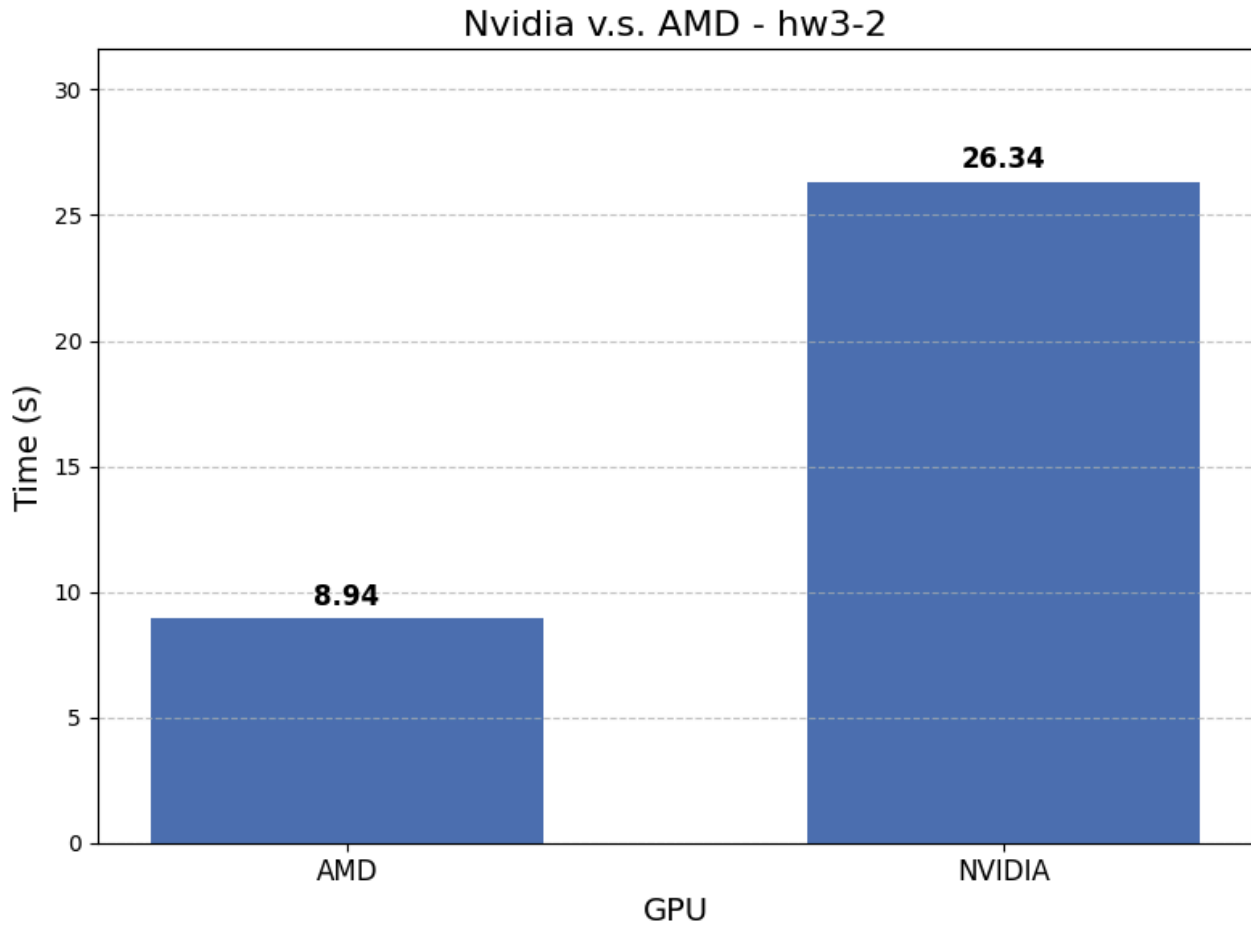
TLE+

NA
- Each testcase will run for 10s+time limit. If the p and show the result with TLE+
- NA is not accepted. It can be wrong answer, sec
- The rank is based on Time + Penalty time

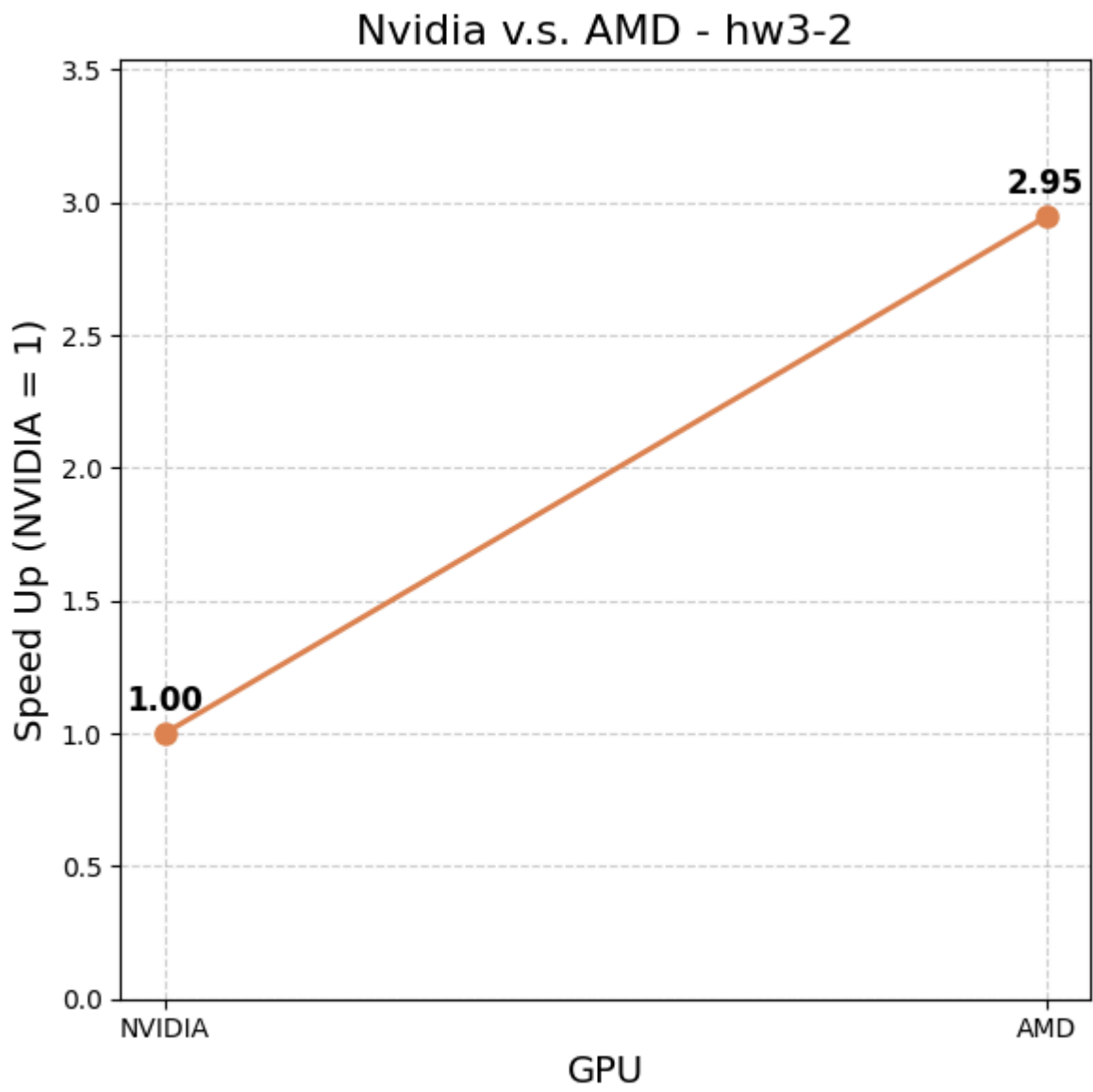
User	Rank	Passed	Time	Penalty
pp24s020	1	7	32.54	
pp24s077	2	7	34.05	
pp24s104	3	7	34.94	
pp24s062	4	7	37.00	
pp24s094	5	7	37.29	

## hw3-2 Experiment result

testcase: p29k1

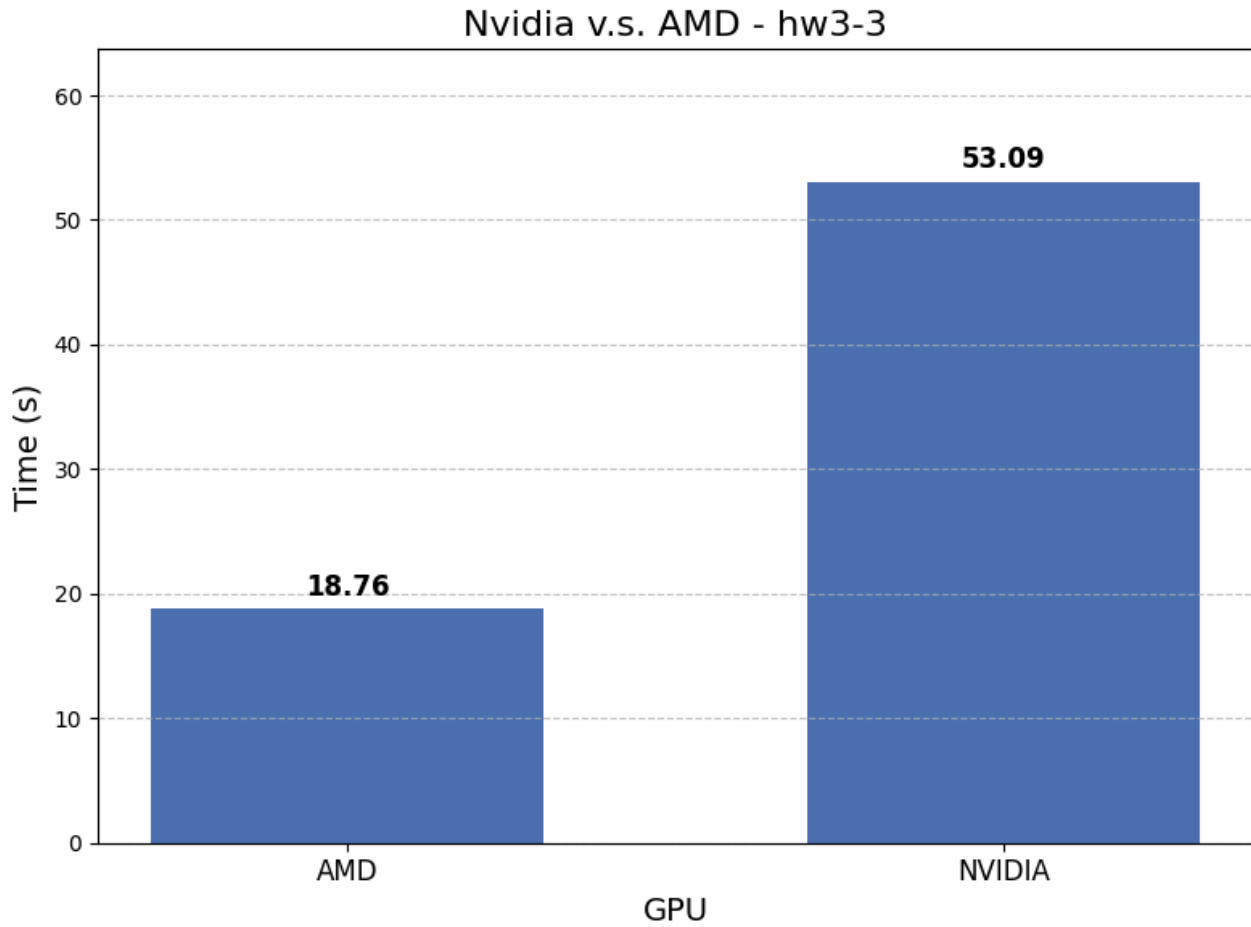




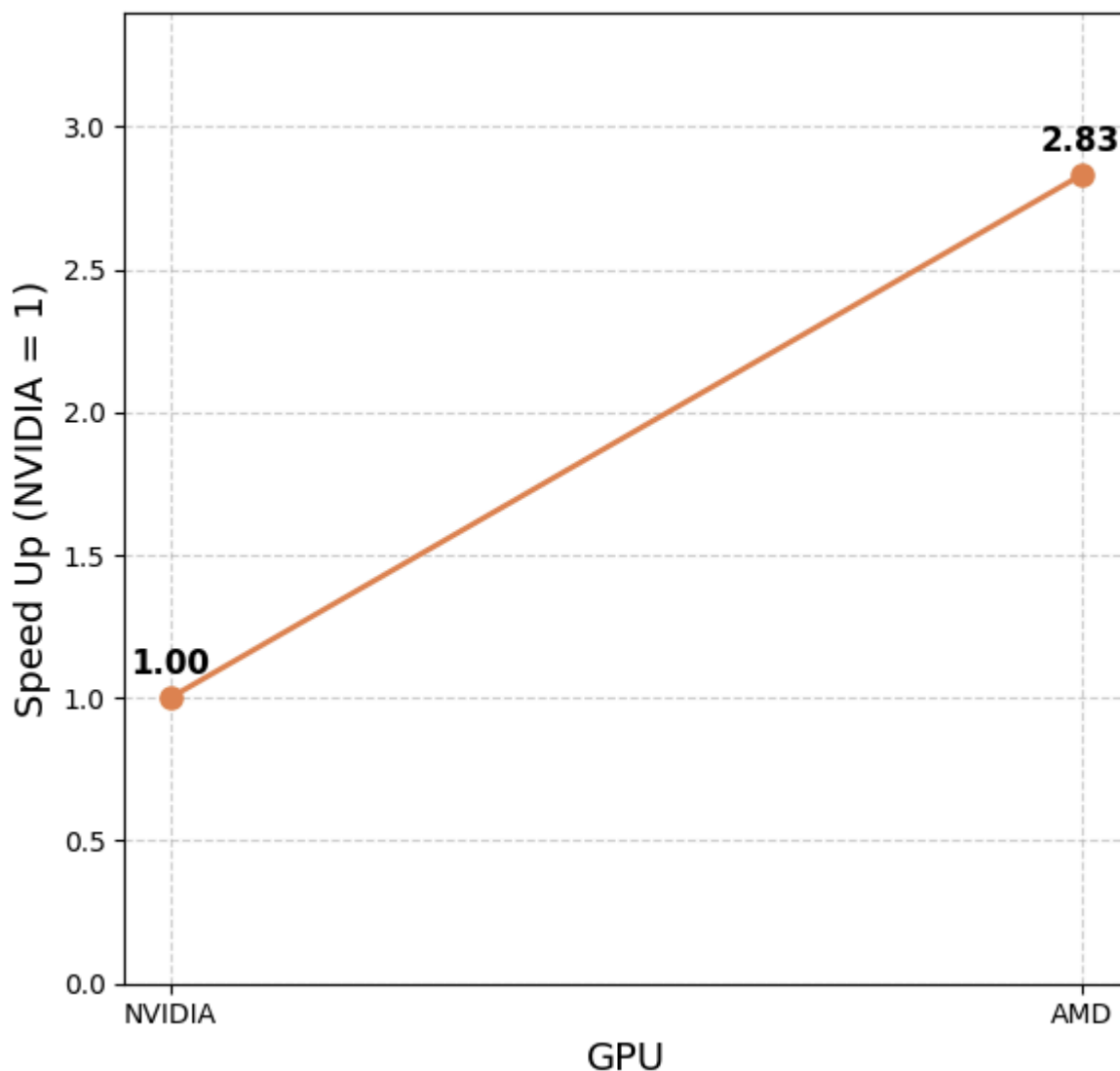


## hw3-3 Experiment result

testcase: c07.1



## Nvidia v.s. AMD - hw3-3



### Discussion

不管是在單 GPU 的版本還是多 GPU 版本中，AMD 的版本速度都比 Nvidia 快。

### Other Optimization

在大筆測資比較難 Accept 的情況下，最後因為發現後面測資光是 `read_input` 所需要花費的時間就很多了，加上 `read_input` 裡面有雙層 `for loop`，這讓我想要將它也寫成 `kernel function`:

Before:

```

void read_input(const char *filename, int **host_matrix, int &n, int &m, int &padded_n)
{
    FILE *file = fopen(filename, "rb");
    fread(&n, sizeof(int), 1, file); // 5
    fread(&m, sizeof(int), 1, file); // 10

    padded_n = ((n + 64 - 1) / 64) * 64;
    size_t size = padded_n * padded_n * sizeof(int); // 68*68
    *host_matrix = (int *)malloc(size);

    for (int i = 0; i < padded_n; ++i)
    {
        for (int j = 0; j < padded_n; ++j)
        {
            (*host_matrix)[i * padded_n + j] = (i == j) ? 0 : INF;
        }
    }

    // Read all edges at once
    int *edges = (int *)malloc(3 * m * sizeof(int));
    fread(edges, sizeof(int), 3 * m, file);

    for (int i = 0; i < m; ++i)
    {
        (*host_matrix)[edges[3 * i] * padded_n + edges[3 * i + 1]] = edges[3 * i + 2];
    }
    fclose(file);
}

```

After:

```

__global__ void setEdges(int *d_mat, int *d_edges, int m, int padded_n)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < m) {
        int u = d_edges[3 * idx];
        int v = d_edges[3 * idx + 1];
        int w = d_edges[3 * idx + 2];
        d_mat[u * padded_n + v] = w;
    }
}

__global__ void initMatrix(int *d_mat, int padded_n)
{
    // 一維 Thread 索引
    const unsigned int i = threadIdx.y;
    const unsigned int j = threadIdx.x;
    const int global_i = i + blockIdx.y * 64;
    const int global_j = j + blockIdx.x * 64;

    d_mat[global_i * padded_n + global_j] = global_i == global_j ? 0 : INF;
    d_mat[(global_i + 32) * padded_n + global_j] = global_i + 32 == global_j ? 0 : INF;
    d_mat[global_i * padded_n + global_j + 32] = global_i == global_j + 32 ? 0 : INF;
    d_mat[(global_i + 32) * padded_n + global_j + 32] = global_i + 32 == global_j + 32 ? 0 : INF;
}

```

在 main function 裡面呼叫:

```
initMatrix<<<dim3(padded_n / 64, padded_n / 64), dim3(32, 32)>>>(device_matrix, padded_n);
```

```
{
    int blocks = (m + threads - 1) / threads;
    setEdges<<<blocks, threads>>>(device_matrix, deviceEdges, m, padded_n);
}
```

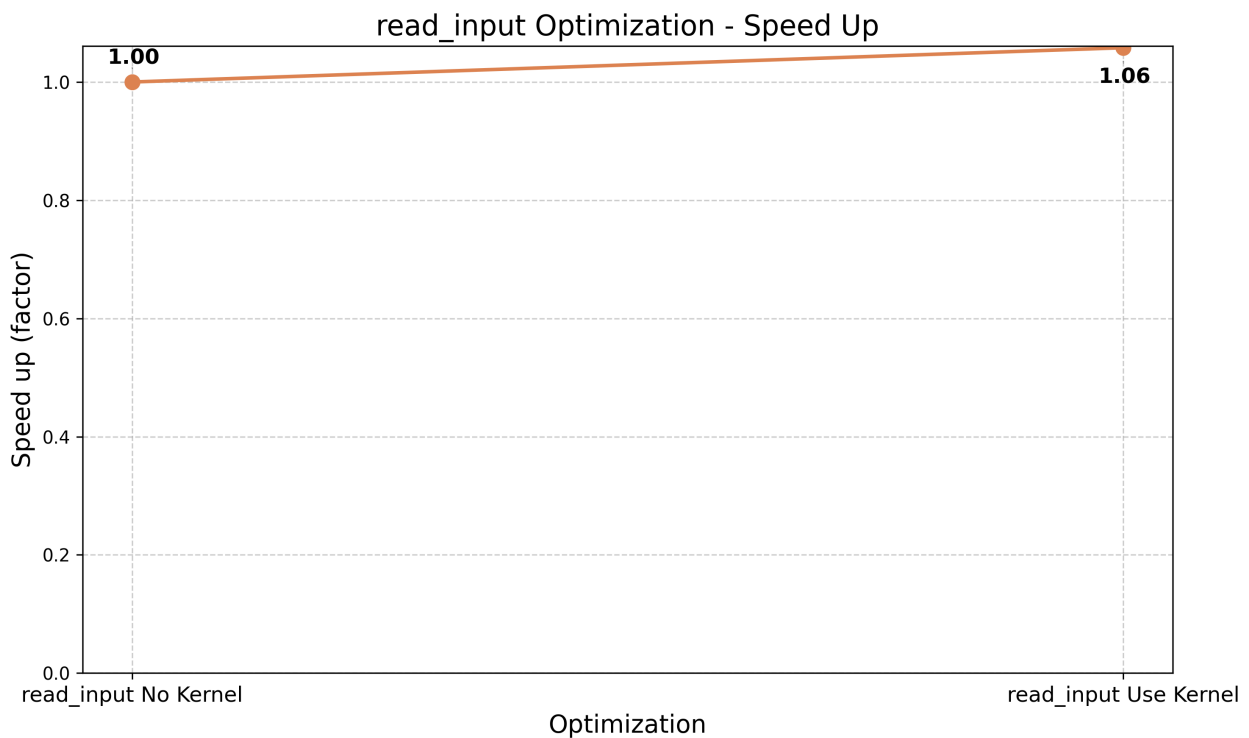
這樣在 p30k1 這個 testcase 大概加快了1.5秒:

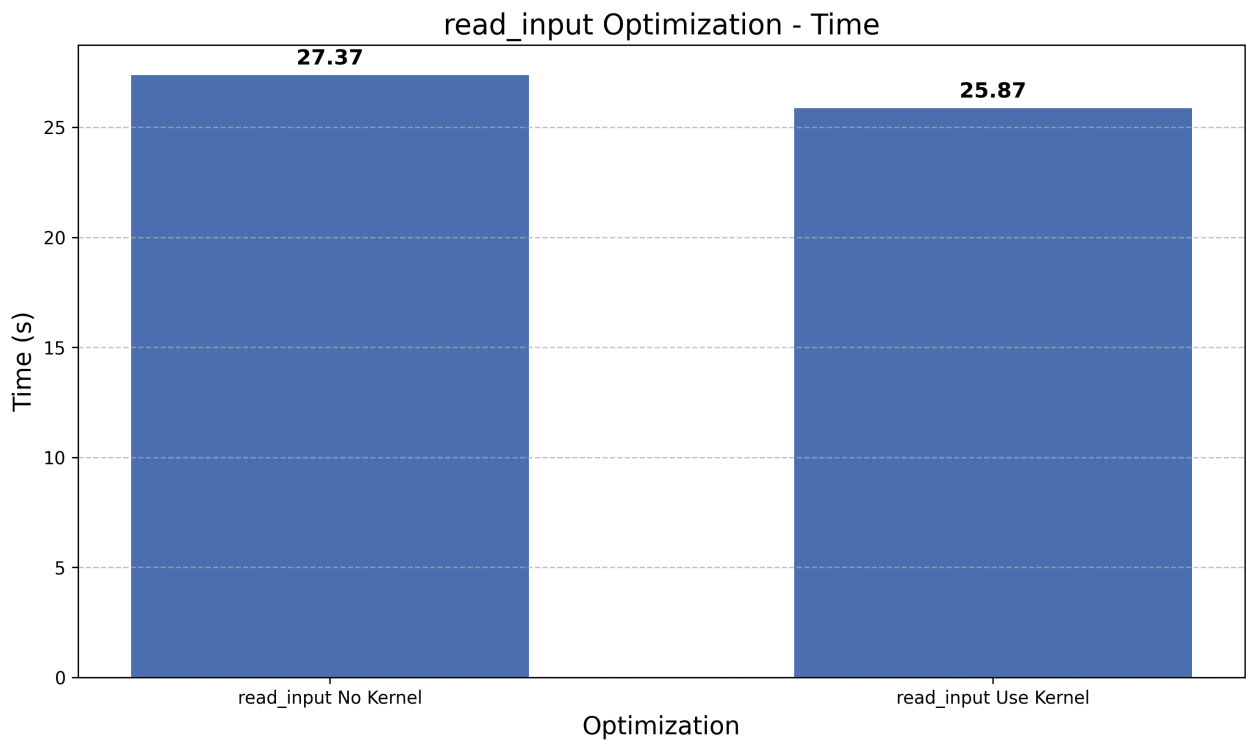
Experiment result:

```
##### Optimization: Unroll #####
##### Time #####
Time: 27.37

##### Optimization: final #####
##### Time #####
Time: 25.87
```

Speedup 加快 1.06 倍:





## Conclusion

在這次作業中，實作了 Floyd-Warshall 演算法的 CPU 及 GPU 版本，並針對不同平台進行了一系列優化。CPU 版本是透過 OpenMP 平行化與 SIMD 指令加速，有效提升了運算效能。而在單一 GPU 版本中，採用了 Padding、Coalesced Memory、Shared Memory、Blocking Factor 及 Loop Unroll 等多項優化技術，顯著減少了 Global Memory 存取次數，提升了記憶體 bandwidth 利用率與運算效率。

在多 GPU 實作中，雖然理論上應能提升運算速度，但實驗結果顯示效能增幅並不如預期，僅達到約 1.10 倍的加速，主要原因在於 GPU 之間的資料傳輸與同步所帶來的額外開銷。此外，Floyd-Warshall 演算法本身的高度資料依賴性限制了多 GPU 的效能發揮。

在這次作業中，發現後面超過 p30k1 的測資大家的通過率都很低，原因透過 profiling 結果來看是因為卡在 computation time 這個 bottleneck，畢竟 Floyd-Warshall 演算法中涉及很多 for loop，很多資料都有相依性，演算法的時間複雜度高達  $O(n^3)$ ，這才是這份作業最難優化的部分。