# CS542200 Parallel Programming
# Homework 3: All-Pairs Short

student ID : 114062616 Name : 莊岱倫

# 1. Implementation

## 1.a Which algorithm do you choose in HW3-1?

- Blocked Floyd–Warshall 演算法
- Block-based round 更新（pivot block → pivot row/col → remaining blocks）
- OpenMP 平行化

---

## 1.b How do you divide your data in HW3-2 and HW3-3?

### HW3-2 (Single GPU – CUDA)

- **Padding 方法**

```
nPad = ((n + B - 1) / B) * B;
```

將原本的問題補齊至對齊block size，所以整個matrix變成nPad x nPad
矩陣切成 rounds×rounds 個 blocks

```
rounds = nPad / B;
```

每個phase處理不同區域的block，依序進行完成一個round

- **Block decomposition**

  **BlockDim = 32 X 32**

  **phase1 : Pivot block**

```
dim3 gridPhase1(1, 1);
phase1<<<gridPhase1, blockDim>>>(Dist_d, nPad, B, r);
```

**phase2 : Pivot row , Pivot col**

```
dim3 gridPhase2(rounds - 1, 2);
phase2<<<gridPhase2, blockDim>>>(Dist_d, nPad, B, r, rounds);
```

**phase3 : 剩下來的那些block**

```
dim3 gridPhase3(rounds, rounds); phase3<<<gridPhase3,
blockDim>>>(Dist_d, nPad, B, r, rounds);
```

- **Thread mapping**

  每 thread 處理 2×2 micro-tile（4 elements）

```
(off_i , off_j)
(off_i , off_j + 1)
(off_i + 1 , off_j)
(off_i + 1 , off_j + 1)
```

  設計原因

  讀進來一次就可以在更新的地方用**4**次

  使得**thread**存取資料時是連續的

## HW3-3 (Multi-GPU)

Padding 和 thread mapping 的部分跟hw3-2相同這邊針對多GPU版本不同之處說明

- **Divide between GPUs**

  整個 **nPad × nPad** 的距離矩陣先被切成 **B = 64** 的區塊（**block tiles**）:

```
Block grid size = (rounds × rounds)
where rounds = nPad / 64
```

  每 GPU 要負責的 block rows

```
rounds = nPad / B;
blocks_per_gpu = rounds / num_gpus;
remainder      = rounds % num_gpus;
```

根據GPU tid分配

```
start_block = tid * blocks_per_gpu + (tid < remainder ? tid :
remainder)
num_blocks  = blocks_per_gpu + (tid < remainder ? 1 : 0)
end_block   = start_block + num_blocks
```

每張 GPU 分到一段連續的 row blocks 因為 Floyd–Warshall 的運算是以 row 為核心(每次更新都是固定 i row 去讀 k row 和 k column)，所以把矩陣按 block-rows 切給不同 GPU，可以讓每張 GPU 只負責自己那幾段 row 的更新，不會互相衝突，記憶體也只要存屬於自己的區塊;同時資料在每張 GPU 裡是連續的，拷貝最有效率，Phase 1/2/3 的資料交換也最簡單

每張GPU拿到 num_blocks_of_GPU × 64 rows × nPad columns

- **Block decomposition**

  **Phase 1 (Pivot block)**

```
phase1<<<1, blockDim>>>(Dist_local, nPad, r, start_block);
```

  Phase 2: (Pivot row , Pivot column)

  **Phase2 col (兩張GPU都要執行)**

```
phase2_Col<<<num_blocks, blockDim>>
(Dist_local, PivotBlock, nPad, r,   start_block, num_blocks);
```

  tileA = Dist_local[i][r]

  tileB = PivotBlock(來自 owner GPU)

  讓所有 block rows 的 column r 都反映經過 pivot block (r,r) 更新後的最短路。

  **Phase2 row (只有owner GPU執行)**

```
if (tid == owner_id) {
```

```
    phase2_Row<<<rounds, blockDim>>>(Dist_local, nPad, r,
start_block, rounds);
}
```

owner GPU 會 launch rounds 個 blocks

j = 0..rounds-1 但會跳過 j=r去更新所有 (r, j) blocks

**Phase3 (剩下的blocks)**

```
dim3 gridPhase3(rounds, num_blocks);
phase3<<<gridPhase3, blockDim>>>
(Dist_local, RowBuffer, nPad, r, start_block);
```

| **Phase** | 哪些 **Blocks** | 誰負責 |
|---|---|---|
| Phase 1 | (r, r) pivot block | 只有 owner GPU |
| Phase 2 Row | (r, j) | owner GPU |
| Phase 2 Col | (i, r) | 各 GPU 各處理自己的 block |
| Phase 3 | (i, j) 其他區塊 | 各 GPU 各處理自己的 block |

---

# 1.c What's your configuration in HW3-2 and HW3-3? And why?

## HW3-2

- **Blocking factor B = 64**

  因為 64 x 64 的block factor 可以最大化的reuse資料，並且也不會超過硬體限制，同時也對應到我每個thread負責4 elements的實作策略

- **Threads per block = 32 × 32 = 1024**

  透過 32×32 = 1024 threads 正好可完整覆蓋一個 64×64 的
  Floyd–Warshall block tile，因為每個 thread 處理一個 2×2 micro tile

- **Grid**

  Phase 1: 1×1

  Phase 2: (rounds−1) × 2

  Phase 3: rounds × rounds

  根據演算法邏輯每個tile分配一個block

  Phase3是因為index mapping所以多分配了一點block讓讀取時可以連續

## HW3-3

- **Blocking factor B = 64**

  因為 64 x 64 的block factor 可以最大化的reuse資料，並且也不會超過硬體限制，同時也對應到我每個thread負責4 elements的實作策略

- **Threads per block = 32 × 32 = 1024**

  透過 32×32 = 1024 threads 正好可完整覆蓋一個 64×64 的
  Floyd–Warshall block tile，因為每個 thread 處理一個 2×2 micro tile

- **Grid**

  Phase 1: 1×1

  Phase 2 row: (rounds−1) × 2

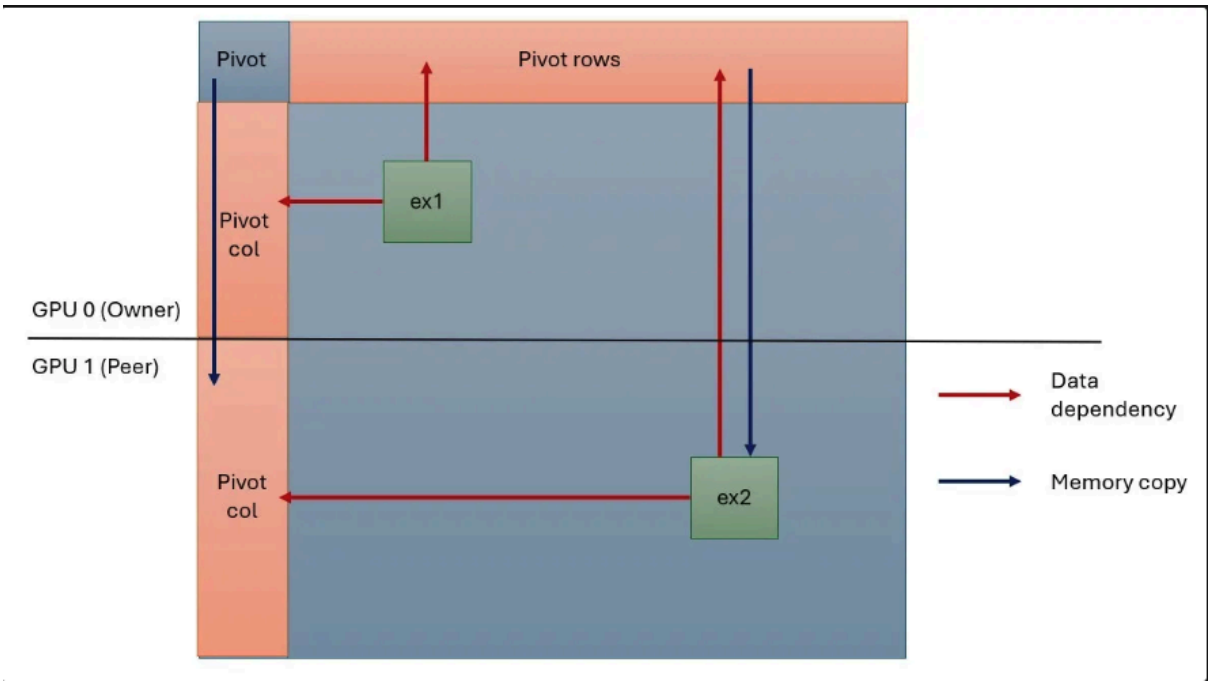
  Phase 2 col : (rounds−1) × 2

Phase 3: rounds × rounds

這裡為了配合多GPU的通訊設計了兩個Phase2，讓GPU之間的通訊可以只傳必須要的資料

---

# 1.d How do you implement the communication in HW3-3?




我採用 多 GPU + peer-to-peer (P2P) 直接傳輸 的方式來實作 Floyd-Warshall 的跨 GPU 溝通。整體流程完全對應 blocked FW 的三個 Phase，分別傳遞：

1. pivot block（B×B）
2. pivot row（B×nPad）

**Phase 1**：**Pivot block** 由唯一 **owner GPU** 計算後「廣播」給所有 **GPU**

在 round r：

1. 計算出 哪個 **GPU** 擁有第 **r** 個 **block row**（owner）。
2. owner GPU 執行 Phase1 更新 pivot block (r, r)
3. 計算完成後，owner 把 pivot block（B×B）複製到 shared buffer：
4. owner GPU 對所有其他 GPU 做硬體 P2P 廣播：

**Phase 2**：**Pivot Row** 由 **owner** 更新後廣播給所有 **GPU**

Phase2 分成：

- **pivot column**：所有 GPU 各自更新自己持有的 column block (i, r)
- **pivot row**：只有 owner 擁有 row r，因此 owner GPU 執行 Phase2_Row

Pivot row

owner GPU 會持有：Pivot block (r,r) Pivot row (r,*)

```
cudaMemcpy(RowBuffer, Dist_local + offset, B*nPad*sizeof(ushort));
```

把整個 pivot row（B × nPad）存到 RowBuffer。

因為其他 GPU 完全不保存 row r，所以這個 buffer 是跨 GPU 唯一的 pivot row 資料來源。

然後再次Procast到其他GPU

```
cudaMemcpyPeerAsync(row_buf_ptrs[p], p, RowBuffer, tid,
B*nPad*sizeof(ushort));
```

所以每張GPU就會有一份複製好的Pivot row Phase3時要使用

**Phase 3**：所有 **GPU** 使用 **PivotBlock + PivotRow** 計算自己區域的 **tile**

在 Phase 3：

1. 所有 GPU 都已經擁有：
    - PivotBlock（B×B）
    - PivotRow（B×nPad）
2. 每個 GPU 使用自己 local 的 pivot column（從 Dist_local 載入）
3. 全 GPU 平行更新每個 (i, j) tile

完全沒有跨 GPU 互相依賴，因此 Phase 3 不需要溝通。

同步機制

cudaDeviceSynchronize() – GPU kernel 層級同步

確保：

owner GPU 在 Phase1、Phase2 內的 kernel 已完成

資料寫入 PivotBlock、RowBuffer 已完成

(2) OpenMP barrier – 多 GPU thread 間同步

確保：

所有 GPU 等待 owner 廣播完成

所有 GPU 等待彼此收到 pivot 資料

---

# 1.e Briefly describe your implementations in diagrams / figures / sentences

### 1. 前置處理（**Pre-processing**）

### (1) Padding 與 Dist 初始化

我先將矩陣 padding 到 nPad = ceil(n / B) * B，使矩陣可完整以 B×B block 處理。

啟動 kernel init_dist_kernel：

- 對角線初始化為 0
- 其他元素初始化為 INF（USHORT_MAX）
- 同時處理邊界 padding 區域

### (2) Edge 更新

使用 update_edges_kernel 逐條將 input 中的 (u, v, w) 更新到 Dist。

---

**2. Blocked Floyd-Warshall 核心**

我採用 **Tile Size B = 64**，並用 **32×32 threads** 來處理：

每個 thread 更新 2×2 微分塊，使 1024 threads × 4 elements = 4096 elements = 剛好一個 64×64 tile。

如此可以：

- 利用 **register** 暫存 **4** 個距離值，降低 global 和 shared memory traffic
- 提升 **thread** 的工作量（**thread-level locality**）
- 與 CUDA warp size（32）完美配合

**3.** 三階段 **Blocked FW**（**Phase 1 / Phase 2 / Phase 3**）

**Phase 1**：更新樞紐區塊 **(Pivot Block, block (r,r))**

運行流程

1. 取出第 r 回合中對角線上的那個 64×64 區塊。
2. 所有 threads 合作把這個 block 載到 shared memory。
3. 用 Floyd–Warshall 的 k-loop，只更新這個 block 內部的距離。
4. 更新完後再寫回 Dist。

結果：此**pivot block** 會供後面兩階段使用。

---

**Phase 2**：更新第 **r** 列與第 **r** 行的其他區塊

概念：**Pivot block** 幫助整條 **row & column** 變更短

1. 針對第 r row 和第 r column 的其他 blocks 各跑一次（不包含 pivot block）。
2. 同時載入(到shared memory)：
    - 該 block 自己的資料
    - 樞紐 block 的資料
3. 用 pivot block 當「中繼點」，更新 row/column 上的距離。
4. 寫回更新後的內容。

結果：整條第 **r row** 與第 **r column** 已經反映經過 **pivot k** 的最短路。

---

**Phase 3**：更新其他不在第 **r row / r column** 的所有區塊

概念：用 **pivot row + pivot column** 影響全地圖

1. 對所有「不在 row r、也不在 col r」的 blocks 分別進行更新。
2. 每個 block 載入：
    - pivot row slice
    - pivot column slice
    - block 本身的資料
3. 用 (i → r → j) 路徑測試是否能縮短 block 內的距離。
4. 更新後寫回。

**4. Output**（後處理）

我使用 mmap + pinned memory（cudaHostRegister）產生輸出檔案：

1. dist_to_out_kernel 把 Dist_d 轉成 int
2. cudaMemcpyAsync + stream 同步移到 host
3. 直接寫進 memory-mapped output file（完全避免中間拷貝）

---

# 2. Profiling Results (HW3-2)

Profile instruction

```
srun nvprof --kernels "phase3*" --metrics \
achieved_occupancy,sm_efficiency,shared_load_throughput,shared_store_throughput,\
gld_throughput,gst_throughput \
./hw3-2 ./testcases/c21.1 ./out
```

```
Invocations                               Metric Name                         Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: phase3(unsigned short*, int, int, int, int)
          9                        achieved_occupancy                      Achieved Occupancy    0.816950    0.874147    0.836668
          9                              sm_efficiency                 Multiprocessor Activity      70.12%      71.40%      70.71%
          9                              gld_throughput                  Global Load Throughput  93.756GB/s  95.270GB/s  94.514GB/s
          9                              gst_throughput                 Global Store Throughput  46.878GB/s  47.635GB/s  47.257GB/s
          9                       shared_load_throughput          Shared Memory Load Throughput  2250.1GB/s  2286.5GB/s  2268.3GB/s
          9                      shared_store_throughput         Shared Memory Store Throughput  93.756GB/s  95.270GB/s  94.514GB/s
```

| | |
|---|---|
| occupancy | 83% |
| sm efficiency | 70% |
| shared memory load throughput | 2268.3 GB/s |
| shared memory store throughput | 94.514 GB/s |
| global load throughput | 94.514 GB/s |
| global store throughput | 47.257 GB/s |

83% 的 occupancy 與 70% 的 SM efficiency 代表 warp 排程與計算管線利用良好，而 global load/store throughput 僅約 30% 的 GTX1080 理論頻寬、同時 shared memory load throughput 高達 2.2 TB/s，顯示 kernel 已成功將運算重心從 DRAM 移入 shared memory，再配合 register 累積更新，因此效能主要受限於 shared memory access latency 與指令管線，而非記憶體頻寬

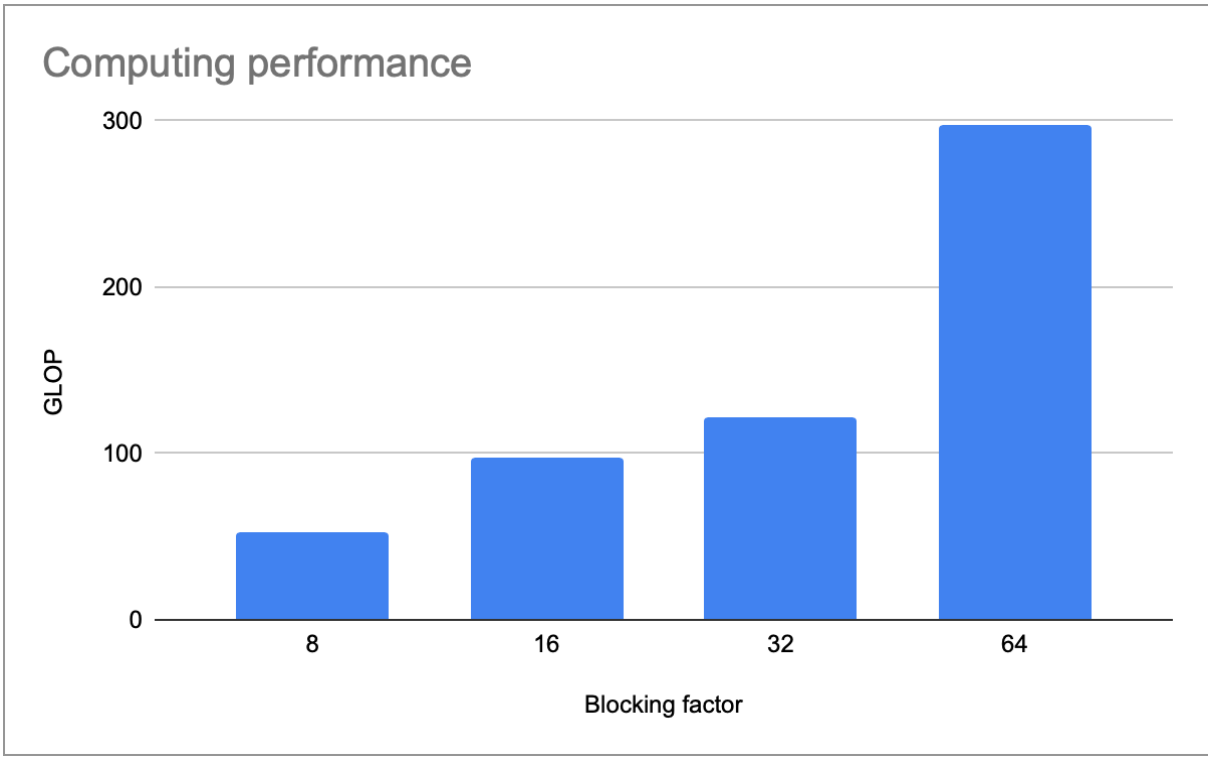
# 3. Experiment & Analysis (NVIDIA GPU)

## 3.a System Spec

我是Apollo gpu cluster 所以不特別放spec

---

## 3.b Blocking Factor (HW3-2)




GLOPS

根據圖中的結果，當 Blocking Factor 從 8 → 16 → 32 → 64 增加時，Computing Performance (GLOP) 呈現明顯的單調遞增，最終在 64 達到最高的 300 GLOP。

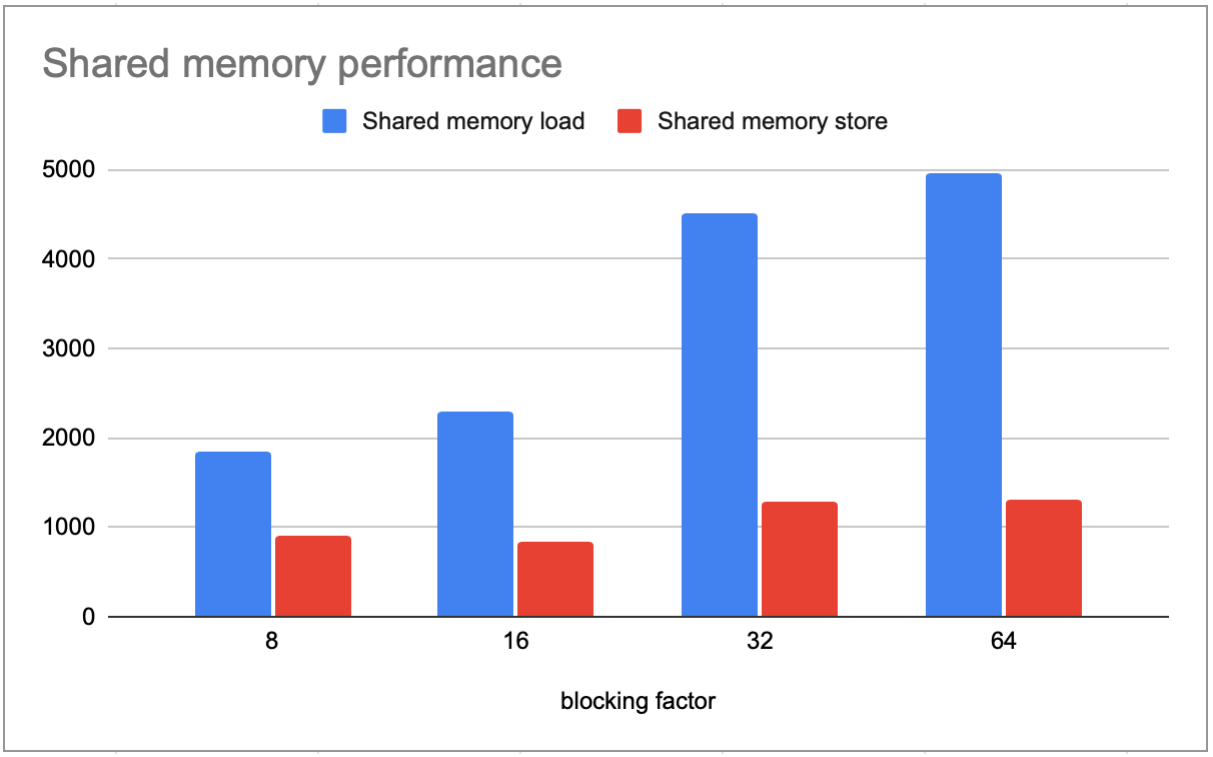
當 block size 越大，表示：

- 同一個 tile 中包含更多的 row / col pair
- 同一個 pivot row/col 可以在 shared memory 被重複使用更多次
- 每載入一次 pivot block 或 tile 可產生更多 FLOPs

所以計算密度（arithmetic intensity）隨 block size 增大而提升。

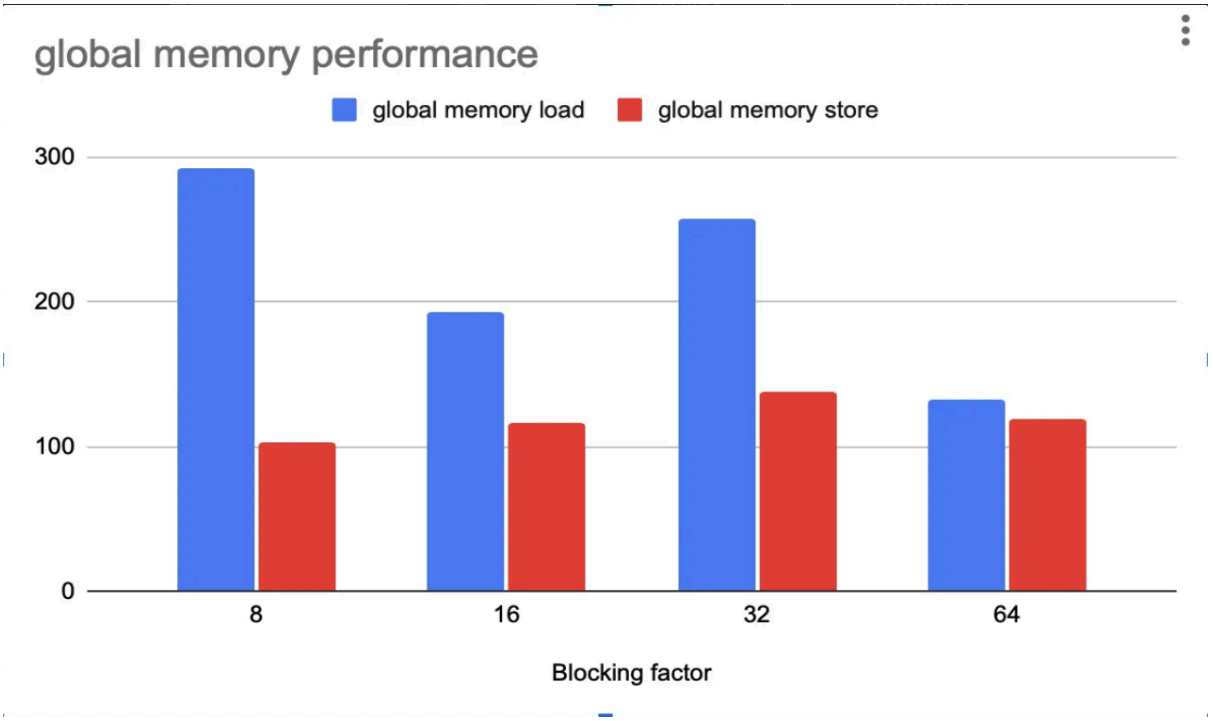
在 block = 64 時：

每次載入的 64×64 tile 帶來最大的 FLOP reuse，因此 GLOP 值最高。




Shared memory throughput

blocking factor = 8 or 16 都會讓shared memory的bandwidth使用不足，只有在**32** 與 **64** 都能吃到 **shared memory** 較高的**bandwidth**但根據**GLOP**的觀察**64**才是更高頻寬且高 **occupancy**的選擇，在這張圖中**64**也是表現最好的**blocking factor**



**Global Memory Bandwidth**

因為有很大部分的資料班移到shared memory重複使用，所以global bandwidth高不代表比較好，要在一個合理的範圍thread一開始搬移至shared memory的bandwidth足夠即可，以圖片來看 8 和16太小kernel 啟動次數太多，反而是64時因為需要的資料都一次搬進去shared memory不用多次班取重複資料，所以bandwidth最低

---

# 3.c Optimization (HW3-2)

優化的過程為每個改善功能的疊加，依照profile看到的方向做優化

**GPU baseline (Time 19.51)**

| | metric | avg | total |
|---|---|---|---|
| 1 | shared_ld_bank_conflict | 0.000000 | 0.000000 |
| 2 | shared_st_bank_conflict | 0.000000 | 0.000000 |
| 4 | achieved_occupancy | 0.947387 | 0.943184 |
| 5 | ipc | 1.758657 | 1.467027 |
| 6 | gld_throughput | 737.412409 | 719.487745 |
| 7 | gst_throughput | 57.513096 | 27.858591 |
| 8 | l2_read_throughput | 737.428298 | 719.535057 |
| 9 | l2_write_throughput | 57.516171 | 27.861855 |
| 10 | shared_efficiency | 0.000000 | 0.000000 |
| 11 | warp_execution_efficiency | 100.000000 | 96.473516 |
| 12 | issue_slot_utilization | 39.625053 | 32.055247 |
| 13 | stall_inst_fetch | 3.657524 | 2.744034 |
| 14 | stall_exec_dependency | 11.086647 | 10.500854 |
| 15 | stall_memory_dependency | 81.676187 | 80.471178 |
| 16 | stall_constant_memory_dependency | 1.034380 | 0.346409 |
| 17 | stall_texture | 0.000024 | 0.000003 |
| 18 | stall_sync | 0.000000 | 0.000000 |
| 19 | stall_pipe_busy | 0.417765 | 0.189777 |
| 20 | stall_other | 7.722840 | 4.179223 |
| 21 | gld_efficiency | 75.982997 | 74.929017 |
| 22 | gst_efficiency | 32.082507 | 20.348786 |

因為完全使用global mem操作，所以global mem throughtput非常，kernel 現在主要被「記憶體存取延遲」嚴重卡住，算力沒有吃滿，優化方向應該集中在降低 memory dependency、減少 global store、提升資料重用

---

Padding (Time 19.07)

| | metric | avg | total |
|---|---|---|---|
| 1 | shared_ld_bank_conflict | 0.000000 | 0.000000 |
| 2 | shared_st_bank_conflict | 0.000000 | 0.000000 |
| 4 | achieved_occupancy | 0.957986 | 0.951264 |
| 5 | ipc | 2.005655 | 1.810458 |
| 6 | gld_throughput | 749.271133 | 740.738237 |
| 7 | gst_throughput | 63.751667 | 30.774228 |
| 8 | l2_read_throughput | 749.289689 | 740.791005 |
| 9 | l2_write_throughput | 63.755385 | 30.778011 |
| 10 | shared_efficiency | 0.000000 | 0.000000 |
| 11 | warp_execution_efficiency | 100.000000 | 99.099340 |
| 12 | issue_slot_utilization | 44.138878 | 39.304763 |
| 13 | stall_inst_fetch | 7.712725 | 3.960537 |
| 14 | stall_exec_dependency | 12.921456 | 12.688551 |
| 15 | stall_memory_dependency | 76.626372 | 75.590427 |
| 16 | stall_constant_memory_dependency | 0.507879 | 0.378951 |
| 17 | stall_texture | 0.000031 | 0.000006 |
| 18 | stall_sync | 0.000000 | 0.000000 |
| 19 | stall_pipe_busy | 0.336278 | 0.243856 |
| 20 | stall_other | 8.570561 | 5.363145 |
| 21 | gld_efficiency | 89.575181 | 89.391811 |
| 22 | gst_efficiency | 34.998890 | 21.228220 |

Padding 明確提升了記憶體存取效率，使指令吞吐與 SM 利用度上升，因為他可以省去一些邊界判斷的檢查

---

Register cacheing (Time = 10.8)

| | metric | avg | total |
|---|---|---|---|
| 1 | shared_ld_bank_conflict | 0.000000 | 0.000000 |
| 2 | shared_st_bank_conflict | 0.000000 | 0.000000 |
| 4 | achieved_occupancy | 0.837795 | 0.829453 |
| 5 | ipc | 2.421754 | 2.339517 |
| 6 | gld_throughput | 914.764590 | 885.182086 |
| 7 | gst_throughput | 22.311331 | 21.589807 |
| 8 | l2_read_throughput | 914.803191 | 885.377149 |
| 9 | l2_write_throughput | 22.319172 | 21.597394 |
| 10 | shared_efficiency | 0.000000 | 0.000000 |
| 11 | warp_execution_efficiency | 100.000000 | 100.000000 |
| 12 | issue_slot_utilization | 50.367436 | 48.660615 |
| 13 | stall_inst_fetch | 4.906819 | 2.788649 |
| 14 | stall_exec_dependency | 34.497757 | 33.866285 |
| 15 | stall_memory_dependency | 9.886983 | 9.521259 |
| 16 | stall_constant_memory_dependency | 1.062738 | 0.837803 |
| 17 | stall_texture | 1.217584 | 1.009083 |
| 18 | stall_sync | 0.000000 | 0.000000 |
| 19 | stall_pipe_busy | 1.759431 | 1.639869 |
| 20 | stall_other | 47.702475 | 46.638134 |
| 21 | gld_efficiency | 82.926829 | 82.926829 |
| 22 | gst_efficiency | 100.000000 | 100.000000 |

IPC

| 指標 | 改善前 | 改善後 |
|---|---|---|
| **IPC** | ~2.00 | **2.42** |

Pipeline被塞得更滿，因為global mem的存取相依被減緩了，可以花更多時間在處理指令

**Memory ThroughputMemory Stalls**

| 指標 | 改善前 | 改善後 |
|---|---|---|
| **gld_throughput** | ~749 GB/s | **914 GB/s** |
| **l2_read_throughput** | ~749 GB/s | **914 GB/s** |

因為memory store的stall大幅下降，因為不用重複一直讀同一個D[i][j]，把整體資料流順暢度都提升了

**Memory Stalls**

| Stall Type | 改善前 | 改善後 | 意義 |
|---|---|---|---|
| **stall_memory_dependency** | **~76%** | **~9.8%** | 核心瓶頸被解鎖 |
| **stall_exec_dependency** | **~12%** | **~34%** | 代表瓶頸轉向 **compute chain**，而非 **memory** |
| **stall_other** | **~8%** | **~47%** | 代表 **pipeline** 正在忙著處理而不是等待 |

因為memory store的stall大幅下降，因為不用重複一直讀同一個D[i][j]，把整體資料流順暢度都提升了

**Issue Slot Utilization**

| 指標 | 前 | 後 |
|---|---|---|
| **Issue Slot Utilization** | ~44% | **50%** |

原本

```
for (int k = 0; k < B; k++) {
    int oldVal = Dist[i * nPad + j];
    int via = Dist[i * nPad + k] + Dist[k * nPad + j];
    Dist[i * nPad + j] = min(oldVal, via);
}
```

改善後

```
int dij = Dist[i * nPad + j];
for (int k = 0; k < B; k++) {
     int via = Dist[i * nPad + k] + Dist[k * nPad + j];
     dij = min(dij, via);
}
Dist[i * nPad + j] = dij;
```

---

1 thread process 4 elements (Time = 8.59)

| | metric | avg | total |
|---|---|---|---|
| 1 | shared_ld_bank_conflict | 0.000000 | 0.000000 |
| 2 | shared_st_bank_conflict | 0.000000 | 0.000000 |
| 4 | achieved_occupancy | 0.907009 | 0.892508 |
| 5 | ipc | 2.241561 | 2.094174 |
| 6 | gld_throughput | 764.363607 | 725.662673 |
| 7 | gst_throughput | 58.797201 | 55.820206 |
| 8 | l2_read_throughput | 764.414469 | 725.771319 |
| 9 | l2_write_throughput | 58.807532 | 55.830014 |
| 10 | shared_efficiency | 0.000000 | 0.000000 |
| 11 | warp_execution_efficiency | 100.000000 | 99.912886 |
| 12 | issue_slot_utilization | 47.938021 | 44.767135 |
| 13 | stall_inst_fetch | 11.292953 | 5.837986 |
| 14 | stall_exec_dependency | 12.484531 | 10.915533 |
| 15 | stall_memory_dependency | 75.227984 | 71.149725 |
| 16 | stall_constant_memory_dependency | 1.378557 | 0.954595 |
| 17 | stall_texture | 4.766012 | 1.876801 |
| 18 | stall_sync | 0.000000 | 0.000000 |
| 19 | stall_pipe_busy | 1.429276 | 1.276730 |
| 20 | stall_other | 18.058510 | 4.748315 |
| 21 | gld_efficiency | 38.461538 | 38.461538 |
| 22 | gst_efficiency | 50.000000 | 50.000000 |

**Global Load Throughput**

| Metric | 改善前 | 改善後 |
|---|---|---|
| **gld_throughput** | **~915 GB/s** | **~764 GB/s** |

我不在需要這麼多次load因為我重複使用了周圍的4 elements的資料

**Achieved Occupancy**

| Metric | 改善前 | 改善後 |
|---|---|---|
| **achieved_occupancy** | 0.8378 | **0.9070** |

因為每個 warp 一次處理更多資料，計算佔比提高、記憶體 I/O 等待時間降低，因此 warp 不需要頻繁切換，讓更多 warp 能持續處於可執行狀態，進而提升 Occupancy、提高 SM 資源利用率，讓 GPU 整體更忙、更有效率。

---

Shared memory (Time = 6.48)

| | metric | avg | total |
|---|---|---|---|
| 1 | shared_ld_bank_conflict | 0.000000 | 0.000000 |
| 2 | shared_st_bank_conflict | 18496.000000 | 332928.000000 |
| 4 | achieved_occupancy | 0.855130 | 0.843596 |
| 5 | ipc | 2.733383 | 2.588445 |
| 6 | gld_throughput | 307.076223 | 253.186303 |
| 7 | gst_throughput | 87.736064 | 72.338944 |
| 8 | l2_read_throughput | 307.152120 | 253.512224 |
| 9 | l2_write_throughput | 87.751480 | 72.351655 |
| 10 | shared_efficiency | 39.285714 | 39.285714 |
| 11 | warp_execution_efficiency | 100.000000 | 99.911024 |
| 12 | issue_slot_utilization | 58.846432 | 55.717714 |
| 13 | stall_inst_fetch | 10.166955 | 5.323307 |
| 14 | stall_exec_dependency | 17.877136 | 16.969814 |
| 15 | stall_memory_dependency | 37.855935 | 27.369052 |
| 16 | stall_constant_memory_dependency | 1.489854 | 1.185965 |
| 17 | stall_texture | 0.127647 | 0.094031 |
| 18 | stall_sync | 4.915293 | 3.797776 |
| 19 | stall_pipe_busy | 4.500625 | 4.163025 |
| 20 | stall_other | 38.840516 | 36.188478 |
| 21 | gld_efficiency | 42.857143 | 42.857143 |
| 22 | gst_efficiency | 50.000000 | 50.000000 |

將Piovt roe Pivot col存入shared mem因為這部分會一直重複用到所以搬到存取速度快的shared mem

**Global Memory**

| 指標 | 改善前 | 改善後（使用 shared memory） |
|---|---|---|
| **gld_throughput** | ~915 GB/s | **~307 GB/s** |
| **gst_throughput** | ~22 GB/s | **~87.7 GB/s** |
| **l2_read_throughput** | ~915 GB/s | **~307 GB/s** |
| **gld_efficiency** | 82.9% | **42.9%** |

明顯看到IO很大一部分因為shared mem加入而被節省，運行過程更少去觸碰到global mem load所以throught put 大幅下降

**IPC**

| 指標 | 改善前 | 改善後 | 改善重點 |
|---|---|---|---|
| **IPC** | ~2.24 | **~2.73** | ALU 更忙、pipeline 更飽 |

IPC 上升代表 memory latency 被有效隱藏，整題運行比重更靠向compute而不是IO

**Memory Stall**

| 指標 | 改善前 | 改善後 | 改善重點 |
|---|---|---|---|
| **stall_memory_dependency** | ~75% | **~38%** | Memory 等待時間明顯下降 |
| **stall_exec_dependency** | ~12% | ~18% | 變成 compute dependency 為主 |
| **stall_pipe_busy** | ~1.7% | ~4.5% | Pipeline 使用率提高 |

pipeline因為IO在hared mem的部分較快所以更少stall發生，整體pipeline運行更加順暢

**Issue Slot Utilization**

| 指標 | 改善前 | 改善後 | 改善重點 |
|---|---|---|---|
| **issue_slot_utilization** | ~47.9% | **~58.8%** | 指令發射更頻繁、空洞更少 |

---

ushrot + no bankconflict (Time = 2.12)

| | metric | avg | total |
|---|---|---|---|
| 1 | shared_ld_bank_conflict | 0.000000 | 0.000000 |
| 2 | shared_st_bank_conflict | 0.000000 | 0.000000 |
| 4 | achieved_occupancy | 0.854881 | 0.832872 |
| 5 | ipc | 2.443041 | 2.402182 |
| 6 | gld_throughput | 108.772834 | 108.475473 |
| 7 | gst_throughput | 54.386417 | 54.237737 |
| 8 | l2_read_throughput | 110.169848 | 108.911711 |
| 9 | l2_write_throughput | 54.407993 | 54.259254 |
| 10 | shared_efficiency | 70.000000 | 70.000000 |
| 11 | warp_execution_efficiency | 100.000000 | 100.000000 |
| 12 | issue_slot_utilization | 50.402767 | 49.560412 |
| 13 | stall_inst_fetch | 2.810940 | 2.165510 |
| 14 | stall_exec_dependency | 35.785947 | 34.887681 |
| 15 | stall_memory_dependency | 1.014400 | 0.860602 |
| 16 | stall_constant_memory_dependency | 2.836438 | 2.296423 |
| 17 | stall_texture | 0.171754 | 0.135151 |
| 18 | stall_sync | 15.460733 | 12.692194 |
| 19 | stall_pipe_busy | 1.598041 | 1.195300 |
| 20 | stall_other | 44.042076 | 42.068448 |
| 21 | gld_efficiency | 75.000000 | 75.000000 |
| 22 | gst_efficiency | 50.000000 | 50.000000 |
| 23 | sm_efficiency | 70.693518 | 69.905423 |
| 24 | shared_load_throughput | 2610.548006 | 2603.411356 |
| 25 | shared_store_throughput | 108.772834 | 108.475473 |

### Bank conflict

此版本把原本的bank conflict全部消除，進一步提升IO效率

### Global memory

| 指標 | 改進前 | 最終版本 |
|---|---|---|
| **gld_throughput** | ~307–900+ GB/s | **108 GB/s** |
| **l2_read_throughput** | ~300–900+ GB/s | **110 GB/s** |
| **gld_efficiency** | ~42% | **75%** |
| **gst_throughput** | ~87 GB/s | **54 GB/s** |

因為ushort讓每一筆資料大小縮短，所以整體IO需要傳輸的量也變小造成IO速度變快

**Shared Memory**

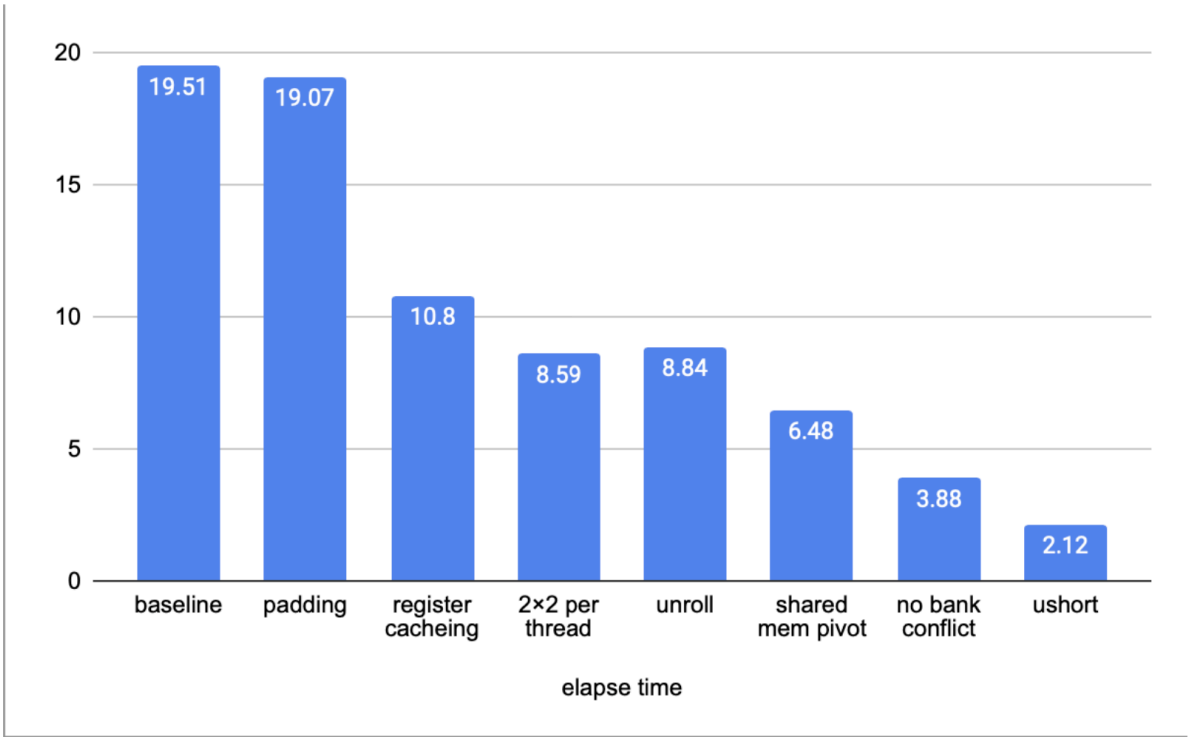
| 指標 | 改進前 | 最終版本 |
| --- | --- | --- |
| **shared_efficiency** | ~39% | **70%** |

Shared memory的使用率大幅提升是因為bank conflict被完全解除

**Stall**

| 指標 | 改進前 | 最終版本 |
| --- | --- | --- |
| **stall_memory_dependency** | ~38–75% | **1%** |
| **stall_exec_dependency** | ~18% | **36%** |

幾乎沒有因為memory而暫停，整體pipeline運行進一步提升



## 3.d Weak Scalability (HW3-3)

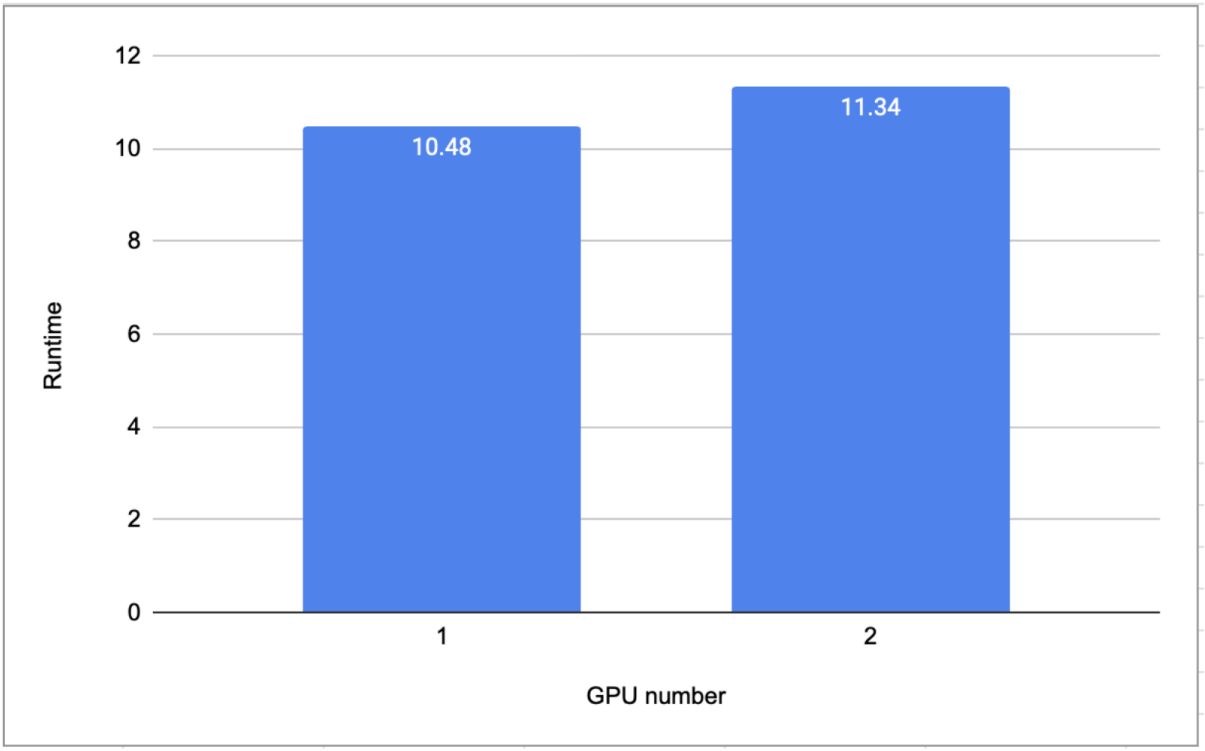
因為FW的運算複雜度大概是2 ^ (1/3) 所以選擇

testcase(23) n=22000 ， testcase(28) n =27000

盡量讓計算量是兩倍的提升

Profile instrustion

```
srun -p nvidia -N1 -n1 --gres=gpu:1 nvprof --print-gpu-summary
./hw3-3 ./testcases/p23k1 ./out
srun -p nvidia -N1 -n1 --gres=gpu:2 nvprof --print-gpu-summary
./hw3-3 ./testcases/p28k1 ./out
```



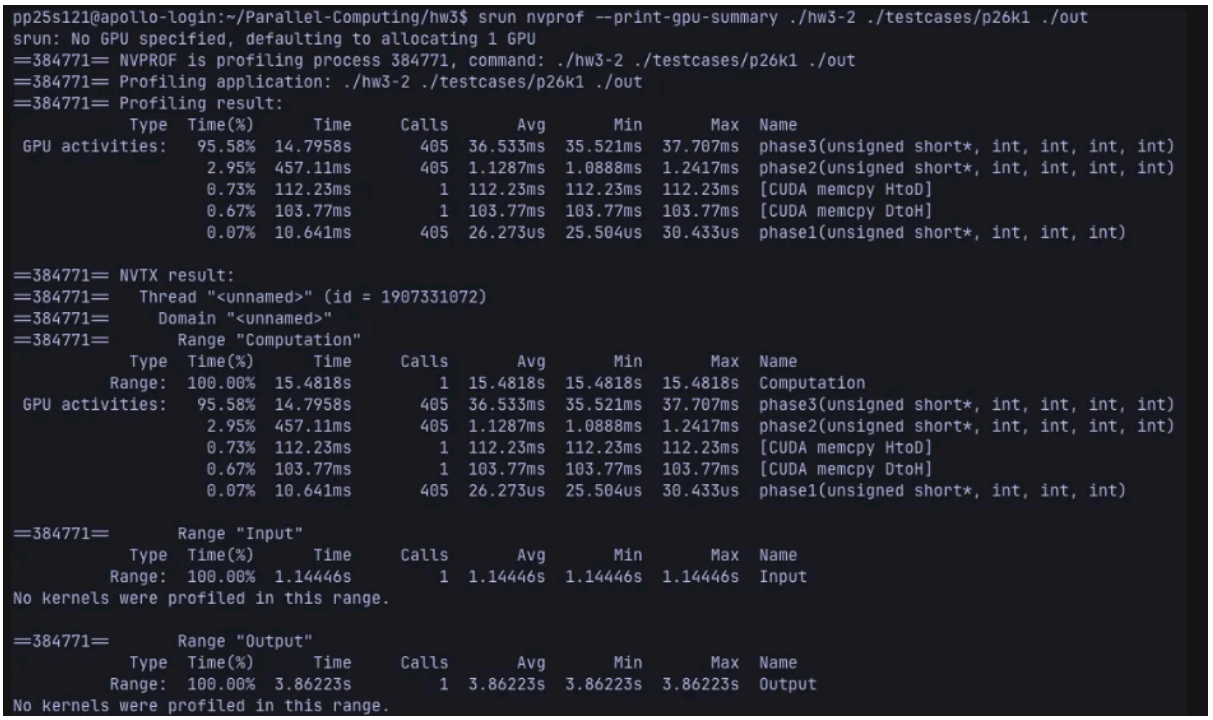

只有因為通訊overhead有些微的時間增加，scalability十分不錯因為不會有大量的資料在gpu之間傳輸，只傳當下任務的所必需使用的資料區段，所以通訊埠沒有造成明顯的效能損失

---

# 3.e Time Distribution (HW3-2)

Profile instrustion

```
srun nvprof --print-gpu-summary ./hw3-2 ./testcases/p26k1 ./out
```



```
pp25s121@apollo-login:~/Parallel-Computing/hw3$ srun nvprof --print-gpu-summary ./hw3-2 ./testcases/p26k1 ./out
srun: No GPU specified, defaulting to allocating 1 GPU
==384771== NVPROF is profiling process 384771, command: ./hw3-2 ./testcases/p26k1 ./out
==384771== Profiling application: ./hw3-2 ./testcases/p26k1 ./out
==384771== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   95.58%  14.7958s       405  36.533ms  35.521ms  37.707ms  phase3(unsigned short*, int, int, int, int)
                    2.95%  457.11ms       405  1.1287ms  1.0888ms  1.2417ms  phase2(unsigned short*, int, int, int, int)
                    0.73%  112.23ms         1  112.23ms  112.23ms  112.23ms  [CUDA memcpy HtoD]
                    0.67%  103.77ms         1  103.77ms  103.77ms  103.77ms  [CUDA memcpy DtoH]
                    0.07%  10.641ms       405  26.273us  25.504us  30.433us  phase1(unsigned short*, int, int, int)

==384771== NVTX result:
==384771==   Thread "<unnamed>" (id = 1907331072)
==384771==     Domain "<unnamed>"
==384771==       Range "Computation"
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
          Range:  100.00%  15.4818s         1  15.4818s  15.4818s  15.4818s  Computation
 GPU activities:   95.58%  14.7958s       405  36.533ms  35.521ms  37.707ms  phase3(unsigned short*, int, int, int, int)
                    2.95%  457.11ms       405  1.1287ms  1.0888ms  1.2417ms  phase2(unsigned short*, int, int, int, int)
                    0.73%  112.23ms         1  112.23ms  112.23ms  112.23ms  [CUDA memcpy HtoD]
                    0.67%  103.77ms         1  103.77ms  103.77ms  103.77ms  [CUDA memcpy DtoH]
                    0.07%  10.641ms       405  26.273us  25.504us  30.433us  phase1(unsigned short*, int, int, int)

==384771==       Range "Input"
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
          Range:  100.00%  1.14446s         1  1.14446s  1.14446s  1.14446s  Input
No kernels were profiled in this range.

==384771==       Range "Output"
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
          Range:  100.00%  3.86223s         1  3.86223s  3.86223s  3.86223s  Output
No kernels were profiled in this range.
```

因為是單節點單卡所以並沒有communication的部分

| 項目 | 秒數 |
| --- | --- |
| Computing (kernel 1 + 2 + 3) | 15.4818 s |
| Memory Copy H2D | 112.23 ms |
| Memory Copy D2H | 103.77 ms |
| Input | 1.14446 s |
| Output | 3.86223 s |

整體程式執行狀態已經變成compute bound，尤其是phase3佔了很大一部分的GPU time，這也證明前幾個版本對memory的優化有奏效

---

# AMD GPU Porting and Analysis

**Profile instruction**

```
srun -p amd -N1 -n1 \
rocprofv2 --basenames -i ./profile-amd/metrics.txt -o
profile-amd-p2
0k1 -d ./profile-amd/profile-result \
./hw3-2-amd ./testcases-amd/p20k1 ./p20k1.out
```

## a. Implementation

cuda 和 Hip其實功能非常相近，只有一些function名稱不同

| CUDA | HIP |
| --- | --- |
| cudaMalloc | hipMalloc |
| cudaFree | hipFree |
| cudaMemcpy | hipMemcpy |
| cudaMemcpyHostToDevice | hipMemcpyHostToDevice |
| cudaMemcpyDeviceToHost | hipMemcpyDeviceToHost |
| cudaDeviceSynchronize | hipDeviceSynchronize |
| cudaMallocHost | hipHostMalloc |
| cudaFreeHost | hipHostFree |

因為kernel內部結構都不變，基本上就是把function名稱改完就能正常運行

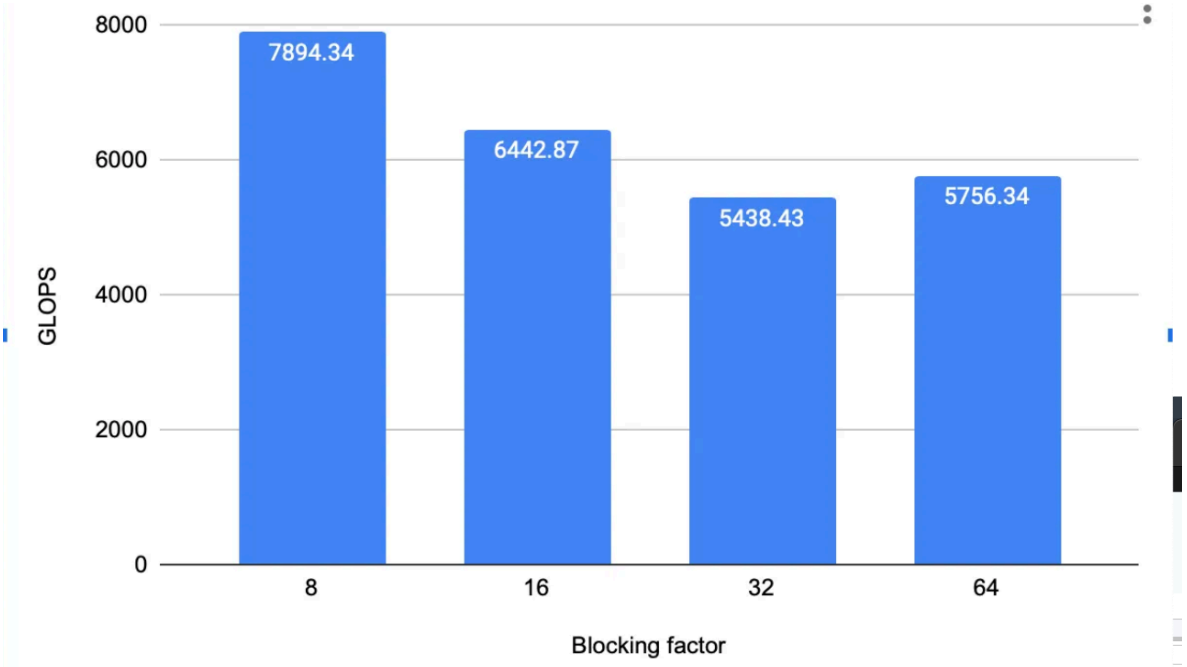
---

## b. Experiment & Analyze

### Blocking Factor on AMD (MI210)

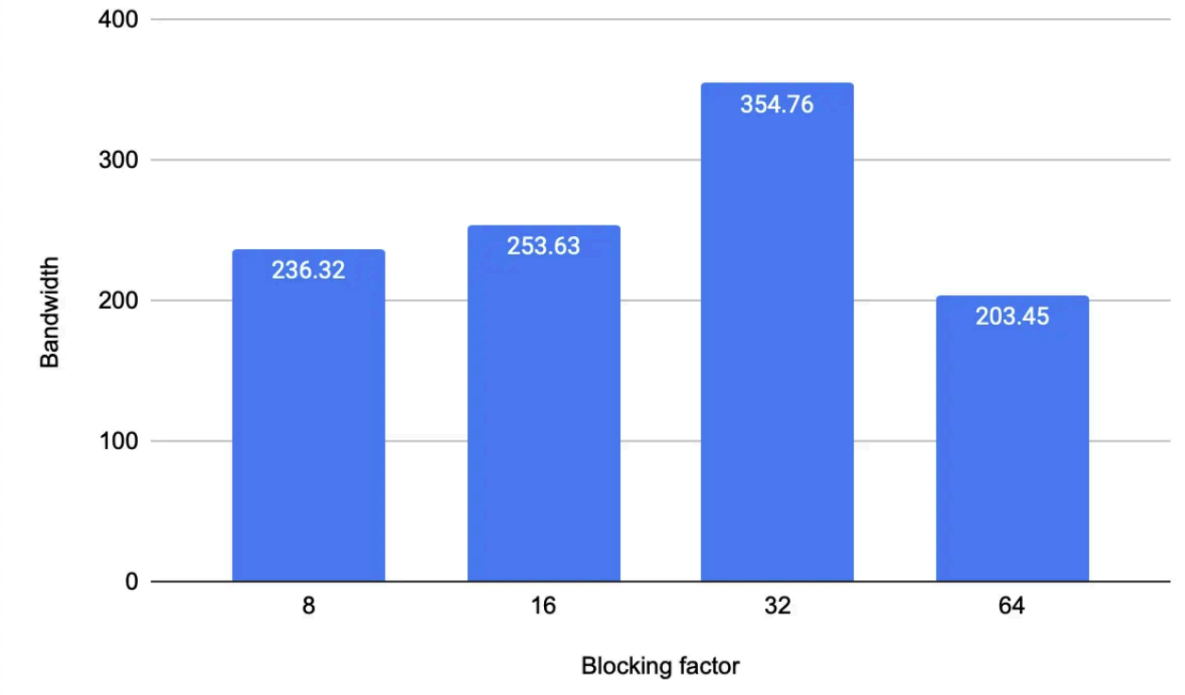
B = 32 最佳

GLOP



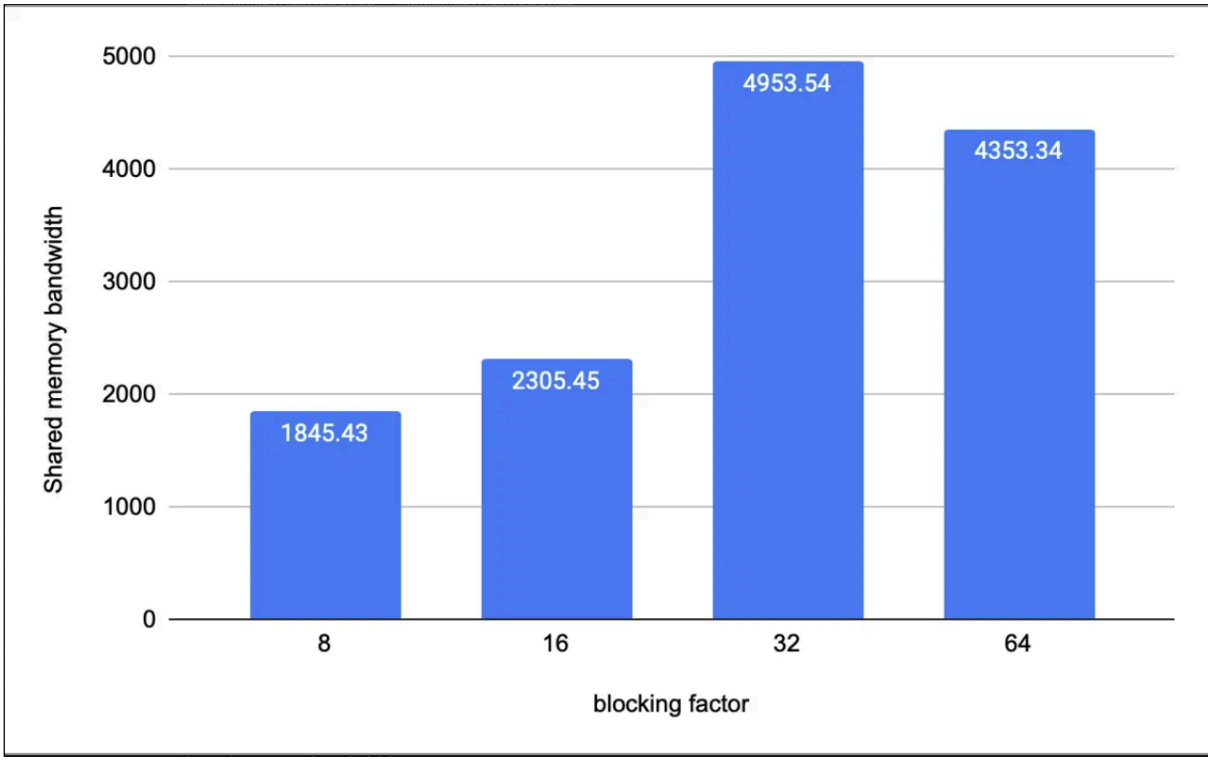

隨著 B 從 8→16→32 持續上升，在 B=32 時達到最高；這代表在 MI210 上 32×32 的 tile 已經能提供足夠的資料重複使用與並行度，讓大量整數 min/add 運算把 ALU 填滿，而 B=64 雖然 tile 更大，但因為每個 block 需要更多 LDS 空間與暫存器，降低了每個 CU 可同時常駐的 wave/block 數，導致整體 GOPS 反而掉下來，說明此時受到 occupancy 限制。

Global memory performance



B=8/16 由於 tile 太小，對 HBM2e 的存取次數多、重用度差，雖然 MI210 有高達 1.6 TB/s 的理論頻寬，實際量測卻無法有效吃滿；B=32 時 global BW 提升到接近曲線峰值，同時因為大部分中間資料都能在 LDS 內重用，單位運算所需的 global traffic 也最低，形成「GOPS 高 + global BW 利用率高但不浪費」的最佳平衡；B=64 雖然 global BW 不會明顯變差，但前面提到的 occupancy 與 LDS 壓力讓整體效能（GOPS）已經開始下滑。

Shared memory performance




B=8 與 16 的 shared BW 只有約 1.8k 與 2.3k（任意單位），而 B=32 直接跳到約 4953，接近前兩者的兩倍，B=64 則稍微下降到約 4353。這表示在 MI210 每個 CU 只有 64 KB LDS 的硬體條件下，B=32 剛好讓 tile 足夠大，可以在 LDS 裡做高效率的資料重用與平行載入，最大化 LDS 管線吞吐；B=64 時 LDS 使用量過大、active blocks 變少，甚至更容易產生 bank conflict，導致 shared BW 雖然仍高，但已低於 B=32 的峰值。

## Compare NVIDIA vs AMD

| 概念 | NVIDIA (GTX 1080, CUDA) | AMD (MI210, ROCm/HIP) |
|---|---|---|
| thread | thread | work-item / thread |
| block | block / CTA | work-group |
| warp 單位 | **warp = 32 threads** | **wavefront = 64 threads** |
| SM / CU | SM (Streaming Multiprocessor) | CU (Compute Unit) |

整體再將code移植過來的過程中可以看到profile結果都顯示，只是API名稱是無法取得好的效能，因為這兩張卡不只我上述提到warp中threads數不同，頻寬以及各階層記憶體大小都不同，我直接拿針對GTX 1080有優勢的code來改，其實沒辦法完全發揮amd卡，但由於時間的關係也只有做warp threads mapping部分的優化，課程結束後會再花更多時間把amd卡的效能也tune到完全發揮出來

## AMD-specific Optimization

### Wavefront 64–Aligned Thread Mapping

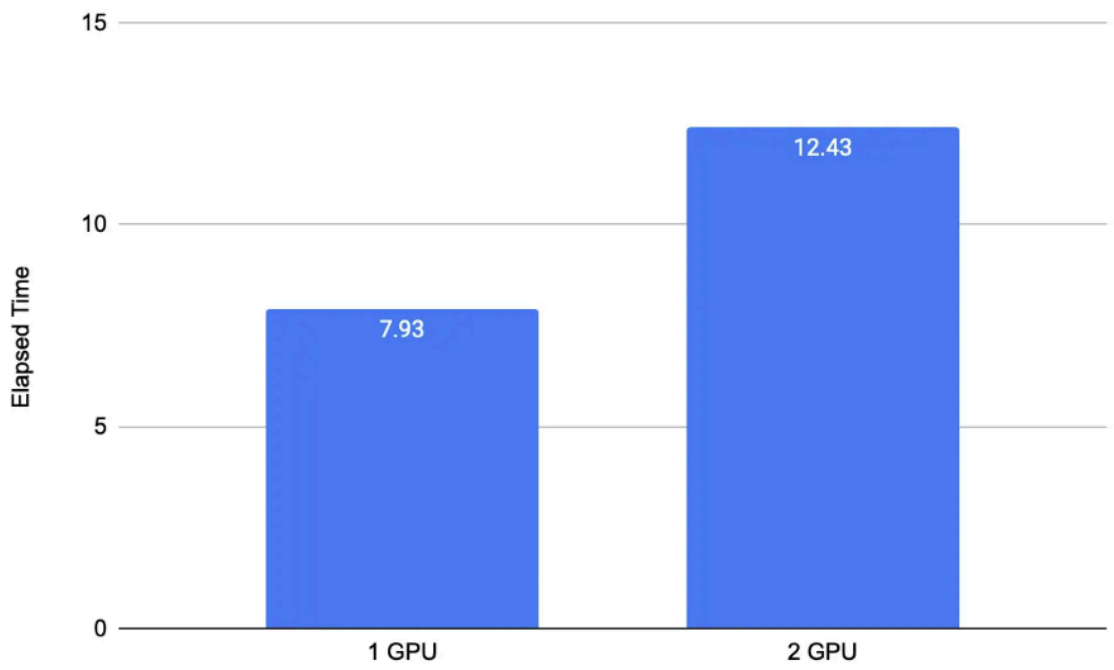
在 AMD GPU 上，一個 Wavefront 由 **64** 個 **threads** 組成。因此，我將原本採用 NVIDIA warp 對齊的 blockDim(32, 32)，重新設計為 blockDim(64, 16)，讓 **X** 維度正好對齊 **64 threads**。

改善：

- **Global memory access** 完整 **coalescing**
  64 個 threads 讀寫連續位址使記憶體頻寬利用率提高
- **Shared memory bank conflict** 大幅降低
  不同 threads 打到不同 bank，符合 AMD LDS 結構
- **Wavefront** 不會出現閒置或浪費
  不像原本 warp=32 的 mapping 會導致硬體使用不均

改善適合AMD Gpu maaping的寫法後，occupancy就明顯有上升，證明原本的寫法會讓一部分的threads是空轉的狀態

## AMD Weak Scaling




圖表顯示我的weak scalability不太好，因為我在改寫amd版本的code時將資料整行整列的都搬運到兩張GPU上，所以有很多多餘的資料傳遞

並且我沒有針對amd gpu有的硬體做到完全的優化，所以有很多運算資源是閒置的，導致資料傳輸pipeline可能受到延遲的影響間接影響道gpu之間的通訊