Homework 3: All-Pairs Shortest Path Report

1. Implementation

1.a Which algorithm do you choose in hw3-1?

我採用 Blocked Floyd-Warshall algorithm，主要將三個部分利用 OpenMP 平行化：

1. 初始化 `Dist` 矩陣的部分，用 `#pragma omp parallel for collapse(2) schedule(static)` 來平行化兩層迴圈

```
#pragma omp parallel for collapse(2) schedule(static)
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        if (i == j) {
            Dist[i][j] = 0;
        } else {
            Dist[i][j] = INF;
        }
    }
}
```

2. 輸出前，需將無窮大的值統一設為 `INF`，確保輸出一致性，這部分用 `#pragma omp parallel for collapse(2) schedule(static)` 平行化處理距離矩陣中的每一 row

```
#pragma omp parallel for collapse(2) schedule(static)
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        if (Dist[i][j] >= INF) Dist[i][j] = INF;
    }
}
```

3. 在 `cal` 函式中，內層迴圈更新每個區塊內的距離的部分，用 `#pragma omp parallel for schedule(static)` 來平行化。

```
#pragma omp parallel for schedule(static)
for (int i = block_internal_start_x; i < block_internal_end_x; ++i) {
    for (int j = block_internal_start_y; j < block_internal_end_y; ++j) {
        if (Dist[i][k] + Dist[k][j] < Dist[i][j]) {
            Dist[i][j] = Dist[i][k] + Dist[k][j];
        }
```

```
        }
    }
```

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

hw3-2

在 hw3-2 中，我先將 input 矩陣 (Dist) 的 x 方向與 y 方向擴充到 64 的倍數，讓 number of rows 與 number of columns 都可以被 64 整除。接著，我把整個 Dist 矩陣切成一塊塊 64x64 的 block 來處理。每個小區塊會對應到 GPU kernel 中的 block，然後在 block 裡由 32x32 個 threads 負責計算，所以每個 thread 需要負責計算 4 筆 data (phase 1、phase 2、phase 3 的 kernel 都是這樣)。

hw3-3

在 hw3-3 中，我將完整的 N * N Dist 矩陣分配到兩張 GPU 上，因為這樣實作比較簡單，不過在大測資的情況下 global memory 可能會爆掉，然後，因為 phase 3 的計算量最大，所以我只有將 phase 3 分給兩個 GPU 來做，分配方式如下：

假設整個矩陣被拆成 R 個 row block，GPU 0 會被分配到 $[0, R/2)$ 的 row blocks (無條件捨去)，GPU 1 會被分配到 $[R/2, R]$ 的 row blocks，確保覆蓋到完整的矩陣。

1.c What's your configuration in hw3-2, hw3-3? And why?

hw3-2

- **Blocking factor = 64**:
  選擇 64 的原因是希望能盡量利用 shared memory。在執行 Phase 3 時，需要同時載入三個 block 的資料到 shared memory 中。
  計算方式如下：
  - 每個 integer 用 4 bytes 表示
  - 一個 block 最多可用的 shared memory 為 49152 bytes
  - 將 49152 bytes 除以 4 (bytes/int) 再除以 3 (三個 block)
    → (49152 / 4) / 3 = 4096 int
  - 4096 int 可以用一個 64x64 的區塊來存 (64*64 = 4096)
    因此，用 64 做為 blocking factor，可以最大化利用 shared memory 的資源。

- **Threads per Block = 32 x 32**:
  - 因為一個 block 最多能使用 1024 個 threads，所以為了盡量將工作量平行化，所以選擇用滿全部的 threads，也就是每個 block 以 32x32 threads 執行。

- **Grid Configuration**:
  整體的矩陣大小在 padding 後是 padded_n x padded_n，所以 x 方向與 y 方向會被切割成

n_round = padded_n / 64 份，為了完整覆蓋整個矩陣，kernel 的 grid size 設置如下：

- Phase 1：只需針對位於該回合 pivot block 執行 → 使用 1x1 block grid
- Phase 2：需要更新該 pivot row 與 pivot column 上的 block → grid 設為 (2, n_round)（"2" 分別對應到 row-block 與 column-block 的更新）
- Phase 3：同時更新整個矩陣中非 pivot row 與 column 的其他 block → grid 設為 (n_round, n_round)

hw3-3

- hw3-3 其實大部分跟 hw3-2 一樣，只是差在 Grid Configuration 的部分 Phase 3 設成 `dim3 phase3_grid(n_round, block_rows_for_gpu)`，也就是第二個維度的部分變成那張 GPU 要處理幾個 row blocks

1.d How do you implement the communication in hw3-3?

在 hw3-3 中，我透過 Peer-to-Peer memory copy 與 cudaMemcpyPeerAsync 在兩張 GPU 間交換必要的 pivot row，以便另一張卡能在接下來的 phase 1、phase 2、phase 3 中讀取正確的資料。作法如下：

- 每回合 (round) 前的一開始先判斷 pivot row 屬於哪張 GPU；若需要給另一張 GPU 使用，便透過 `cudaMemcpyPeerAsync` 將該 row 複製過去。
- 使用 `cudaStreamSynchronize()` ，確保 pivot row 完整拷貝後，雙方再各自執行後續階段的 kernel。
- 最後將 GPU0 與 GPU1 各自負責的 block rows 回傳到 host，合併成完整結果。

1.e Briefly describe your implementations in diagrams, figures, or sentences.

hw3-2

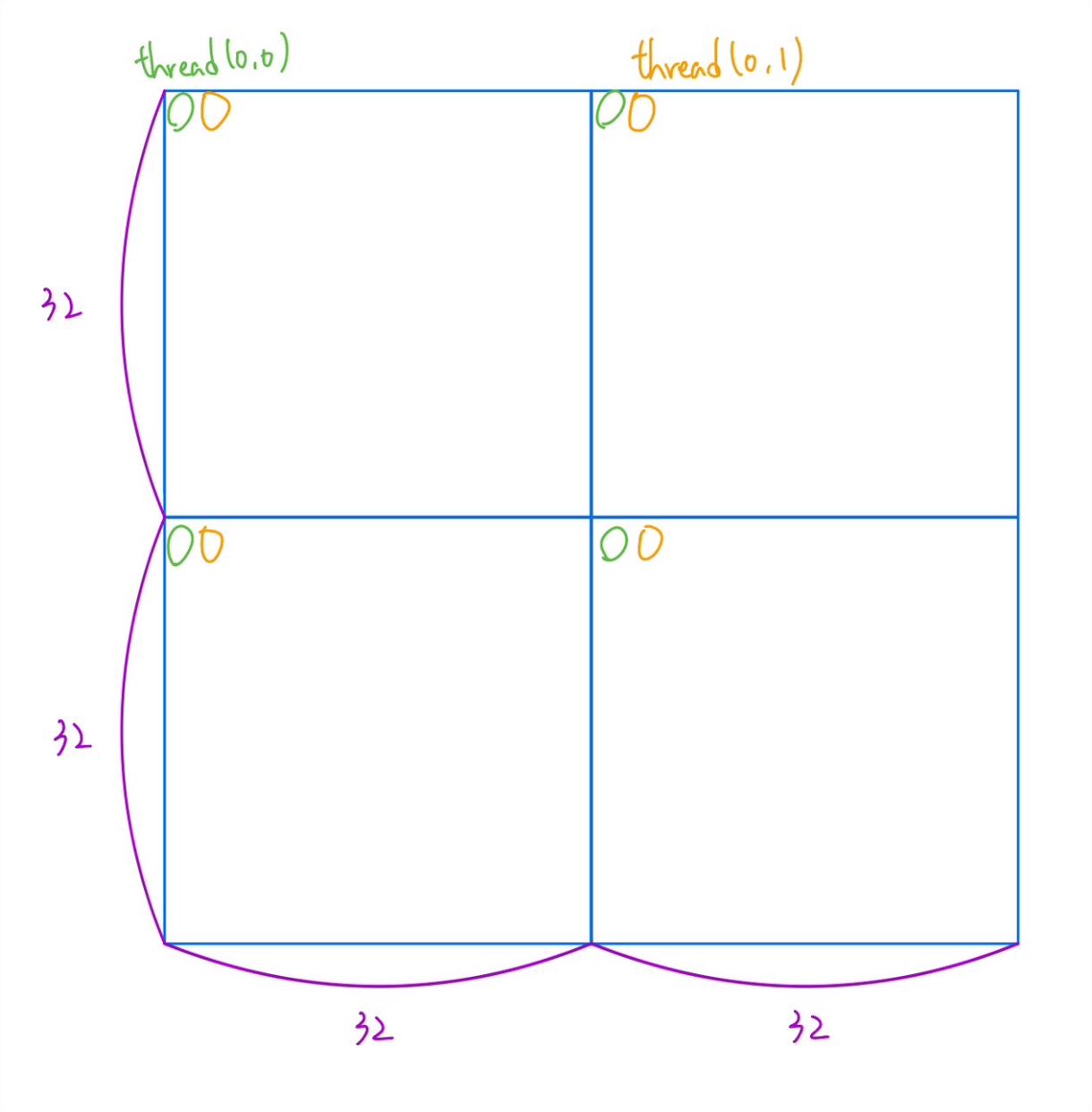
記憶體搬移 (Host ↔ GPU)：

在進行計算前，會先用 `cudaMalloc()` 在 GPU 上分配一塊能容納整個距離矩陣的記憶體，接著用 `cudaMemcpy(..., cudaMemcpyHostToDevice)` 將 input data 由 host 端複製到 device 端。計算完成後，再以 `cudaMemcpy(..., cudaMemcpyDeviceToHost)` 將結果複製回 host 端。

Phase 1 ~ Phase 3 說明：

為了計算完整的距離矩陣，總共會執行 `n_round` 個回合，`n_round = padded_n / 64;`，`padded_n` 是將 vertex 數量 padding 到 64 的倍數 (為了配合 blocking factor 64)，在每個回合中，都會執行 Phase 1 ~ Phase 3 kernel，三個 kernel 都是對應到一個 64 * 64 的 block，並且三個 kernel mapping 的方式都一樣，都是讓每個 thread 負責矩陣中的四個位置。以一個 64x64 的區塊為例，將其視為四個 32x32 的子區域組合在一起（左上、右上、左下、右下）。假設有個 thread 是 (0,0)，它就會同時處理位於這個 64x64 區塊中左上角 (0,0)、右上角

(0,32)、左下角 (32,0) 和右下角 (32,32) 這四個點的資料，而 thread (0,1) 就會負責 (0,1)、(0,33)、(32,1)、(32,33)，以此類推，32 * 32 個 threads 分別對應到左上、右上、左下、右下四個 32 * 32 的矩陣區塊，如下圖所示：



透過這種對應方式，可以讓一個 warp (32 個 threads) 在存取 global memory 時是 coalesces 的。至於每個位置的計算方式就是根據不同 phase 而需要到不同的區塊查找資料並比較更新，說明如下：

Phase 1 (更新 Pivot Block)：
在這個階段，會針對目前回合的 pivot block（位於 pivot 行與 pivot 列交叉處的 64x64 區塊）進行 Floyd-Warshall 的基本步驟更新。具體來說：

1. 將該 pivot block 的資料載入 shared memory 中，以利後續的快速存取。
2. 在迴圈中，對每個可能的中介點 k 進行更新計算：

```
for (int k = 0; k < BLOCK_WORK; k++) {
    // 利用 pivot block 的 row 與 column 進行距離更新
    block[ty][tx] = min(block[ty][tx], block[ty][k] + block[k][tx]);
    ...
    __syncthreads(); // 確保所有 threads 完成此輪更新後再進行下一輪
}
```

我們在每次迴圈結束後使用 `__syncthreads()`，是因為 phase 1 的 iterations 之間有 dependency。若沒有同步，某些 threads 可能會在其他 threads 尚未更新完成時就讀取資料，導致不正確的結果。透過 `__syncthreads()` 確保所有 thread 都在同一階段完成後再進入下一階段。

Phase 2 (更新 Pivot Row & Pivot Column)：
Phase 2 是在完成 pivot block 的更新後，利用該已更新的 pivot block 當作「中介」來修正該 pivot 行與 pivot 列上的其他 64x64 區塊。作法是：

1. 將 pivot block 與目標的 row-block 或 column-block 同時載入 shared memory。
2. 與 Phase 1 類似，使用一個迴圈針對每個中介點 k 進行更新：

```
for (int k = 0; k < BLOCK_WORK; k++) {
    // 利用已更新的 pivot 資訊更新該行或該列的其他 block
    current[ty][tx] = min(current[ty][tx], current[ty][k] + pivot[k][tx]);
    ...
    __syncthreads(); // 同步確保該輪更新完成
}
```

同樣地，必須在每次更新迴圈中使用 `__syncthreads()` 來確保整個 block 的 threads 同步，因為 phase 2 的 iterations 之間也有 dependency。

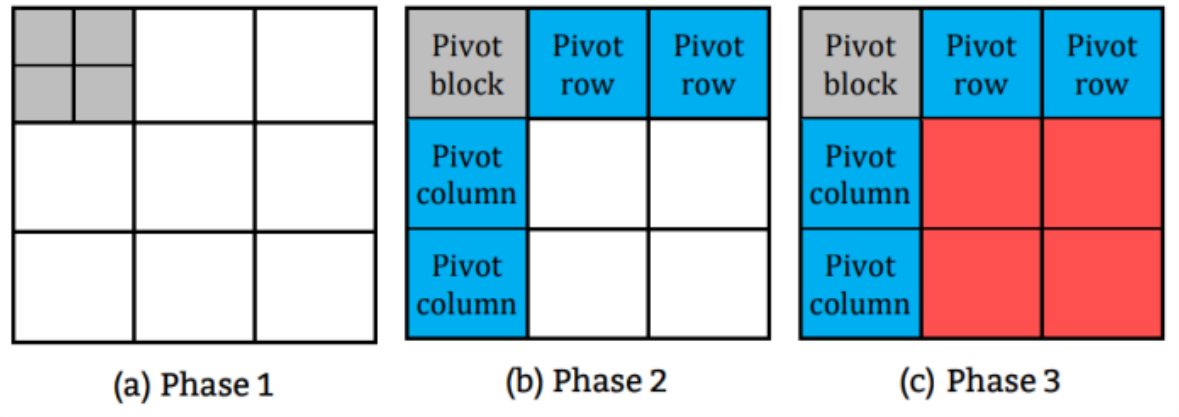
Phase 3 (更新剩餘區域)：
在完成 pivot block 以及 pivot row/column 的更新後，我們已擁有較完整的距離資訊。Phase 3 負責更新整個矩陣中不屬於 pivot row/column 的區域。此時：

1. 同時將 row pivot block、column pivot block，以及當前目標區域的 block 載入 shared memory。
2. 利用 row pivot 與 column pivot 的資料，嘗試對目標 block 的每個元素進行更新：

```
for (int k = 0; k < BLOCK_WORK; k++) {
    // 使用 row pivot 與 column pivot 的資料計算更短路徑
    current[ty][tx] = min(current[ty][tx], row_pivot[ty][k] + col_pivot[k][tx
    ...
    // 這裡不需要 __syncthreads();
}
```

在這個迴圈中，我們不需要 `__syncthreads()`，因為 phase 3 更新的邏輯是 `current[ty][tx] = min(current[ty][tx], row_pivot[ty][k] + col_pivot[k][tx]);`，雖然看起來下一個迴圈會用到上一個迴圈的 current，但因為 row_pivot 與 col_pivot 都是 phase 2 就計

算好的固定的資料，所以迴圈執行的順序並不重要，所有迴圈執行完 current 就會更新為最小值。



hw3-3

hw3-3 與 hw3-2 的 kernel 實作完全一樣，流程如下：

1. 兩張 GPU 都配置一塊完整的距離矩陣。

2. 在每個回合 (round) 開始前，交換 pivot row：
   假設兩張 GPU 為 GPU0 與 GPU1。我會先判定目前回合的 pivot block row 是由哪張 GPU 所負責，若判定該 row 在 GPU0，就將該 row 從 GPU0 傳輸到 GPU1，反之亦然。

   - 例如：

   ```
   int pivot_owner = (r < gpu_data[0].end_block) ? 0 : 1;
   int other_gpu = 1 - pivot_owner;
   ...
   // 若 pivot row 位於 pivot_owner 這張卡上，就把它透過 cudaMemcpyPeerAsync 送給另
   cudaMemcpyPeerAsync(
       gpu_data[other_gpu].dist + pivot_row_start * padded_n,
       other_gpu,
       gpu_data[pivot_owner].dist + pivot_row_start * padded_n,
       pivot_owner,
       copy_size,
       gpu_data[pivot_owner].stream
   );
   ```

   因為這個 pivot row 在後面的 phase 1、phase 2、phase 3 都會被另外一張卡用到，所以必須先傳。

3. 同步等待傳輸完成，再進行 phase 1 與 phase 2：
   為了確保當另一張卡需要讀取 pivot block row 時，該 row 已經拷貝完畢，我透過 `cudaStreamSynchronize(...)` 在交換完成後做同步。

   - 完成後，phase 1 與 phase 2 就可以在兩邊 GPU 上各自執行：

- phase 1：更新當前 pivot block (兩張卡都做)
- phase 2：更新 pivot row 與 pivot column 上的其他區塊 (兩張卡都做)

4. phase 3 時，依照分配處理自己負責的 block rows ：
   為了分攤計算量，我將完整的 block rows 切成兩等份，讓 GPU0 負責前半段的 row blocks，GPU1 負責後半段的 row blocks。這部分並不需要持續交換中途計算的 block，因為所有需要的 pivot row、pivot column 資訊在前面階段已經更新並交換完畢。每張卡只要在進行更新前，讀取 local memory 裡對應位置的資料即可。

5. 最終合併結果：
   由於 GPU0 與 GPU1 都只保證它們各自負責的 row blocks 內計算結果正確，所以在所有回合 (round) 都結束後，必須分別將 GPU0 與 GPU1 負責的 row blocks 複製回 host 端 (CPU) 進行組合。

---

2. Profiling Results (hw3-2)

這邊我使用 `p11k1` 測資作為 input，對計算量最多的 `phase3_kernel`（占了 GPU 總運行時間的 96.2%）進行 profile，並利用 nvprof 收集以下效能指標（metrics），包含：

- **achieved_occupancy**
- **sm_efficiency**
- **gld_throughput** (Global Load Throughput)
- **gst_throughput** (Global Store Throughput)
- **shared_load_throughput** (Shared Memory Load Throughput)
- **shared_store_throughput** (Shared Memory Store Throughput)

實驗平台為 apollo-gpu-server (NVIDIA GeForce GTX 1080)

Profiling Metrics 結果

Metric	Min Value	Max Value	Avg Value
achieved_occupancy	0.921463	0.925500	0.923149
sm_efficiency	99.82%	99.92%	99.90%
gld_throughput	213.22 GB/s	216.02 GB/s	214.60 GB/s
gst_throughput	71.074 GB/s	72.007 GB/s	71.532 GB/s
shared_load_throughput	3482.6 GB/s	3528.4 GB/s	3505.1 GB/s
shared_store_throughput	284.29 GB/s	288.03 GB/s	286.13 GB/s

Global Memory 的存取模式分析

1. **Global Load 次數較多**：
   在程式中，kernel 一開始便將要進行 Floyd-Warshall 更新的資料 (row_pivot, col_pivot, current) 從 global memory 載入至 shared memory。如下列程式碼片段所示，每個 thread 負責將對應位置的資料讀取到 shared memory：

```
// Load from global memory to shared memory
row_pivot[ty][tx]      = dist[(block_y + ty) * const_n + (block_start + tx)];
row_pivot[ty][tx + 32] = dist[(block_y + ty) * const_n + (block_start + tx +
// ... 省略其他 row_pivot load ...

col_pivot[ty][tx]      = dist[(block_start + ty) * const_n + (block_x + tx)];
col_pivot[ty][tx + 32] = dist[(block_start + ty) * const_n + (block_x + tx +
// ... 省略其他 col_pivot load ...

current[ty][tx]        = dist[(block_y + ty) * const_n + (block_x + tx)];
current[ty][tx + 32]    = dist[(block_y + ty) * const_n + (block_x + tx + 32)]
// ... 省略其他 current load ...
```

2. **Global Store 次數較少**：
   完成所有的 Floyd-Warshall 更新計算後，kernel 最後只需要將更新後的結果 (current block) 回寫到 global memory：

```
dist[(block_y + ty) * const_n + (block_x + tx)]            = current[ty][tx];
dist[(block_y + ty) * const_n + (block_x + tx + 32)]       = current[ty][tx +
dist[(block_y + ty + 32) * const_n + (block_x + tx)]       = current[ty + 32][
dist[(block_y + ty + 32) * const_n + (block_x + tx + 32)] = current[ty + 32][
```

   在整個 kernel 中，每個 block 對 global memory 的寫回 (store) 發生僅在最終階段，相對於前面大量的 global load 行為，global store 的次數較少，約為 global load 的三分之一。此現象正好與 profile 結果中所顯示的 global load throughput 約為 global store throughput 的三倍相吻合。

Shared Memory 的存取模式分析

1. **Shared Memory Load 次數較多**：
   將資料載入到 shared memory 後，程式進入下列的迴圈進行 Floyd-Warshall 的核心更新：

```
for(int k = 0; k < BLOCK_WORK; k++) {
    current[ty][tx] = min(current[ty][tx], row_pivot[ty][k] + col_pivot[k][tx
    current[ty][tx + 32] = min(current[ty][tx + 32], row_pivot[ty][k] + col_p
    current[ty + 32][tx] = min(current[ty + 32][tx], row_pivot[ty + 32][k] +
    current[ty + 32][tx + 32] = min(current[ty + 32][tx + 32], row_pivot[ty +
}
```

在這個過程中，每次迭代都需要多次從 `row_pivot` 與 `col_pivot` 中讀取資料 (shared memory load)，並根據這些中繼值來更新 `current`。由於迴圈跑 `BLOCK_WORK` 次，而且每次都會對同一個 shared memory block 進行多次讀取，因此 shared memory load 的次數遠高於最初的寫入次數。

2. **Shared Memory Store 次數較少**：
   Shared memory 的初始化在前半段已完成，一旦將 global memory 資料讀入 shared memory 後，對 shared memory 的「寫入」只是在起始階段(將 global 載入 shared) 及部分更新 `current` 時發生。

   相較之下，`row_pivot` 與 `col_pivot` 在計算過程中僅需被讀取，不需再次回寫，故 shared memory 在計算階段主要以讀取為主。這也正好解釋了為何在 profiling 結果中，shared memory 的 load throughput 顯著高於 store throughput。

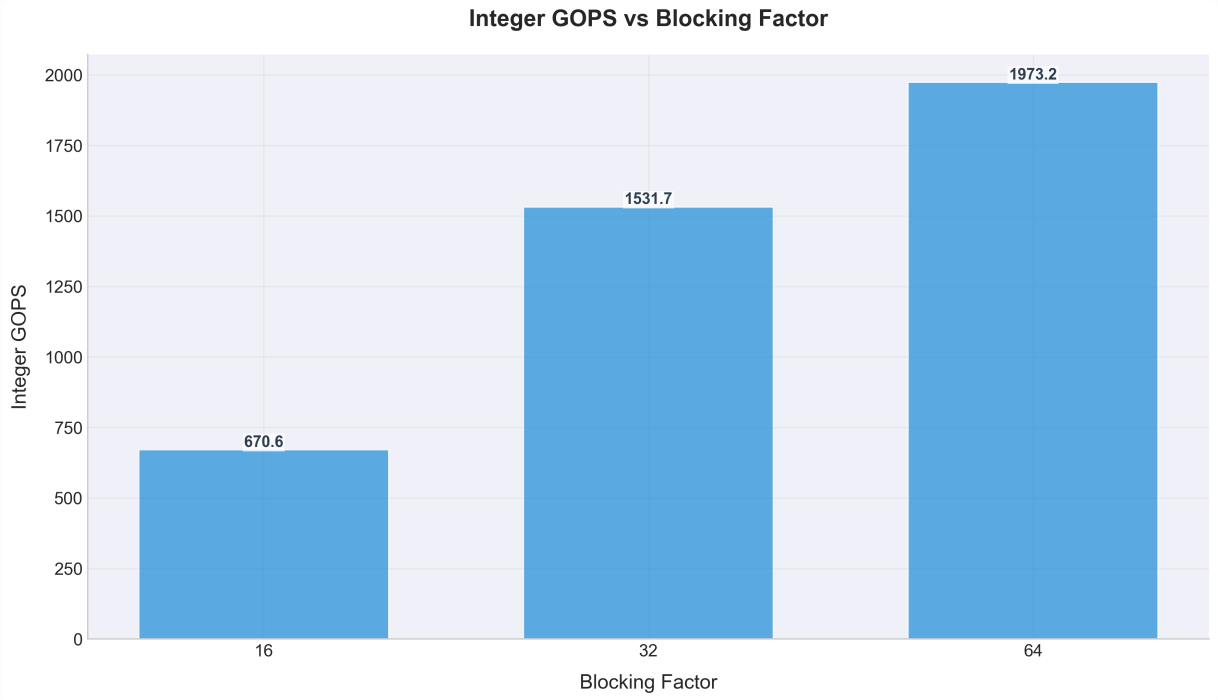
---

3. Experiment & Analysis

3.a System Spec

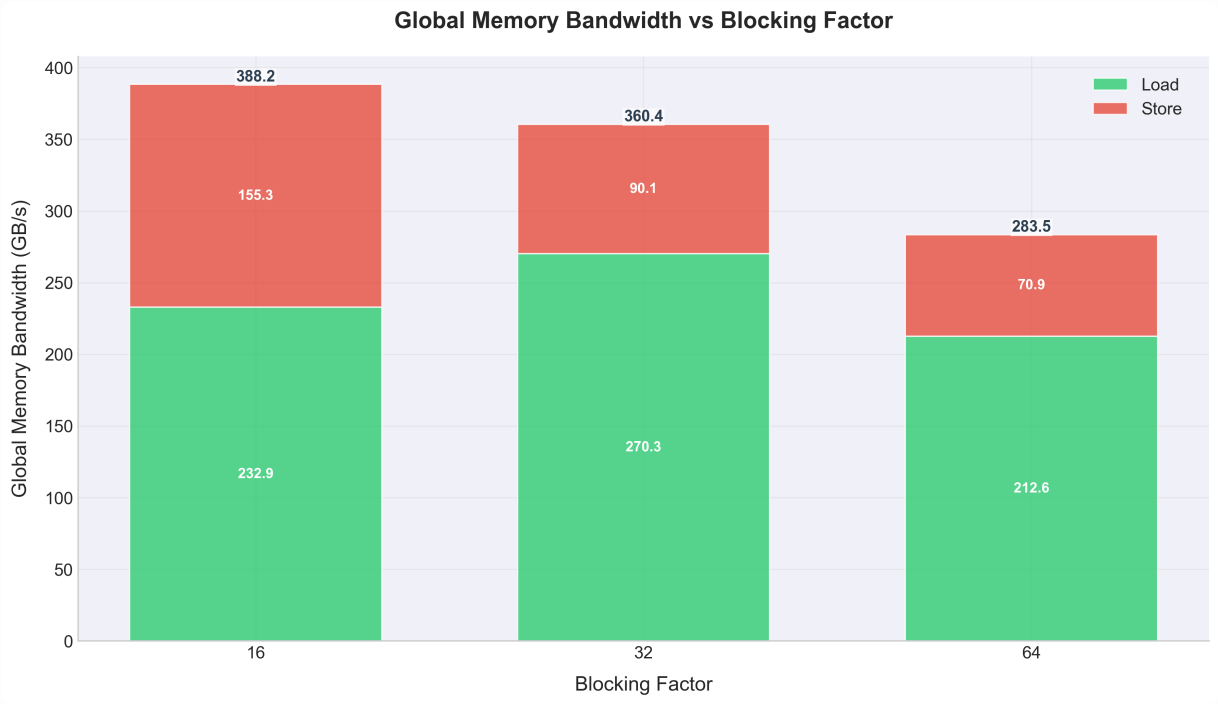
- 實驗平台為 apollo-gpu-server (NVIDIA GeForce GTX 1080)
- 選用測資：p11k1

3.b Blocking Factor (hw3-2)



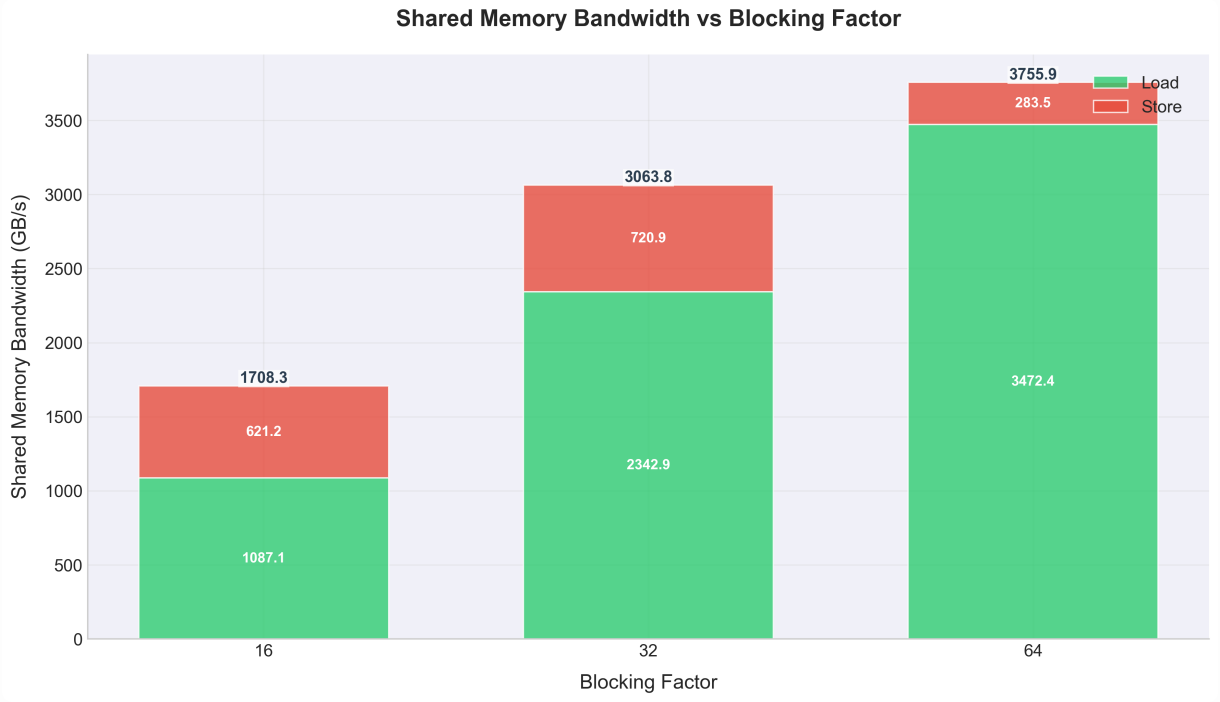
從以上圖表可以看到，隨著 blocking factor 的提升，integer GOPS 也隨之提高。這樣的結果顯示，適當增大 block 的維度能更有效地利用 GPU 上的運算資源，進而提高整體執行效率。雖然還

需考慮 register、shared memory 等資源的使用限制，但至少從這張圖來看，選擇較大的 blocking factor 確實能帶來加速的效果。



從這張圖可以發現，當我們把 blocking factor 調大後，整體的 global memory bandwidth 使用率下降了。在同一個測資的情況下，增加 blocking factor 後，理論上 kernel 每次存取的資料量應該是增加的，如果此時測得的 bandwidth（資料存取量 / 執行時間）反而下降，那就代表執行時間的增加幅度比資料量的增加還要大。換句話說，同樣多的資料，花了更長的時間去處理。

這樣的現象可能表示程式在增加 blocking factor 後，雖然整體資料量上來了，但 kernel 並沒有有效率地同時加速存取和運算，相反地可能因為排程、記憶體存取模式或硬體資源利用率的問題而引入更高的等待成本。結果是雖然「理應」讀寫更多資料，卻因為這些額外的開銷，導致實際的有效頻寬(每秒可處理的資料量)還下降了。



從圖中可以看到，隨著 blocking factor 增大，shared memory 的整體頻寬使用也隨之提高，尤其是 load 這部分明顯跳得很高。在同一個測資的情況下，增加 blocking factor，表示 kernel 一次要處理的資料量變多了（整體資料量，一個 thread 還是處理四筆資料），所以，到 shared memory 存取的量也變多了，在存取資料量變多的情況下，bandwidth 增加表示 kernel 執行時間上升的幅度比存取資料量上升的幅度還慢，表示 shared memory 使用效率有提升。

3.c Optimization (hw3-2)

在第一章 implementation 的部分我是直接描述優化過的程式碼的實作，這邊我會一步一步說明我是如何從一開始寫得可以過 correctness 的測資優化到最終的版本。

優化一

我重新規劃了在「Phase 2」階段所使用的 grid size，以避免對不需要處理的區塊進行無謂的計算。在原來的版本中，Phase 2 的 kernel 呼叫是針對整個 n×n 區塊網格執行，這表示即使是與中樞 (pivot) 區塊無關的行或列，也會被額外處理，徒增執行時間。經過修改後，我針對「行」與「列」兩種不同方向分開定義了較小範圍的 grid（針對行時使用 grid_row，針對列時使用 grid_col），讓程式在 Phase 2 階段只對相關的 row-block 或 col-block 進行計算，如此一來便能明顯減少整個計算過程中的多餘工作量。

```
// 原本對 grid 全域進行計算
phase2_kernel<<<grid, block>>>(dist_gpu, r, B, 1); // rows
phase2_kernel<<<grid, block>>>(dist_gpu, r, B, 0); // columns

// 修改後針對行與列分別定義合適的 grid 大小
phase2_kernel_row<<<grid_row, block>>>(dist_gpu, r, B);
phase2_kernel_col<<<grid_col, block>>>(dist_gpu, r, B);
```

優化二

在這個優化中，我做了兩件事情。首先，我將原始的 `n` 值 padding 到 `B` 的整數倍，。當 `n` 被 padding 成 `B` 的倍數後，所有 block 的處理量都一致，就能減少 kernel 中的 if branch，降低 warp divergence。此外，我也將原本使用 `malloc` 分配的 host-side 記憶體改為使用 `cudaHostAlloc` 配合 page-locked memory，這可加速 host 與 device 間的資料傳輸。

```
// 調整 n 為 B 的倍數
int padded_n = ((n + B - 1) / B) * B;

// 使用 cudaHostAlloc 分配 page-locked memory
cudaHostAlloc((void**)&Dist, padded_n * padded_n * sizeof(int), cudaHostAllocDefa
```

優化三

在這個優化中，我增加了每個 block 實際要處理的資料量。原本 block size 設為 32，一個 block 對應 32×32 的資料範圍，改成透過讓每個 thread 處理 4 筆資料，使得每個 block 可以處理的尺寸增加到 64×64，我們能在每次 kernel 呼叫時將更多資料載入 shared memory。這樣做的原因是：在 Phase 3 階段，我們必須同時載入三個 block 的內容到 shared memory，而使用 64×64 的配置後，剛好能填滿一個 block 最多可用的 shared memory 空間 (48 KB)，且提高 block size 會使得矩陣被切割為更大的 block，因而減少了 kernel launch 的次數。

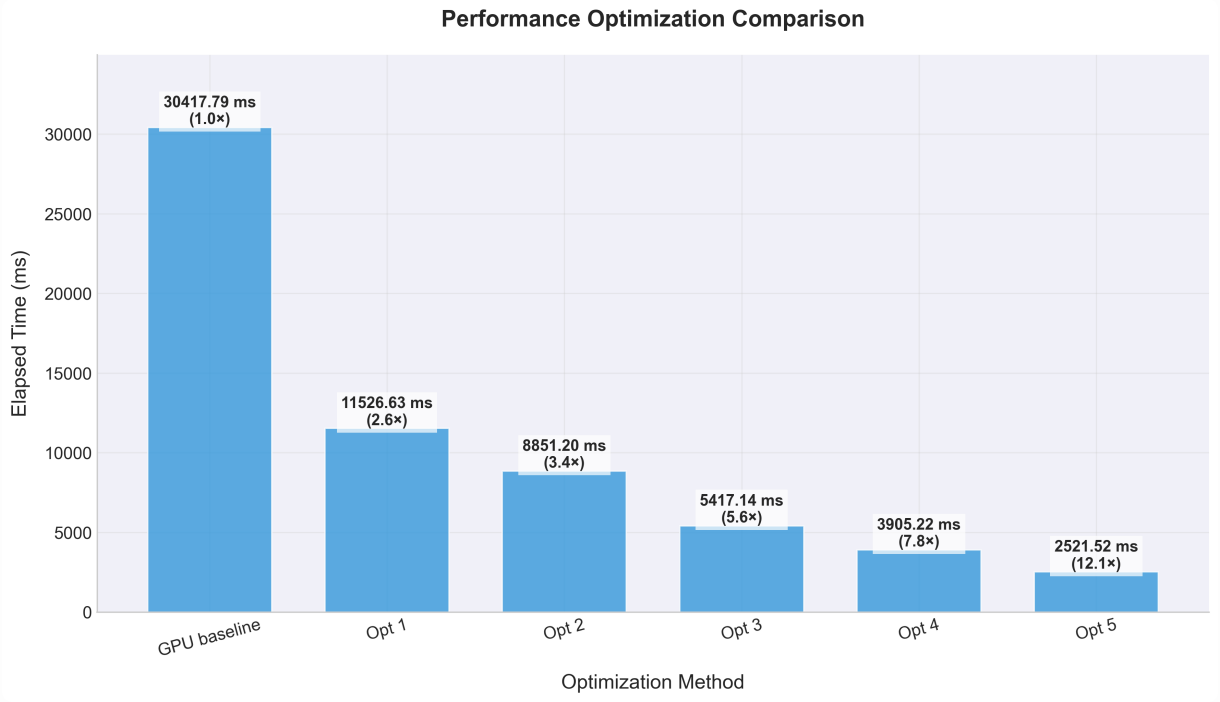
優化四

在這個優化中，我把 phase 3 每個 iterations 間的 `__syncthreads()` 拿掉。因為在這個迴圈中，其實不需要 `__syncthreads()`，因為 phase 3 更新的邏輯是 `current[ty][tx] = min(current[ty][tx], row_pivot[ty][k] + col_pivot[k][tx]);`，雖然看起來下一個迴圈會用到上一個迴圈的 current，但因為 row_pivot 與 col_pivot 都是 phase 2 就計算好的固定的資料，所以迴圈執行的順序並不重要，所有迴圈執行完 current 就會更新為最小值。

優化五

在先前的版本中，會透過 if 判斷來比較兩個路徑的距離並更新最短路徑距離。在這個版本中，我改用 min 函數去更新最短路徑值，這樣就不需要每次都判斷條件是否成立，避免 warp divergence 的發生。

```
// 使用 min 函數簡化更新:
block[ty][tx] = min(block[ty][tx], block[ty][k] + block[k][tx]);
```



這張圖片是我用 `p11k1` 這筆測資測試不同版本的 GPU 程式所花的時間以及 speedup 比較，可以看到每一個優化都減少了對同一筆測資的執行時間，原本我想加入 CPU 版本的執行時間，但會執行太久被 server kill 掉。

3.d Weak Scalability (hw3-3)

因為 blocked floyd-warshall 的時間複雜度是 $O(N^3)$，所以這邊我挑選 n^3 會相差兩倍的測資來做實驗，分別是 `c05.1` 跟 `c06.1`，`c05.1` 的 node 數為 80，`c06.1` 的 node 數為 100，

$$\left(\frac{100^3}{2}\right)^{\frac{1}{3}} = 79.37$$

所以兩者 node 數的三次方大約為兩倍。

- 測資規模：
  - `c05.1`：(N=80)，理論運算量約 (80^3 = 512,000)
  - `c06.1`：(N=100)，理論運算量約 (100^3 = 1,000,000) (約為前者 2 倍)
- 實驗結果：
  - hw3-2 執行 `c05.1` → 0.098 s
  - hw3-3 執行 `c06.1` → 0.209 s

在理想情況下，兩個程式執行的時間要相同，因為問題規模與 GPU 數量同步倍增，但實際上 hw3-3 的執行時間是 hw3-2 的 2.13 倍，其實也可以預期，因為我覺得我 hw3-3 的平行化沒有做得很好，應該要兩張 GPU 各分配一半的矩陣，然後 phase 1 ~ phase 3 都分開計算會比較理想，而且兩張 GPU 還有 communication 跟 synchronization 的開銷，本來就很難達到理想的加速，再加上我用來實驗的的測資很小，因為我這部分拖太晚才做，server 不知道為什麼在大測資上會

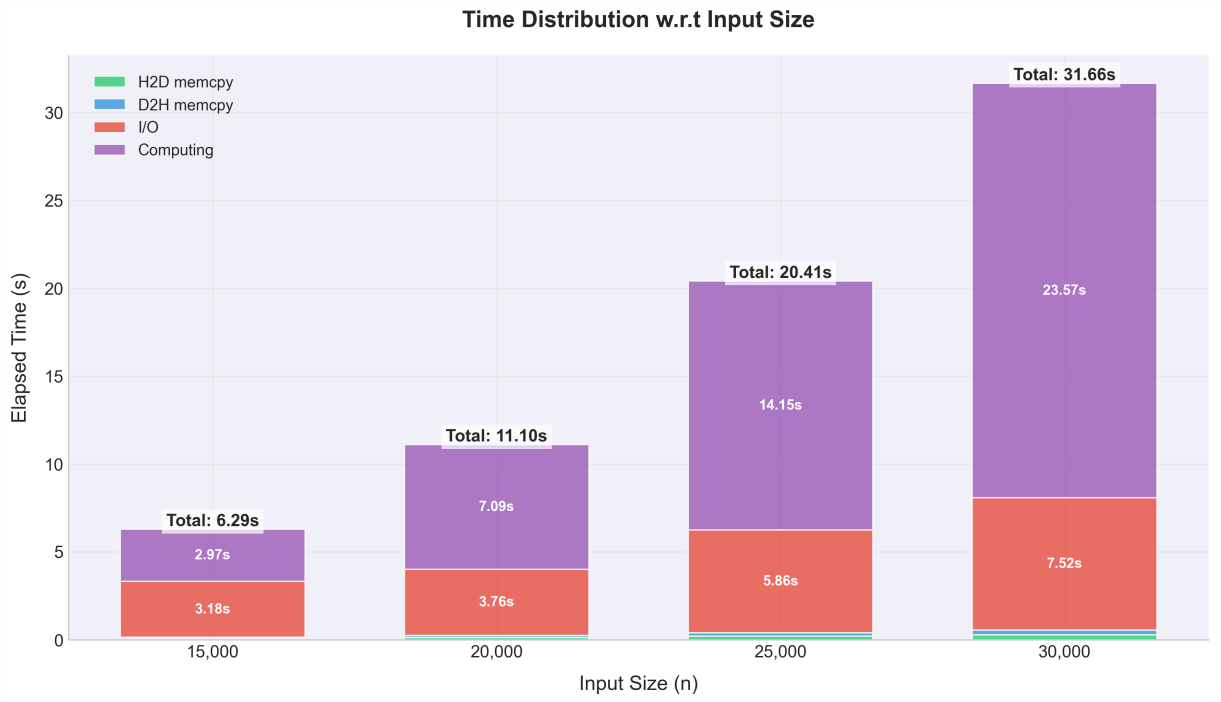
segmentation fault，在小測資上 communication 跟 synchronization 的開銷就相對佔比較大，無法輕易被平行化效益攤平。

3.e Time Distribution (hw3-2)

這邊我在程式中插入 `clock_gettime(CLOCK_MONOTONIC, ...)` 來量測不同階段的執行時間，並且適當地加入 `cudaDeviceSynchronize()` 以確保計時點的準確性。

1. I/O 時間：在 `input()` 函式讀檔前後、`output()` 函式寫檔前後記錄時間，以計算純 I/O 開銷。
2. Memory Copy (H2D, D2H) 時間：在 `cudaMemcpyHostToDevice` 與 `cudaMemcpyDeviceToHost` 前後記錄時間。
3. Computing 時間：包住 kernel 呼叫並在 kernel 呼叫後使用 `cudaDeviceSynchronize()`，以確保 kernel 真正執行完，再記錄時間。

Testcase	Input Size (n)	Edges (m)	H2D (s)	D2H (s)	I/O (s)	Computing (s)
p15k1	15000	5591272	0.075382	0.069414	3.179241	2.966898
p20k1	20000	264275	0.131889	0.122632	3.757107	7.087633
p25k1	25000	5780158	0.204518	0.191201	5.859598	14.151625
p30k1	30000	3907489	0.293983	0.275111	7.516197	23.570675



從以上圖表可以看出，隨著輸入規模的增加（從 15k 一路到 30k 的節點數），總執行時間明顯攀升，主要貢獻來自 I/O 及 computing。I/O 部分的成長不難理解，因為讀寫更大量的資料檔案本來就需要更多時間。而計算時間的增加更明顯，表示 Blocked Floyd-Warshall 演算法本身的複雜度使得計算時間相對於資料規模成倍增加。

H2D 和 D2H（Host 到 Device 與 Device 到 Host）傳輸的時間相對整體來說並不是主要瓶頸，即使隨著輸入增大也僅有小幅增加。這也顯示出 GPU 記憶體傳輸在這些測資下並非瓶頸，反而是演算法本身的計算階段和檔案讀寫在大規模資料下迅速主導了整體效能。從這張圖，可以清楚地看出：對於更大規模的測資，單純改善 I/O 或記憶體傳輸速度的效益有限，應該著重在優化 kernel 計算的部分。

---

4. Experiment on AMD GPU

hw3-2

```
c02.1    0.42   wrong answer: 336cf3ee1247b36994c9e3780f2876b2e11d507ae43dc28cb275c326357212a4,190
c03.1    0.37   wrong answer: b1ba34991a2618023e608f9eb8944683692213bdf6cde07dfde25be4d2ccec38,640
c01.1    0.36   wrong answer: 74af148e937fbf39aba0c24b7f67cb3249fa7f409d16d5392d429a89d4b75405,64
c04.1    0.37   wrong answer: ab73c73586d1919d68c0b333583913e602a81ed5e4a955ea0b3ac69436597203,1900
c05.1    0.37   wrong answer: ad5be2fe01925119931a53fe71ded655ae1d371f1b631b8a6268e28472a83fed,6400
c07.1    0.37   wrong answer: 2f85579b4940d2d5c446d8c995b259056e43431bbf277a80b6ecc2f9744324ef,19000
c08.1    0.42   wrong answer: 6fcf899048bb72a72165f53ff0b308d3d729cd3f71283756bed522ee6f9c8447,64a04
c09.1    0.37   wrong answer: 5f9c4cbf7fc620941bea3b521a2783503300e185285da579557e5043ac77d872,64000
c10.1    0.42   wrong answer: 2c2ee49cbc3a952f2b171ac0097b7e66bbba38b54ae23d0de8ffdfb36a4c165c,f4240
c11.1    0.46   wrong answer: f3c1ff470de0c0409ef77b8079abfc97e86056f2460f36845f7b3a74c291088e,12cce4
c12.1    0.47   wrong answer: 72d767086a507cf8a4c39d3bc8a4176dd01742926382cea8063f68ea07f9f75a,f4240
c06.1    0.37   wrong answer: 5c8dace081e890ba40364ab80572daa05d3d725b3a2e1a49c6663cf2c6dc7e52,9c40
c15.1    0.62   wrong answer: 3c09bb0b82cc78ec176e58359eb8251eba5fdec90c925584728a1b47d8f7e232,24d944
c13.1    0.52   wrong answer: 29157e297c49ccd874e912549b1b99a879dc9873f28d19a9d06f7f9f2752aed1,12cce4
c14.1    0.67   wrong answer: 6e9e03ba64494f447b8d03618a9cd1d439b4a54aaae10638396ee7aaa6938978,3ce9c4
c17.1    1.67   wrong answer: 3506db7ddffdc479c9d41a93c9c220c0d7914e2a669826d6eadc8ae6acaf3b5f,10d2a40
c16.1    1.07   wrong answer: 0fc20bf78abd03a1b6d79e56cd4bfd0c06d4790a3785dbf85302539973c4bc5b,895440
c18.1    2.82   wrong answer: d84de378d021693f6c4208468eb28f84bcefcc21cb5f1ef249f5c99c8dda4760,2255100
c19.1    2.22   wrong answer: 3b5e92870ac9283fc8ec4d75e44345cec3c9c8f545ac1fbfdaded40eb8afdb8d,10d2a40
c20.1    6.53   wrong answer: 2e99d0e204ab7c017b0223990b87352281c5ab0b49af11ceb6bdfa52f14794ab,5f5e100
c21.1    7.38   wrong answer: 54260c7c21eb53f5b3c20ad3778f363c4205308daa6d9fcc7d68ac1d38185833,5f5e100
p11k1   19.73   wrong answer: 083c5d61442f6bbe8835fb2515b0259090765af2e12b424390ef176a201d7310,1cd94100
p12k1   33.15   time limit exceeded:  {timeout}
p13k1   26.69   wrong answer: 900002dce9b84059874fc7b7f8f19151f527972e7e15a0188caedb7b40570185,284af100
p14k1   40.00   time limit exceeded:  {timeout}
p15k1   36.41   time limit exceeded:  {timeout}
```

我把我在 GTX 1080 上最快的版本改成 AMD 版本後變成 wrong answer，而且速度慢很多，可能是轉換過程中出了一些問題。

hw3-3

```
make: Leaving directory '/share/judge_dir/.judge_exe.3473172532'
c03.1    0.97   accepted
c02.1    0.62   accepted
c01.1    0.62   accepted
c04.1    7.43   time limit exceeded:  {timeout}
c05.1   18.00   time limit exceeded:  {timeout}
c06.1   90.00   time limit exceeded:  {timeout}
c07.1  110.00   time limit exceeded:  {timeout}
```

hw3-3 的部分看起來有過正確性，但跑得比較慢所以後面幾個測資 tle 了，不確定是我的寫法在 AMD GPU 上會跑比較慢還是 AMD 本身的硬體就比較慢的關係，我猜應該是前者。

---

5. Experience & Conclusion

透過這個作業我第一次學到了如何寫一個 cuda 程式，我覺得光要寫對就不容易了，因為資料要怎麼 mapping 到 GPU 上要想得很清楚，如果要進一步優化也要很理解 GPU 的架構跟 profiling 的方式，總之學到很多。