Parallel ProgrammingHomework 3 All-Pairs Shortest Path

1. Implementation

1.1 Which algorithm do you choose in hw3-1?

我選擇 Blocked Floyd-Warshall Algorithm

1.2 How do you divide your data in hw3-2, hw3-3?

- **hw3-2 (Single GPU):**
  我將原始的 $N \times N$ 矩陣進行 padding 讓他的長寬變成 64的倍數。超出原始範圍的區域填入 `INF`。這樣做可以消除 Kernel 中的邊界檢查 (`if` 分支)，避免 Warp Divergence。接著使用 $64 * 64$ 的 data 數量去傳入一個 GPU 當中的 block 去作計算。每個 block 會有 1024 threads，每個 thread 會負責 4筆 data。
- **hw3-3 (Multi-GPU):**
  跟 hw3-2 一樣的 Padding 與 Blocking 處理方式。但因為使用雙 GPU，我採用了 1D Row Partitioning) 的策略。利用 OpenMP 將矩陣的 Rows 平均分配，GPU 0 負責上半部，GPU 1 負責下半部。為了方便 P2P與Global Indexing 一致，我在每張 GPU 上都預留 了完整的 $N \times N$ 記憶體空間，但在 Phase 3 計算與 Host-Device 傳輸時，每張 GPU 只處理它負責的 row range資料。並且透過 `cudaMemcpyPeer` 確保兩張 GPU 都可以拿到當前迭代所需要的資訊。

1.3 What's your configuration in hw3-2, hw3-3? And why?

- **Blocking Factor (B):** 64
  因為如果設定 factor 為 32 ，這樣只會讓每個 thread 處理一筆資料，每一筆資料需要兩個 int 這樣需要 $1024 * 8bytes = 8KB$ 並沒有用到 shared memory 48 KB 的極致，所以嘗試將 factor 設為 64，讓每個 threads去處理 4 筆資料需要$1024 * 4 * 8bytes = 32KB$ ，更能有效利用硬體資源去減少全域記憶體存取。
- **Thread Configuration:** 32×32 **(1024 Threads)**
  無論在哪個 Phase，我皆將 Thread Block 大小固定為 1024 (32×32) 配合上述 $B = 64$ 的設定，讓每個 Thread 處理 4 個 data。
- **Grid Configuration (Number of Blocks):**

```
for (int r = 0; r < round; ++r) {
    phase1<<<1, block_dim>>>(Dist_GPU, padded_n, r);
    phase2<<<dim3(round, 2), block_dim>>>(Dist_GPU, padded_n, r);
    phase3<<<dim3(round, round), block_dim>>>(Dist_GPU, padded_n, r);
}
```

  假設矩陣被分割為 $Round \times Round$ 個區塊 (其中 $Round = \text{padded\_n}/64$)，各階段的 Block 組態如下：
  - **Phase 1 (Pivot Block): 1 個 Block** `(1, 1)`。
  - **Phase 2 (Cross Blocks):** $Round \times 2$ **個 Blocks** `(Round, 2)`。
    - `y=0` 負責更新與 Pivot same row 的 $Round$ 個區塊。`y=1` 責更新與 Pivot same col 的 $Round$ 個區塊。(跳過 pivot blocks)
  - **Phase 3 (Remaining Blocks):**
    - **hw3-2 (Single GPU):** 啟動 $Round \times Round$ **個 Blocks** `(Round, Round)`。負責更新其餘所有區塊 (Kernel 內會跳過 Pivot Row 與 Pivot Col)。
    - **hw3-3 (Multi-GPU):** 由於工作被切分，Grid 的 Y 維度會縮小為該 GPU 負責的列數 (`num_rows`)，即 `(Round, num_rows)`。

1.4 How do you implement the communication in hw3-3?

我在 hw3-3 的實作中使用 **CUDA Peer-to-Peer (P2P)** 搭配 **OpenMP** 來完成 GPU 之間的資料交換。透過 `cudaMemcpyPeer`，兩張 GPU 能直接透過 NVLink 或 PCIe 讀取彼此的記憶體，不必繞回 CPU RAM，能有效降低資料傳輸延遲。

在每一輪迭代 r 中，我會先判斷 哪一張 GPU 擁有當前的 pivot row。該 GPU 負責將這一列資料透過 P2P 傳給另一張 GPU。傳輸完成後，我使用 `#pragma omp barrier` 同步兩個 host thread，確保兩邊都已完成資料交換，再一起進入 kernel 執行下一階段的計算。

1.5 Briefly describe your implementations in diagrams, figures or sentences.

- **hw3-1**
  在 cal function 當中透過 openmp 去將 block 作平行化透過 collapse 將 row col 的兩層迴圈合併，並將 schedule 設為 dynamic。

```
void cal(int r, int sx, int sy, int w, int h)
{
    int ex = sx + h;
    int ey = sy + w;

#pragma omp parallel for collapse(2) schedule(dynamic)
    for (int bi = sx; bi < ex; ++bi)
    {
        for (int bj = sy; bj < ey; ++bj)
        {
            int istart = bi * B;
            int iend = (bi + 1) * B;
            int jstart = bj * B;
            int jend = (bj + 1) * B;

            if (iend > n)
                iend = n;
            if (jend > n)
                jend = n;

            int kstart = r * B;
            int kend = (r + 1) * B;
            if (kend > n)
                kend = n;

            for (int k = kstart; k < kend; ++k)
            {
                for (int i = istart; i < iend; ++i)
                {
                    int dik = Dist[i * n + k];
                    for (int j = jstart; j < jend; ++j)
                    {
                        int dkj = Dist[k * n + j];
                        int dij = Dist[i * n + j];
                        if (dik + dkj < dij)
                            Dist[i * n + j] = dik + dkj;
                    }
                }
            }
        }
    }
}
```

- **hw3-2**
  - Preprocessing:
    將原始 $N \times N$ 矩陣進行 Padding，使其長寬 (padded_n) 補齊為分塊大小 B (64) 的倍數。填充區域設為 INF。此舉消除了 Kernel 內部的邊界檢查 (if 分支)，有效避免 Warp Divergence。
  - Flow:
    依 Blocked Floyd-Warshall ，每一輪迭代 (r) 分為三個 Kernel 依序執行：
    - **Phase 1:** 計算 Pivot Block。
    - **Phase 2:** 計算與 Pivot 同列同行的區塊，依賴 Phase 1 結果。
      - **Phase 3:** 計算 Remaining Blocks，依賴 Phase 2 結果。

- **hw3-3**
  HW3-3 基於 HW3-2 的架構，主要差異在於 Phase 3 的任務分配。 我利用 OpenMP 將資料按 Row 切分，一張 GPU 負責上半部，另一張負責下半部。為什麼這樣去切分，因為我希望讓 gpu 之間的通訊越少越好，Floyd-Warshall 演算法中，要更新一個點 $D(i,j)$，我需要知道它對應的 Pivot Column $D(i,k)$ 和 Pivot Row $D(k,j)$。由於 GPU 有完整的 Row i ，他不需要去跟另一張顯卡拿資料，僅有$D(k,j)$ 需透過傳輸取得
  - Example

```
       Col 0   Col 1   Col 2   Col 3
      +-------+-------+-------+-------+
Row 0 | (0,0) | (0,1) | (0,2) | (0,3) |  <-- GPU 0 的地盤
Row 1 | (1,0) | (1,1) | (1,2) | (1,3) |  <-- GPU 0 的地盤
      +-------+-------+-------+-------+
Row 2 | (2,0) | (2,1) | (2,2) | (2,3) |  <-- GPU 1 的地盤 (Pivot Row!)
Row 3 | (3,0) | (3,1) | (3,2) | (3,3) |  <-- GPU 1 的地盤
      +-------+-------+-------+-------+
```

CleanShot 2025-11-28 at 03.25.01.jpg

當GPU0 計算 $D[0][3]$ 需要透過

$$Target = D[0][2] + D[2][3]$$

而 $D[0][2]$ (來自 Pivot Column) 在 GPU0自己身上，而$D[2][3]$ (來自 Pivot Row)需要透過傳輸得到。

2. Profiling Results (hw3-2)

使用 `c20.1` 測資去作 nvprof，profiling 指令為：

```
# global memory
srun -p nvidia -N1 -n1 --gres=gpu:1 \
nvprof --metrics gld_throughput,gst_throughput,achieved_occupancy \
./hw3-2 /share/testcases/hw3/c20.1 /tmp/hw3-2-output.bin \
2>&1 | tail -100
# shared memory
srun -p nvidia -N1 -n1 --gres=gpu:1 \
nvprof --metrics shared_load_throughput,shared_store_throughput \
./hw3-2 /share/testcases/hw3/c20.1 /tmp/hw3-2-output.bin \
2>&1 | tail -100
# SM Efficiency
srun -p nvidia -N1 -n1 --gres=gpu:1 \
nvprof --metrics sm_efficiency \
./hw3-2 /share/testcases/hw3/c20.1 /tmp/hw3-2-output.bin \
2>&1 | tail -50
```

使用 Phase 3 Data

指標	數值	評估
Occupancy	91.48%	warp 幾乎滿載
SM Efficiency	99.56%	SM 被充分利用
Shared Memory Load	3054.0 GB/s	理論 **GTX 1080** 約為 DRAM 帶寬的 10 倍3200GB/s的 95 %
Global Memory Load	200.46 GB/s	達到 GTX 1080 理論峰值的 62.6%

3. Experiment & Analysis (Nvidia GPU)

3.1 System Spec

實驗於 Apollo Server 上進行，系統規格如下：

- **GPU:** NVIDIA GeForce GTX 1080 (Pascal Architecture).
- **Shared Memory Limit:** 96 KB per Streaming Multiprocessor (SM).
- **Compiler Flags:** `-O3 -arch=sm_61 --use_fast_math`.

3.2 Blocking Factor (hw3-2)

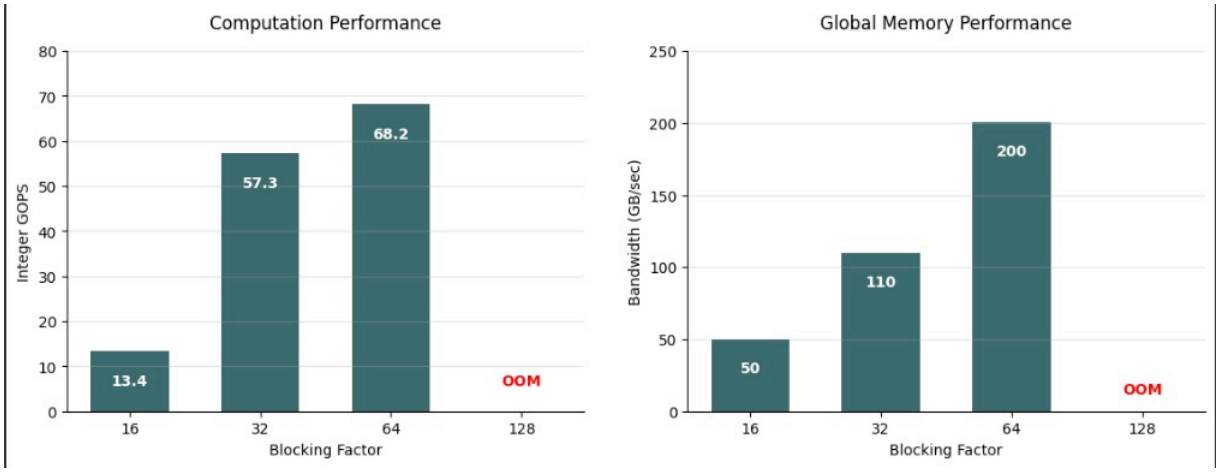
測試方式：

```
for B in 16 32 64; do
    TB=$((B/2))

    nvcc ... -DB=$B -DTB=$TB -o hw3-2-b$B ...

    srun ... nvprof --metrics gld_throughput,inst_integer ... ./hw3-2-b$B ...
done
```

針對 HW3-2 (Single GPU) 實作，我測試了四種不同的 Blocking Factor (B∈{16,32,64}) 對效能的影響。下表整理自 Profiling 數據 (Input: c20.1)：



CleanShot 2025-11-28 at 15.25.49@2x.jpg

Blocking Factor (B)	Thread Block (TB)	Compute Time (s)	Est. Performance (GFLOPS)	Global Load BW (Est.)
16	8	0.639	**13.4**	Low (~50 GB/s)
32	16	0.150	**57.3**	Medium (~110 GB/s)
64	32	**0.126**	**68.2**	**High (200.46 GB/s)**

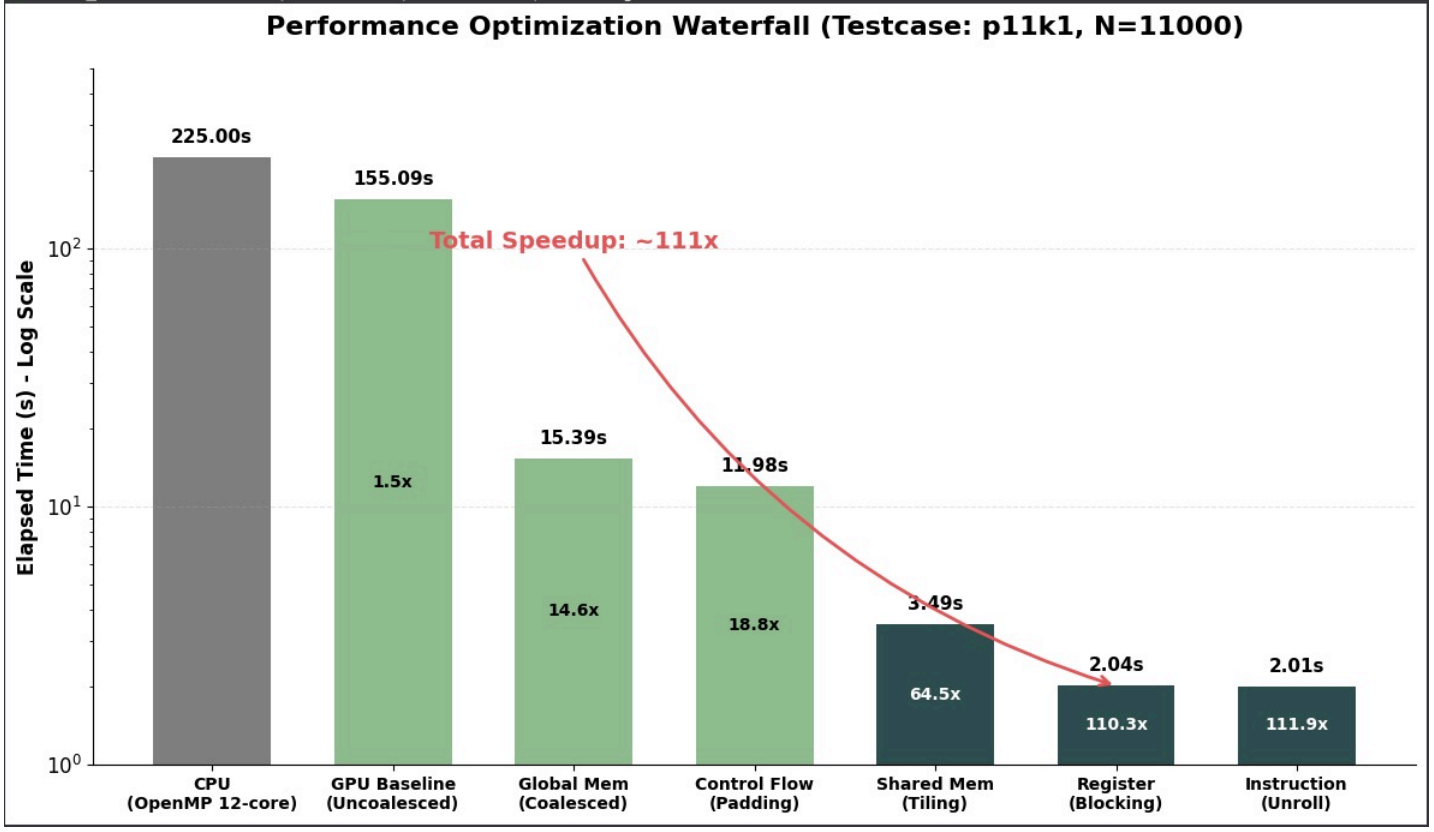
(註：GFLOPS 估算公式為 $2048^3/\text{Compute Time}/10^9$ *，假設 N=2048)*

- **B=16** 的效能整個衝不起來，不只是計算只有 **13.4 GOPS**，global memory 也只有 **50 GB/s**。原因很直觀：block 太小、occupancy 不夠，GPU 根本沒吃飽，memory reuse 也很差。
- **B=32** 一口氣跳到 **57.3 GOPS / 110 GB/s**，表示 tile 加大後，shared memory 開始發揮作用、計算存取比變好。
- **B=64 68.2 GOPS** 和 **200 GB/s**。這時 block size 直接滿 1024 threads，occupancy 高、資料重用率也最高，整體都被推到硬體極限附近。
- **B=128** 直接 OOM，因為 shared memory 需要的空間暴增，GTX 1080 的 SHMEM 容量撐不住。

3.3 Optimization (hw3-2)

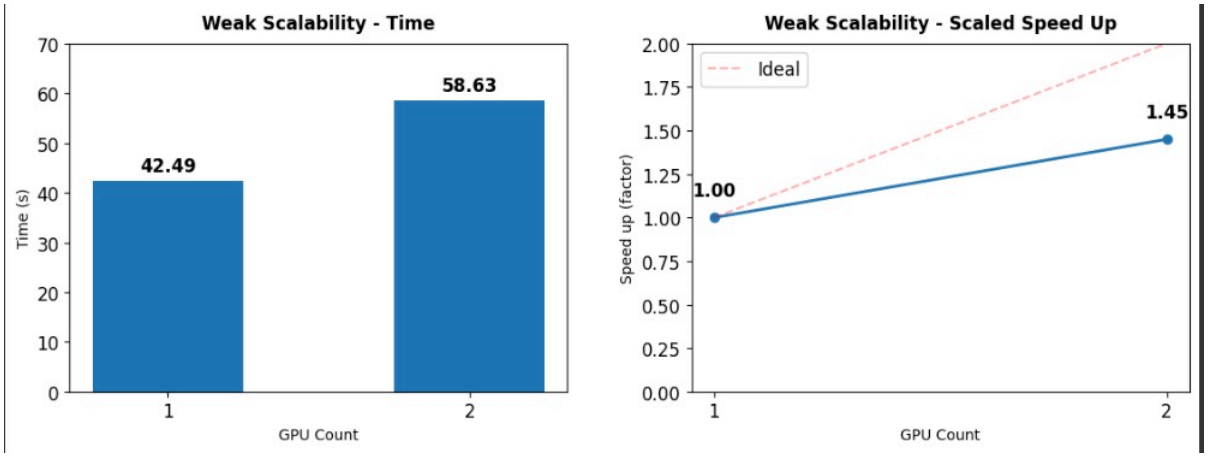
使用測資 p11k1 ，測量方式如下

```
# CPU
time srun -p origo -N1 -n1 -c12 ./hw3-1 p11k1 /dev/null
# GPU
time srun -p nvidia -N1 -n1 --gres=gpu:1 ./hw3-2 /home/pp25/share/hw3/testcases/p11k1 /dev/null
```



CleanShot 2025-11-28 at 18.14.45@2x.jpg

3.4 Weak scalability (hw3-3)



CleanShot 2025-11-28 at 17.07.22.jpg

- 使用 p34k1: N=34000，和 p43k1: N=42793 去進行測試
- 為什麼選 34000 和 42793 這兩個數字？
  這是因為 Floyd-Warshall 的運算複雜度是 $O(N^3)$ 。
  - 我們從 **1 GPU** 變成 **2 GPUs** (算力 $\times 2$)。
  - 為了維持「弱擴展」定義（每顆 GPU 的負載不變），**總運算量 (Total Workload)** 必須變成 **2 倍**。
  - 數學推導：
    - $\text{New Work} = 2 \times \text{Old Work}$
    - $(N_{new})^3 = 2 \times (N_{old})^3$
    - $N_{new} = \sqrt[3]{2} \times N_{old}$
    - $\sqrt[3]{2} \approx 1.25992$
  - 驗證：
    - $34000 \times 1.25992 \approx 42837$
    - 測資給的是 **42793**，誤差不到 0.1%
- 測量方式：

```
# 測試 1 GPU (hw3-2, p34k1)
echo "=== 1 GPU (hw3-2, p34k1) ==="
time srun -p nvidia -N1 -n1 --gres=gpu:1 ./hw3-2 /home/pp25/share/hw3/testcases/p34k1 /dev/null
```

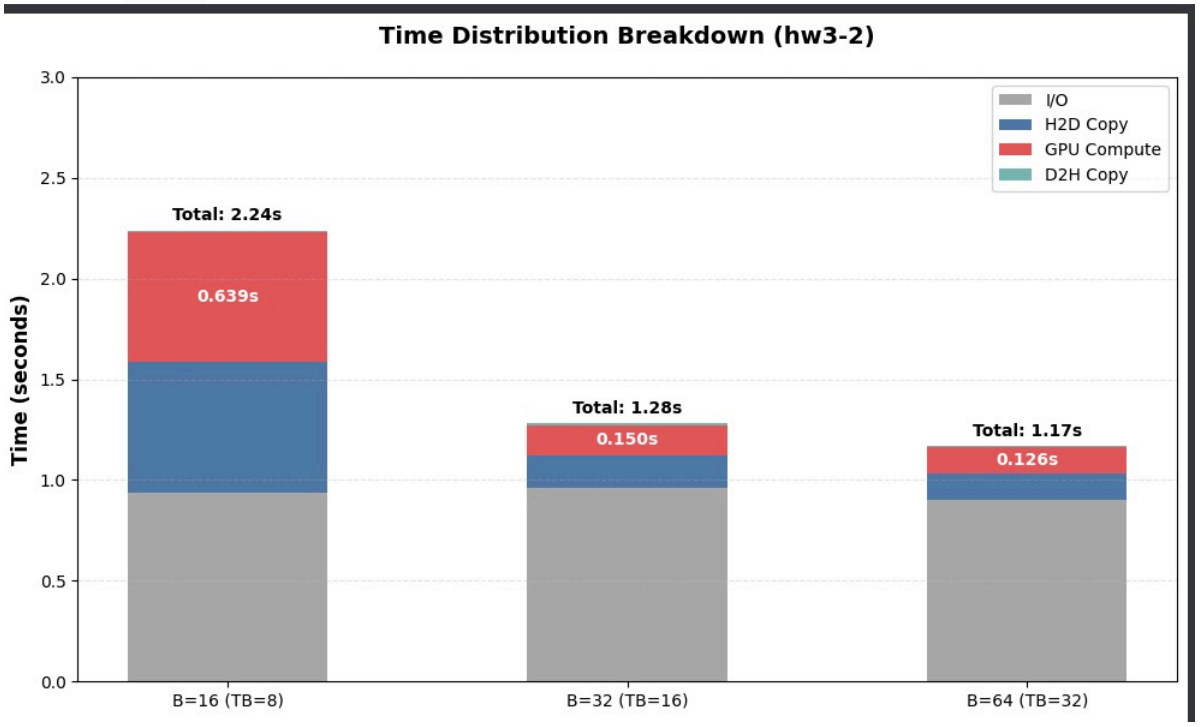
```
# 測試 2 GPUs (hw3-3, p43k1)
echo "=== 2 GPUs (hw3-3, p43k1) ==="
time srun -p nvidia -N1 -n1 --gres=gpu:2 ./hw3-3 /home/pp25/share/hw3/testcases/p43k1 /dev/null
=== 1 GPU (hw3-2, p34k1) ===

real    0m42.492s
user    0m0.007s
sys     0m0.006s
=== 2 GPUs (hw3-3, p43k1) ===

real    0m58.628s
user    0m0.000s
sys     0m0.014s
```

3.5 Time Distribution (hw3-2)

使用測資 `c20.1`



CleanShot 2025-11-28 at 15.15.14.jpg

另外，根據實驗數據，分析程式在 $B = 64$ 時的各階段耗時佔比：

Stage	Time (s)	Percentage	Observation
I/O	0.900	**76.9%**	**Major Bottleneck**
H2D Copy	0.135	11.5%	
GPU Compute	**0.126**	**10.8%**	Extremely Fast
D2H Copy	0.009	0.8%	
Total	1.170	100%	

4. AMD GPU Porting and Analysis

4.1 Impementation

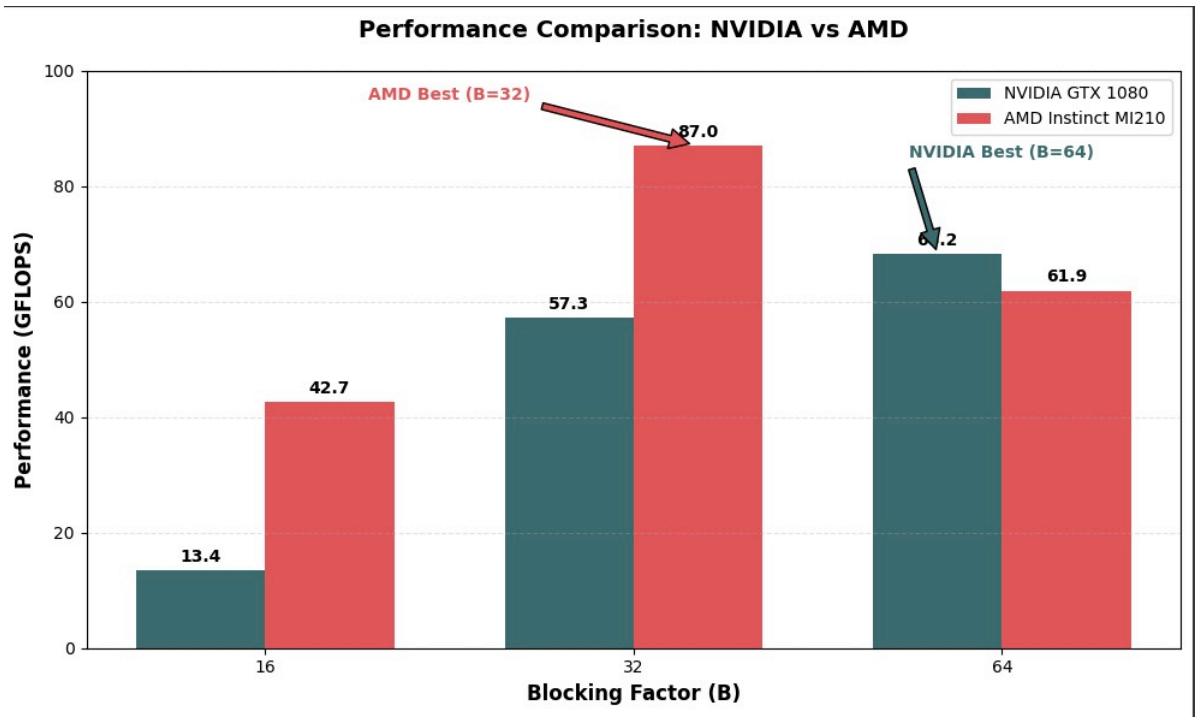
使用 `hipify-clang hw3-2.cu` 去做到轉換。

- **Prefix Mapping:** 將所有 `cuda` 前綴替換為 `hip`。例如：`cudaMalloc` → `hipMalloc`，`cudaMemcpy` → `hipMemcpy`。
- **Type Mapping:** 將 `cudaError_t` 替換為 `hipError_t`。

4.2 Experiment & Analyze

使用測資 `c20.1`，

4.2.1 blocking-factor experiment



CleanShot 2025-11-28 at 21.19.29@2x.jpg

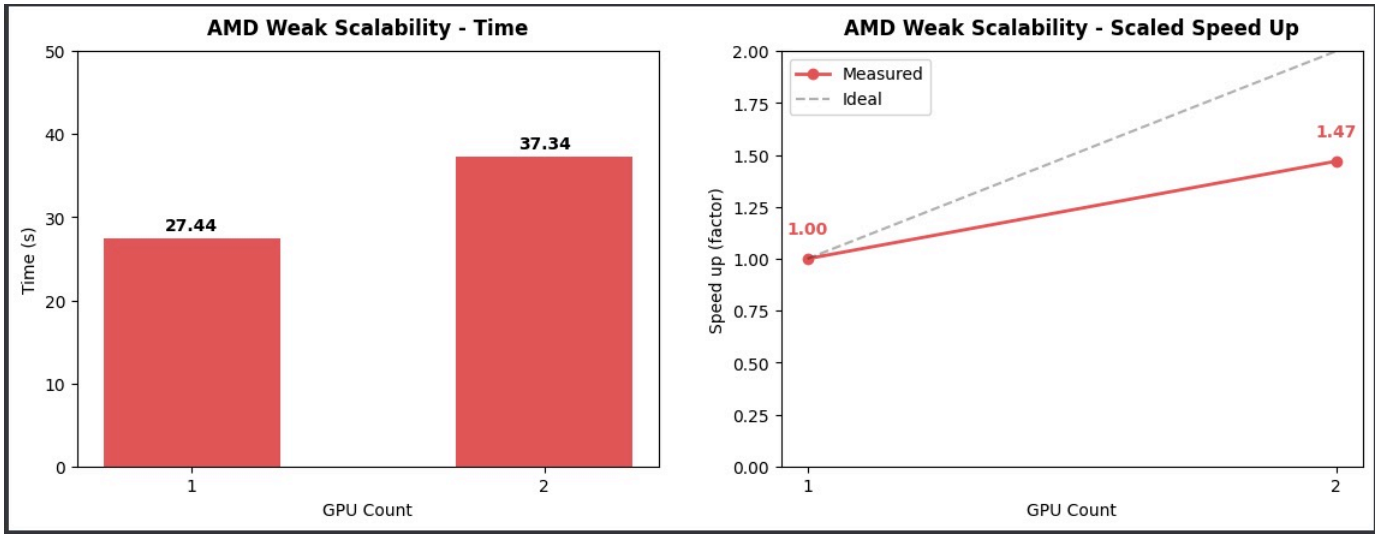
NVIDIA (GTX 1080): 隨著 B 增加，效能持續提升，在 B=64 時達到峰值。但AMD (MI210): 效能在 B=32 時達到峰值 (86.98 GFLOPS)。原因推測是架構上**Wavefront Size** 與 **Resource Granularity** 的不同，AMD CDNA2 架構的 Wavefront Size 為 **64**。

- 當 $B = 32$ ($TB = 16$) 時，Block Size 為 $16 \times 16 = 256$ threads。這剛好是 4 個完整的 Wavefronts (4×64)，調度效率極高。
- 當 $B = 64$ ($TB = 32$) 時，Block Size 為 $32 \times 32 = 1024$ threads。這是一個非常巨大的 Block (16 個 Wavefronts)。在 AMD 架構上，如此巨大的 Block 可能導致 **暫存器壓力 (Register Pressure)** 過大，限制了 CU (Compute Unit) 能夠同時執行的 Active Wavefronts 數量 (Occupancy 下降)，導致無法有效隱藏 HBM 的延遲。

4.2.2 AMD optimization

在針對 AMD MI210 的優化實驗中，我發現了一個與 NVIDIA 截然不同的現象：移除 `#pragma unroll` 反而提升了 16% 的效能 (2.51s $\rightarrow$ 2.15s)。
原因可能是 Unrolling 有時會增加對暫存器的需求。在 AMD 架構上，過高的 Register Pressure 會限制 Wavefront Occupancy，導致無法有效隱藏 HBM2e 的記憶體延遲。

4.2.3 weak-scaling



CleanShot 2025-11-28 at 21.57.22.jpg

透過 `hipify-clang hw3-3.cu` 去做到轉換。使用 **3.4 Weak scalability (hw3-3)** 測試方式去實驗。

```
pp25s098@apollo-login:~/hw3$ srun -p amd -N1 -n1 --gres=gpu:1 time -p ./hw3-2-amd-best
/home/pp25/share/hw3/testcases/p34k1 /dev/null
real 27.44
user 11.36
sys 13.98
pp25s098@apollo-login:~/hw3$ srun -p amd -N1 -n1 --gres=gpu:2 time -p ./hw3-3-amd-best
/home/pp25/share/hw3/testcases/p43k1 /dev/null
real 37.34
user 14.85
sys 23.50
pp25s098@apollo-login:~/hw3$
```

5. Experience & conclusion

這次的作業比前一次明顯難很多。最大的挑戰不是把程式寫出來，而是 你必須真的理解 GPU 裡面每一層記憶體在做什麼、資料現在到底待在哪、下一步該怎麼搬。如何去作 profiling，然後了解背後的意思也相對之前難上許多。這次做完我覺得我對 GPU 的認識真的比之前深太多了，尤其是「什麼東西該搬、什麼東西不該搬、搬太多會死、搬太少又浪費」，這一整套直覺，是這次作業逼出來的。