Implementation

Which algorithm do you choose in hw3-1?

- 使用OpenMP針對block_FW()中phase2的cal()呼叫以 `#pragma omp parallel sections` 以及 `#pragma omp section` 做優化。phase3的雙層迴圈cal()呼叫以 `#pragma omp parallel for collapse(2) schedule(dynamic)` 做優化
-
```
/* Phase 2: Update the related row and column blocks */
#pragma omp parallel sections
{
    #pragma omp section
    cal(B, r, r, 0, r, 1);
    #pragma omp section
    cal(B, r, r, r + 1, round - r - 1, 1);
    #pragma omp section
    cal(B, r, 0, r, 1, r);
    #pragma omp section
    cal(B, r, r + 1, r, 1, round - r - 1);
}
```
-
```
/* Phase 3: Update the remaining blocks */
#pragma omp parallel for collapse(2) schedule(dynamic)
for (int i = 0; i < round; ++i) {
    for (int j = 0; j < round; ++j) {
        if (i != r && j != r) {
            cal(B, r, i, j, 1, 1);
        }
    }
}
```
- 使用OpenMP針對cal()最外層迴圈以 `#pragma omp parallel for collapse(2) schedule(dynamic)` 做優化
-
```
void cal(
    int B, int Round, int block_start_x, int block_start_y, int block_width, int block_height) {
    int block_end_x = block_start_x + block_height;
    int block_end_y = block_start_y + block_width;

    #pragma omp parallel for collapse(2) 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) {
```

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

hw3-2 & hw3-3 divide data的方式相同

- 將 n×n 的 Dist 做 padding，使得 n 為 64 的倍數（64 為我選擇的 blocking factor）。接著以 64×64 筆資料為單位送進一個 GPU 的 block 中執行。在每一個 block 中會有 32×32 個 thread，每個 thread 會負責處理 4 筆資料。

```
n_padded = n + (BLOCK_SIZE - (n % BLOCK_SIZE)) % BLOCK_SIZE;
// 分配 host memory (大小為 n_padded * n_padded)
    Dist_h = (int*)malloc((size_t)n_padded * (size_t)n_padded * sizeof(int)
```

將 n 向上對齊到 64 的倍數，例如若 n=130，則 n_padded 會變為 192。

What's your configuration in hw3-2, hw3-3? And why? (e.g. blocking factor, #blocks, #threads)

hw3-2,hw3-3使用相同的配置

blocking factor

- 程式中將 tile 大小設定為 64×64 (即 BLOCK_SIZE=64)。
- 每個 block 會透過 shared memory 讀寫該 64×64 區塊，加速 Floyd-Warshall 更新。
- 之所以選 64×64，是因為 GPU 一個 block 最多可以容納 1024 個 threads，而程式中採用 dim3(32, 32) 的 threads configuration，剛好對應到 64×64 資料的載入與處理（每個 thread 會負責 4 筆資料）。

blocks & threads

- Threads：每個 block 固定為 32×32 threads；
- Blocks：
  - 在 phase1，只需要處理對角區塊 (pivot block) => 啟動 1 個 block。
  - 在 phase2，需要處理 pivot row 與 pivot column 的其他區塊 => 啟動 2 × (round - 1) 個 blocks，其中 round = n_padded / 64。
  - 在 phase3，處理其餘區塊 => 啟動 (round - 1) × (round - 1) 個 blocks。

```
void block_FW() {
    int round = n_padded / BLOCK_SIZE;  // 例如 n_padded=192 => round=3
    // 我們的 kernel 會使用 32x32 個 threads，並在 kernel 內處理 64x64 區塊
    dim3 threads(32, 32);
    for (int r = 0; r < round; r++) {
        // Phase1: 對角區塊
        phase1_kernel<<<1, threads>>>(Dist_d, n_padded, r);
        // Phase2: 同 row 和同 column 的所有區塊
        //  - gridSize = (2, round-1)，意即 row-block + col-block 各一橫排/直排
        //    blockIdx.x = 0 or 1 => 0: 代表 col blocks, 1: 代表 row blocks
        //    blockIdx.y = 0..(round-1) 但跳過 r
        if (round > 1) {
            dim3 grid2(2, round - 1);
            phase2_kernel<<<grid2, threads>>>(Dist_d, n_padded, r);
        }
        // Phase3: 其餘區塊
        //  - gridSize = (round-1, round-1)，代表跳過 r 的所有 row, col 組合
        if (round > 1) {
            dim3 grid3(round - 1, round - 1);
            phase3_kernel<<<grid3, threads>>>(Dist_d, n_padded, r);
        }
    }

    nvtxRangePop();
}
```

How do you implement the communication in hw3-3?

如果round是奇數就交給gpu0計算，round是偶數就交給gpu1計算並在每個round結束時透過 `cudaMemcpyPeer` 把更新後的整個 Dist (大小 n_padded*n_padded) 從拷貝到另外一台gpu，確保兩卡資料一致。

```
size_t distSize = (size_t)n_padded * (size_t)n_padded * sizeof(int);
cudaMemcpyPeer( (other_gpu == 0 ? Dist_d0 : Dist_d1), other_gpu,
                (this_gpu  == 0 ? Dist_d0 : Dist_d1), this_gpu,
                distSize );
```

Briefly describe your implementations in diagrams, figures or sentences.

Padding

- 以 BLOCK_SIZE = 64 為基準，將 n padding 到 64 的倍數 (n_padded)；若 n=130，則 n_padded=192。
- 透過一般 malloc 分配 Host 端記憶體，並以 cudaMalloc 分配 GPU 端的記憶體。

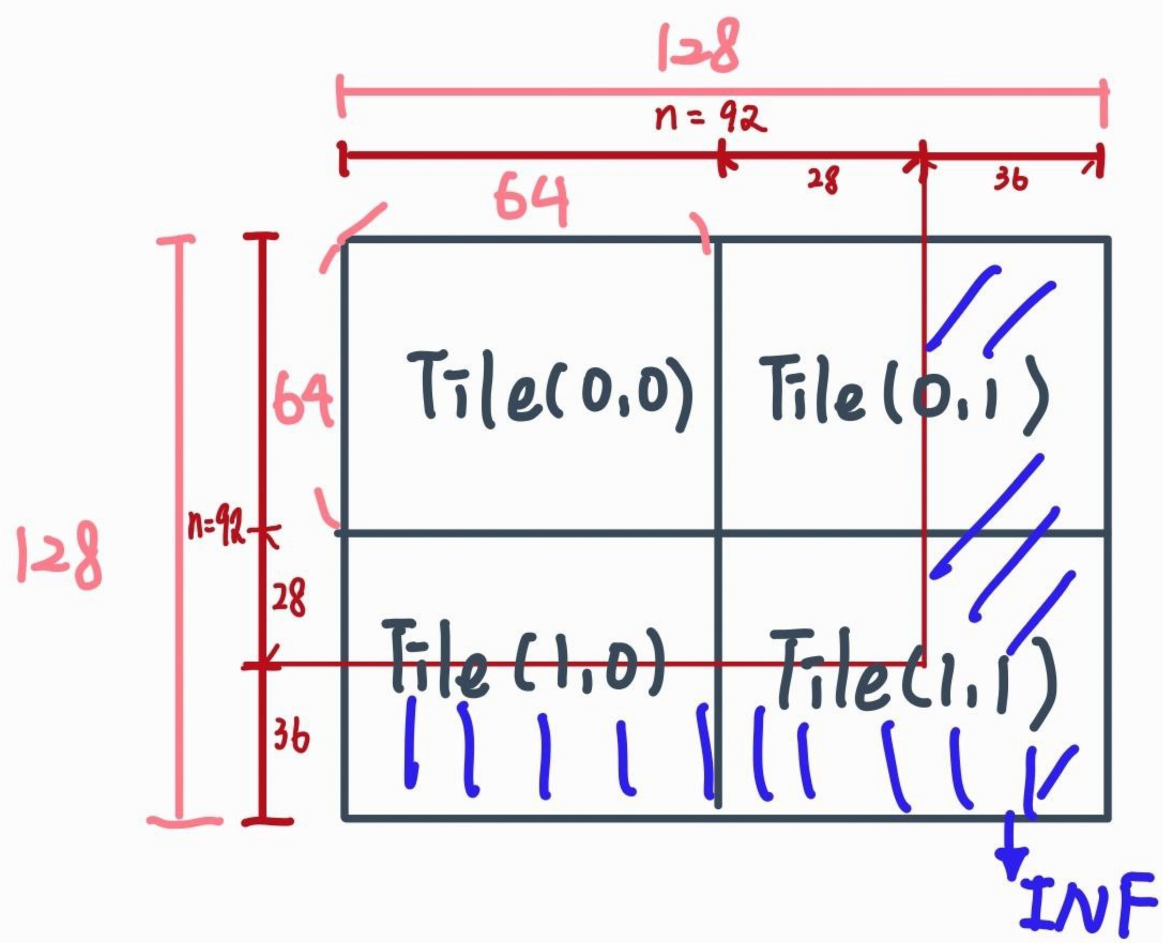
Kernel & Shared Memory

- 實作了三個 kernel：phase1_kernel, phase2_kernel, phase3_kernel，分別對應到 對角區塊 (phase1)、同 row/column 區塊 (phase2) 和 其他區塊 (phase3)。
- 每個 kernel 中都以 **shared** int s[...] 存放資料到share memory
  - phase1負責對角區塊只涉及一個區塊，單一的 64×64 shared memory大小已經足夠
  - phase2需要存放2*64×64 區塊的資料：一個是對角區塊所在行的區塊（row-part），另一個是對角區塊所在列的區塊（column-part）。
  - phase3負責更新除對角區塊、同行區塊和同列區塊之外的所有其他區塊。這些區塊的更新依賴於對角區塊的同行和同列區塊的資料。因此同樣需要存放2*64×64 區塊的資料
- 每個 block 有 32×32 個 threads，每個 thread 在載入或更新階段會負責 4 個元素（例如 (i, j), (i, j+32), (i+32, j), (i+32, j+32)）。更新後再將結果寫回 global memory。

```
// 寫回 global memory phase1為例
    dist[(b_i + i)*n_padded + (b_j + j)]            = s[i*64 + j];
    dist[(b_i + i)*n_padded + (b_j + j + 32)]       = s[i*64 + (j + 32)];
    dist[(b_i + i + 32)*n_padded + (b_j + j)]       = s[(i + 32)*64 + j];
    dist[(b_i + i + 32)*n_padded + (b_j + j + 32)]  = s[(i + 32)*64 + (j +
```

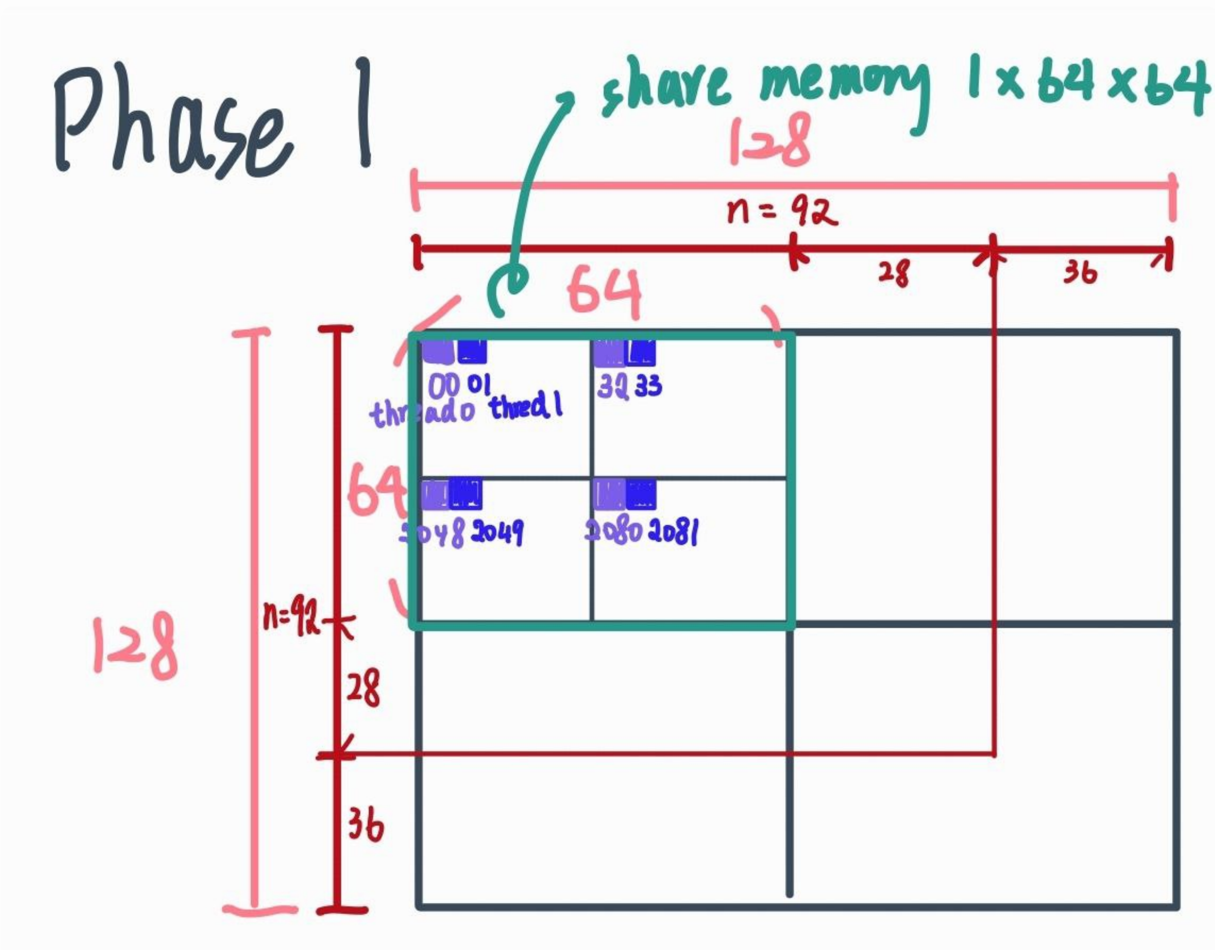
圖片說明

舉n=92為例 `n_padded=n+(64-(nmod64))mod64` n_padded=92，距離矩陣 Dist 將被擴展為 128×128，並劃分為 2×2 的 64×64 tile。`round = n_padded / 64` round = 2。



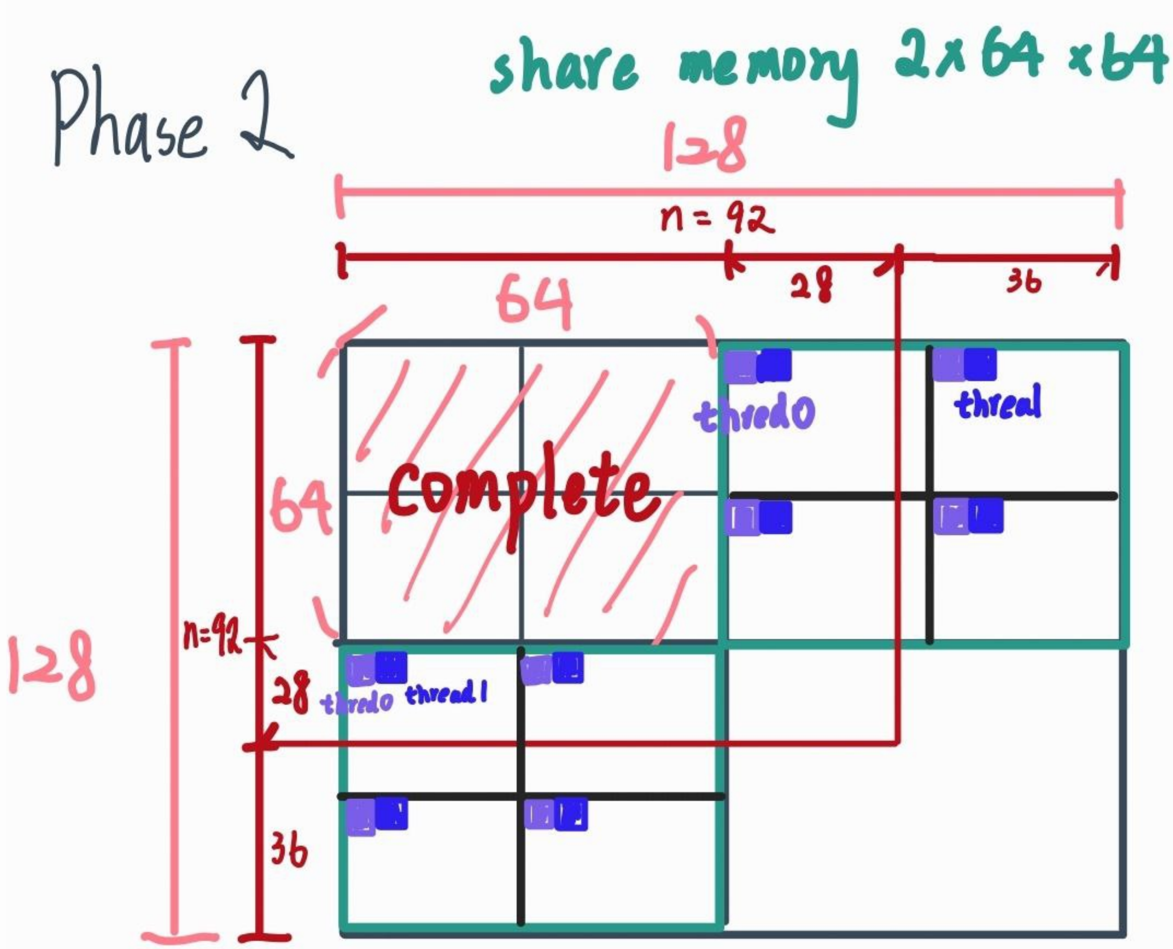
- Phase1使用32*32thread每個thread負責計算4筆Tile(0,0)資料，將Tile(0,0)載入Shared Memory (s[64][64]):

```
Shared Memory (s[64][64]):
+-----+-----+-----+-----+ ... +-----+
| s00 | s01 | ... | s63 |     |
+-----+-----+-----+-----+ ... +-----+
| s64 | s65 | ... | s127|     |
+-----+-----+-----+-----+ ... +-----+
| ... | ... | ... | ... |     |
+-----+-----+-----+-----+ ... +-----+
|s4032| ... | ... |s4095|     |
+-----+-----+-----+-----+ ... +-----+

每個 thread 負責載入 4 個元素：
- Thread (0,0) 載入 s00, s32, s2048, s2080
- Thread (0,1) 載入 s01, s33, s2049, s2081
- ...
- Thread (31,31) 載入 s31, s63, s2079, s2111

```



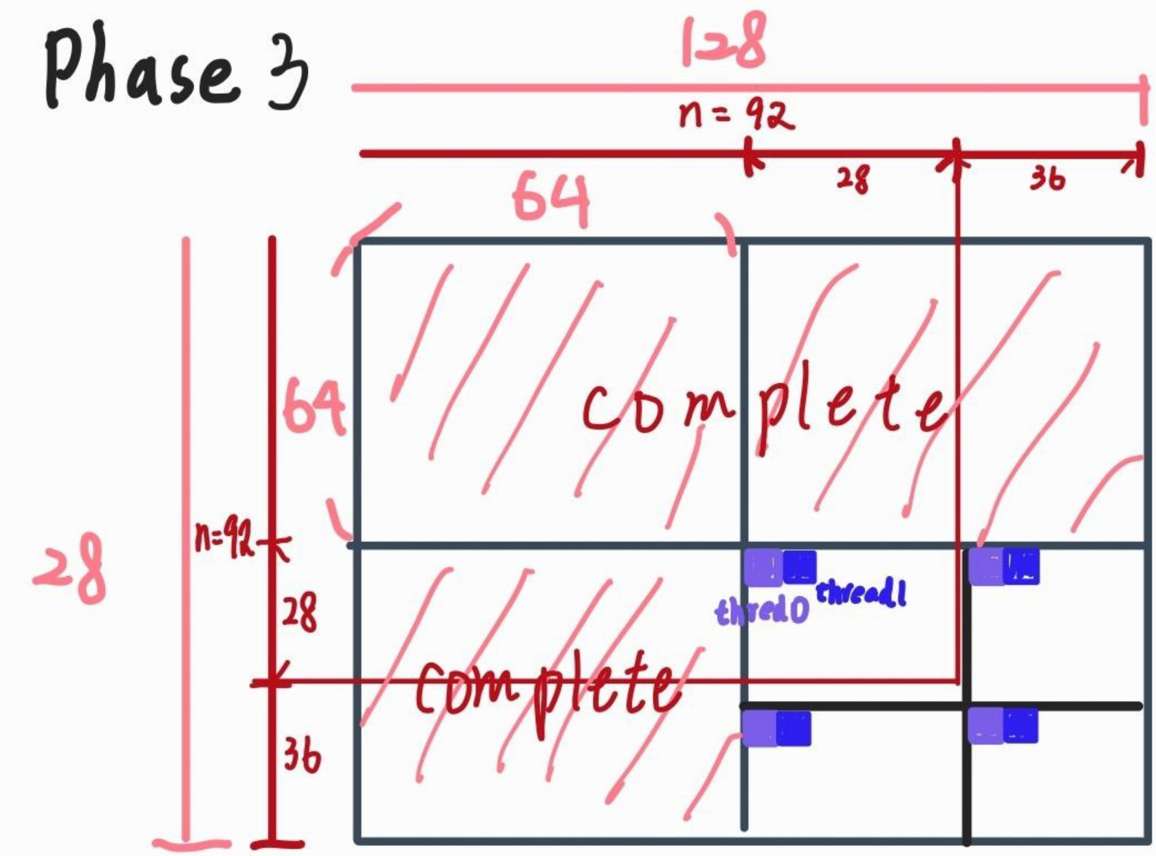
- Phase2使用32*32thread每個thread負責計算4筆Tile(0,1) row資料以及Tile(1,0)column資料，將Tile(0,1),Tile(1,0)載入Shared Memory (s[2*64][64]):

```
Shared Memory (s[2*64][64]):
+-----+-----+-----+-----+ ... +-----+-----+-----+-----+ ... +-----+
| s00 | s01 | ... | s63 |     | s4096 | s4097 | ... | s8191 |     |
+-----+-----+-----+-----+ ... +-----+-----+-----+-----+ ... +-----+
| s64 | s65 | ... | s127|     | s4096+64 | ... | s8191+64 | ...|
+-----+-----+-----+-----+ ... +-----+-----+-----+-----+ ... +-----+
| ... | ... | ... | ... |     | ... | ... | ... | ... |     |
+-----+-----+-----+-----+ ... +-----+-----+-----+-----+ ... +-----+
|s4032| ... | ... |s4095|     | s8192 | ... | s12287| ... |     |
+-----+-----+-----+-----+ ... +-----+-----+-----+-----+ ... +-----+

每個 thread 負責載入 4 個元素來自 row 或 column tile：
- **Tile (0,1)**:
    - Thread (0,0) 載入 s4096, s4128, s0, s32
    - Thread (0,1) 載入 s4097, s4129, s1, s33
    - ...
- **Tile (1,0)**:
    - Thread (0,0) 載入 s0, s32, s4096, s4128
    - Thread (0,1) 載入 s1, s33, s4097, s4129
    - ...
```



- Phase3使用32*32thread每個thread負責計算4筆Tile(1,1)

```
Shared Memory (s[2*64][64]):
+-----+-----+-----+-----+ ... +-----+-----+-----+-----+ ... +-----+
| s00 | s01 | ... | s63 |     | s4096 | s4097 | ... | s8191 |     |
+-----+-----+-----+-----+ ... +-----+-----+-----+-----+ ... +-----+
| s64 | s65 | ... | s127|     | s4096+64 | ... | s8191+64 | ...|
+-----+-----+-----+-----+ ... +-----+-----+-----+-----+ ... +-----+
| ... | ... | ... | ... |     | ... | ... | ... | ... |     |
+-----+-----+-----+-----+ ... +-----+-----+-----+-----+ ... +-----+
|s4032| ... | ... |s4095|     | s8192 | ... | s12287| ... |     |
+-----+-----+-----+-----+ ... +-----+-----+-----+-----+ ... +-----+

每個 thread 負責載入 4 個元素來自 Tile (1,1) 和 Pivot Tiles (0,1) & (1,0):
- Thread (0,0) 載入 s4096, s4128, s0, s32
- Thread (0,1) 載入 s4097, s4129, s1, s33
- ...
```



Profiling Results (hw3-2)

以下為使用 c21.1 為測資做 profiling 的結果，可以發現在 phase2 和 phase3 處理的資料比較龐大，不管在 occupancy 、 sm efficiency 、各種 throughput 上都比只處理 1 個 block 的 phase1 來的大上許多，將資源更加充分利用。

```
==963631== Profiling result:
==963631== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: phase3(int, int, int*, int)
         79                        achieved_occupancy                        Achieved Occupancy    0.939451    0.944743    0.943030
         79                             sm_efficiency                   Multiprocessor Activity      99.49%      99.67%      99.60%
         79                    shared_load_throughput             Shared Memory Load Throughput  2931.2GB/s  3078.5GB/s  2956.9GB/s
         79                   shared_store_throughput            Shared Memory Store Throughput  122.14GB/s  128.27GB/s  123.20GB/s
         79                            gld_throughput                    Global Load Throughput  183.20GB/s  192.40GB/s  184.80GB/s
         79                            gst_throughput                   Global Store Throughput  61.068GB/s  64.135GB/s  61.602GB/s
    Kernel: phase1(int, int, int*, int)
         79                        achieved_occupancy                        Achieved Occupancy    0.499070    0.499147    0.499096
         79                             sm_efficiency                   Multiprocessor Activity       4.44%       4.54%       4.52%
         79                    shared_load_throughput             Shared Memory Load Throughput  109.49GB/s  117.38GB/s  111.07GB/s
         79                   shared_store_throughput            Shared Memory Store Throughput  36.971GB/s  39.635GB/s  37.504GB/s
         79                            gld_throughput                    Global Load Throughput  582.44MB/s  624.40MB/s  590.83MB/s
         79                            gst_throughput                   Global Store Throughput  582.44MB/s  624.40MB/s  590.83MB/s
    Kernel: phase2(int, int, int*, int)
         79                        achieved_occupancy                        Achieved Occupancy    0.893273    0.931404    0.910185
         79                             sm_efficiency                   Multiprocessor Activity      85.99%      91.26%      88.60%
         79                    shared_load_throughput             Shared Memory Load Throughput  2345.9GB/s  2637.7GB/s  2457.5GB/s
         79                   shared_store_throughput            Shared Memory Store Throughput  97.746GB/s  109.90GB/s  102.40GB/s
         79                            gld_throughput                    Global Load Throughput  146.62GB/s  164.85GB/s  153.59GB/s
         79                            gst_throughput                   Global Store Throughput  48.873GB/s  54.951GB/s  51.198GB/s
```

Experiment & Analysis

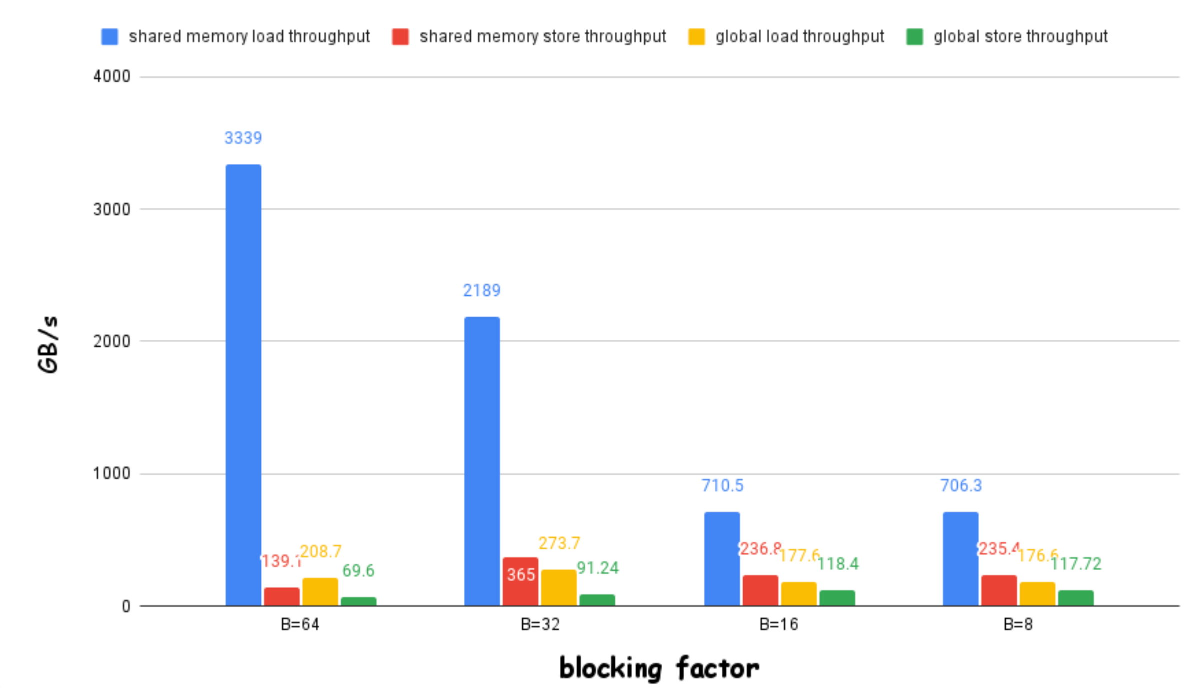
System Spec apollo

Blocking Factor (hw3-2)

Observe what happened with different blocking factors, and plot the trend in terms of Integer GOPS and global/shared memory bandwidth. (You can get the information from profiling tools or manual) (You might want to check nvprof and Metrics Reference)

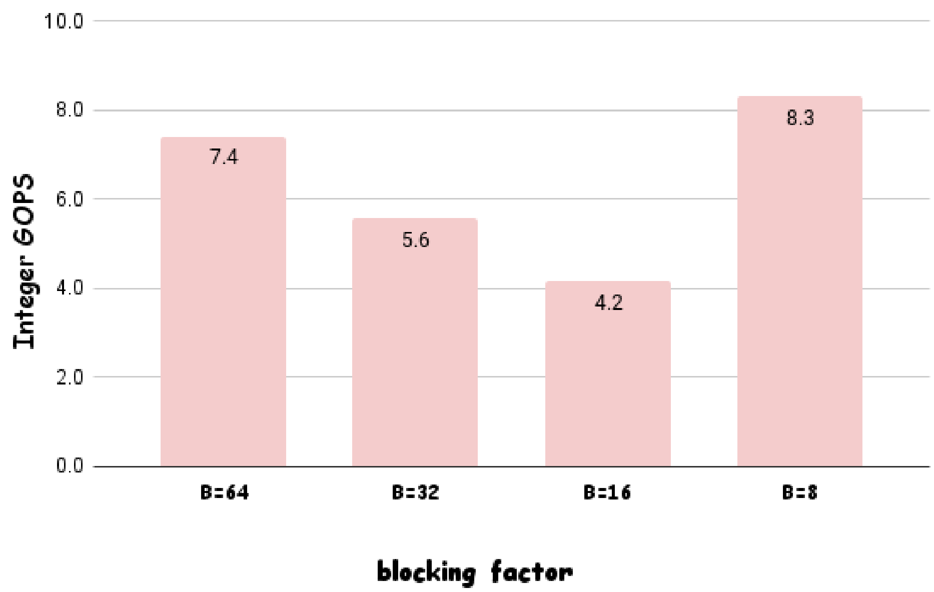
testcase:c21.1 phase3_kernel

由下表可以發現當B=64時，shared memory bandwidth最高，代表當B=64能最大的發揮share memory的功用



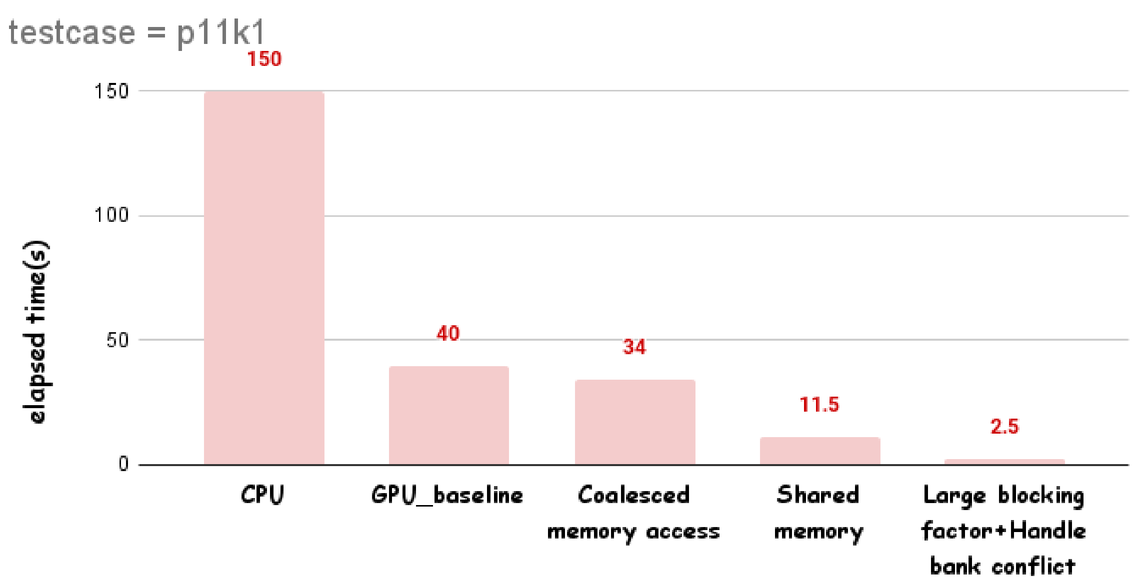
由下表可知當B=8所耗費的運算資源最多，但執行最久。反觀B=64則是使用第二多的運算資源，執

行時間較快



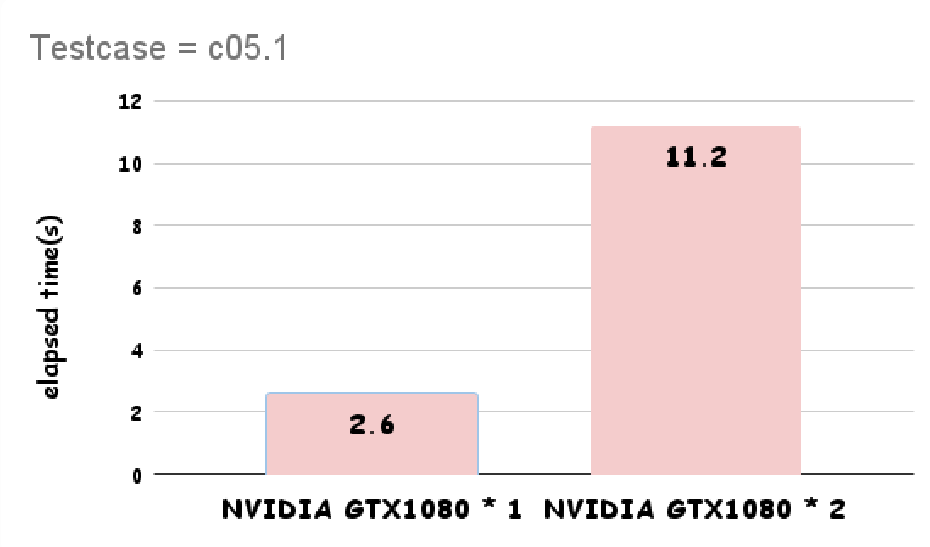
Optimization (hw3-2)

想要通過p11k1以上的testcase必須要解決bank conflict並用更大的blocking factor做計算，若只單純針對原本的cal()做share memory 和 coalesced memory access效能上沒辦法有顯著提升。



Weak scalability (hw3-3)

使用一張GTX1080比兩張GTX1080還要快，那是因為當使用兩張gpu做運算時需要用 `cudaStreamSynchronize` 同步兩張gpu的資料非常耗時，因此單張GTX1080效能比較好



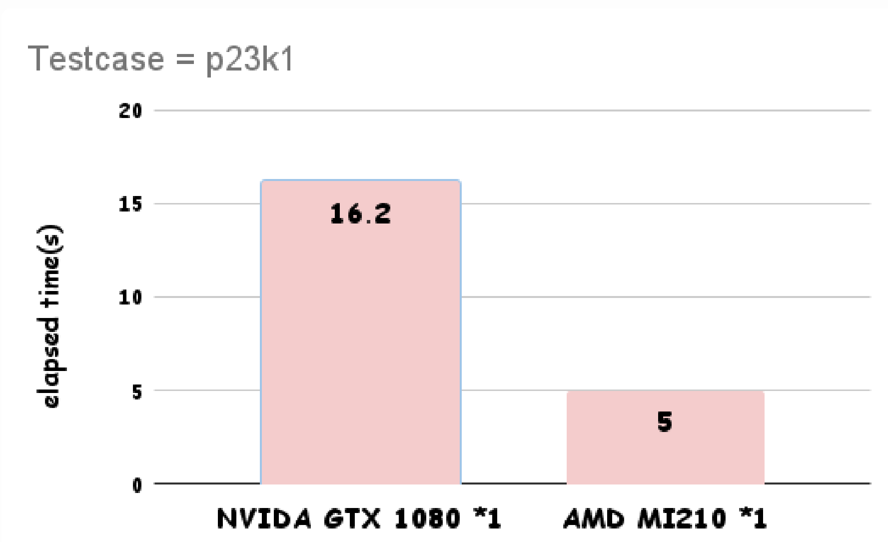
Time Distribution (hw3-2)

以p23k1測資為例，根據nsight system profile結果 total elpased time = 16.5s
computing time = blockFW()計算時間　= 10.423s
memory copy (H2D, D2H) = 1s
I/O time = input()+output()執行時間 = 1s + 3.7s = 4.7s

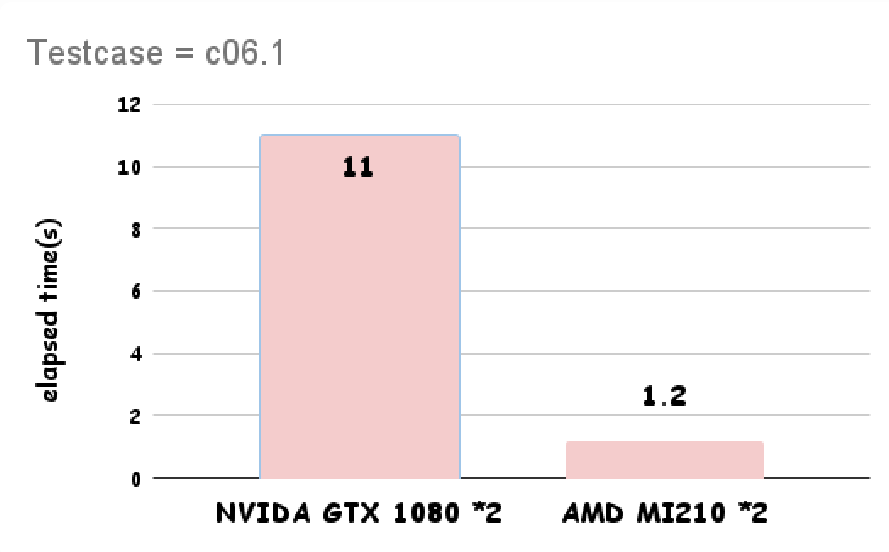
Experiment on AMD GPU

- Share your insight and reflection on the AMD GPU experiment.

- single GPU experiment



- multi GPU experiment



根據實驗結果發現使用AMD MI210運算要比NVIDIA GTX 1080要快
使用一張MI210可以加速2~3倍，使用兩張MI210可以加速約11倍
(跑hw3-3-amd-judge並沒有出錯)

Experience & conclusion

- What have you learned from this homework?
  對我而言我覺得hw3是五項作業裡最難的了，不僅考驗cuda還要考驗數學，前階段我只針對原始code架構中的cal()進行share memory 和 coalesced memory access優化並沒有針對cal()的計算模式做大幅修改，後來上網查找、參考論文以及其他人的做法，才知道原來要將phase1,phase2,phase3分成三個不同的kernel使用不同的block,share memory配置做計算。才好不容易擁有大幅的效能提升。