HW3: All-Pair Shortest Path

107062223 歐川銘

Implementation

Which algorithm do you choose in hw3-1?

在 hw3-1 的實作中，我直接採用的是 Floyed-Warshall 的作法，除了最外面那層 k 有 data dependency 無法做平行化之外，我採用的是 OpenMP，將裡面的雙迴圈讓其他 thread 去分工，同時對 Dist 這個陣列的值去做計算，最後得到結果。

```
// 除了 k 外層迴圈之外，裡面用 omp 讓 thread 去做切分
for (int k = 0; k < n; k++) {
  #pragma omp parallel for schedule(static)
    for (int i = 0; i < n; i++) {
      #pragma unroll 4
      for (int j = 0; j < n; j++) {
        if (Dist[i][k] + Dist[k][j] < Dist[i][j]) {
          Dist[i][j] = Dist[i][k] + Dist[k][j];
        }
      }
    }
  }
```

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

在 hw3-2 跟 hw3-3 中我都是採用一樣的方式去切割 data，為了善用 share memory，我將 block size 設成 64*64，為了避免取值會有 out of bond 的問題，我在一開始如果 n 不是 64 的倍數的話，我會另外設定 N 代表離 n 最近的 64 倍數，這樣之後在運算的時候最多就是讓這些多出來的點進行空運算，這

樣做的好處就是不用考慮陣列取值有 out of bound 的問題，以及在 kernel 裡面也不會有過多的 branch 發生導致降低 performance。

```
#define B 64

// Set N size
if (n % B) N = n + (B - n % B);
else N = n;

// malloc Dist size
Dist = (int*)malloc(N*N*sizeof(int));
```

What is your configuration in hw3-2, hw3-3? Any why?

在上面的問題有提到說我為了要善用 share memory 所以將 block size 設成 64*64，這是因為我在 kernel 裡面最多會用到 64*64*3*sizeof(int) = 49152 Byte 的 share memory，但是因為每個 block 最多只能有 1024 個 thread，所以我每個 block dimension 都設定成 dim3(32, 32) 但是裡面每個 thread要計算 64*64 大小的 block，也就是說每個 thread 每次要去更新 4 個點的值，也因為我的每個 block size 是 64*64，所以我 grid dimension 都設定成 dim3(N/64, N/64)，以此來計算所有的點更新值。

```
int blocks = (N + B - 1) / B;
  dim3 block_dim(32, 32);
  dim3 grid_dim(blocks, blocks);
  for (int r = 0; r < round; ++r) {
    // phase 1
    phase_one<<<1, block_dim>>>(dst, r, N);
    // phase 2
    phase_two<<<blocks, block_dim>>>(dst, r, N);
    // phase 3
    phase_three<<<grid_dim, block_dim>>>(dst, r, N);
  }
```

How do you implement the communication in hw3-3?

我在溝通的時候是透過 Device 之間的 Memory Copy 讓資料從 GPU 複製到另一個 GPU 上，在 hw3-3 裡面我一樣採用 OpenMP 讓每個 thread 負責控制一個 GPU，然後兩個 thread 會共享一個array 裡面存的是 Dist 在個別 GPU 上的位置，然後每個 thread 各自去 set 他們負責的 device GPU，以及去 malloc 在 GPU 上所需要的位置大小，然後呼叫 `cudaDeviceEnablePeerAccess` 允許其他 device 可以把資料複製過來。

```
#pragma omp parallel num_threads(2)
  {
    // get thread number
    unsigned int cpu_thread_id = omp_get_thread_num();
    // thread neighbor number
    unsigned int cpu_thread_id_nei = !cpu_thread_id;

    // thread set its device and malloc same memory in the device
    cudaSetDevice(cpu_thread_id);
    cudaMalloc(&dst[cpu_thread_id], N*N*sizeof(int));

    // 檢查 GPU 負責的起始位置跟總負責的大小
    unsigned int start_offset = (cpu_thread_id == 1) ? round / 2 : 0;
    unsigned int total_row = round / 2;
    if (round % 2 == 1 && cpu_thread_id == 1) total_row += 1;

    // set grid dimension
    dim3 grid_dim(blocks, total_row);

    // 設定 GPU 在 dist 的起始位置跟總Byte數量
    unsigned int dist_offset = start_offset * N * B;
    unsigned int total_byte_num = total_row * N * B * sizeof(int);
    unsigned int one_row_byte_num = B * N * sizeof(int);

    // 將資料從 Dist copy 到 GPU 上
    cudaMemcpy(dst[cpu_thread_id] + dist_offset, Dist + dist_offset, total_byte_n
    // 允許另一個 devise copy 資料給自己
    cudaDeviceEnablePeerAccess(cpu_thread_id_nei, 0);
    #pragma omp barrier

    for (int r = 0; r < round; ++r) {
      unsigned int start_offset_num = r * B * N;
      if (r >= start_offset && r < (start_offset + total_row)) {
        cudaMemcpy(dst[cpu_thread_id_nei] + start_offset_num, dst[cpu_thread_id]
```

```
      }
      // 要等到資料都拿好才開始計算
      #pragma omp barrier
      phase_one<<<1, block_dim>>>(dst[cpu_thread_id], r, N);
      phase_two<<<blocks, block_dim>>>(dst[cpu_thread_id], r, N);
      phase_three<<<grid_dim, block_dim>>>(dst[cpu_thread_id], r, N, start_offset
    }
    // 把資料 copy 回 Dist
    cudaMemcpy(Dist + dist_offset, dst[cpu_thread_id] + dist_offset, total_byte_n
  }
```

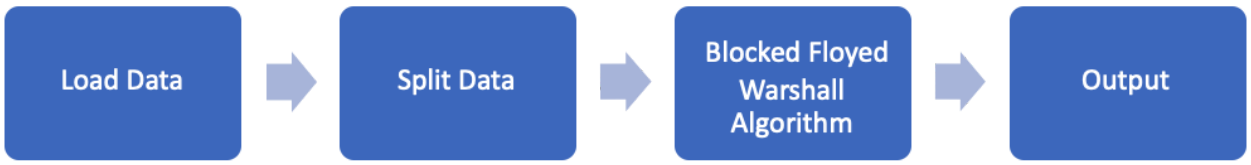
Briefly describe your implementation

CPU Version

1. `Load Data`: 首先我先將資料從 file 裡面讀到 Dist 裡面，我會在全域變數裡面開好一個 50000*50000 大小的 Dist，然後再把資料 load 到裡面去，讓所有的 Thread 都可以共用這個 Dist

2. `Floyed Warshall with OpenMP`: 我在 HW3-1 實作的方式是採用 `Floyed Warshall`，因為 `Floyed Warshall` 最外層的迴圈有 data dependency，所以無法平行化，所以我只有裡面的雙迴圈讓 thread 去做分工，讓每個 pair 去更新經過 k 點的值
3. `Output`: 最後將 Dist 的資料參考 HW3-2 寫回去的方式將資料寫回到 output file 裡面

Single-GPU Version



1. `Load Data`
   - 在 Load Data 的地方我更改了一下讀取 data 的方式，我是使用 mmap function 將 file 資料 map 到 memory 上再把資料 Load 到 Dist 裡面

```
int *pair = (int*)(mmap(NULL, (3 * m + 2) * sizeof(int), PROT_READ, MAP_PRIVA
```

- 為了讓資料處理時更方便，如果 n 大小不是 Block Size 的倍數，我會設置一個 N 為最接近 n 64 倍的數字，多出來的點就讓 thread 做空運算，這樣做的會大幅減少在 kernel 裡面的 branch

```
if (n % B) N = n + (B - n % B); // B = Block Size
else N = n;
```

2. `Split Data`
    - 根據自己設定的 Block Size 將 data 分成 `(N/Block_Size, N/Block_Size)` 數量個 blocks，這樣每個 block 的所有 thread 要負責 Block Size 大小的資料，然後每個在每個 phase 就對需要的資料進行更新，所以 Grid Dimension Size 是 `(N/Block, N/Block)` 然後每個 block 裡面負責 `(Block Size, Block Size)` 的資料
3. `Blocked Floyed Warshall Algorithm`

Phase 1

- 每個 round 的 phase1 就是對現在在 `pivot` `(round, round)` 裡面的資料做一般的 Floyed Warshall，代表對現在這個 Blocked 先做更新，在這邊我的做法是用 `32*32` 數量的 thread 去更新 `64*64` 大小的資料，所以我的 block dimension 都是 `dim3(32, 32)` ，一次只會有一個 block 去做更新

```
int blocks = (N + B - 1) / B; // 代表多少個 block
dim3 block_dim(32, 32);
dim3 grid_dim(blocks, blocks);

phase_one<<<1, block_dim>>>(dst, r, N);
```

- 所以我的每個 thread 都必須要更新 4 個點來把 shared memory 的空間盡量塞滿，為了 `coalesced memory` 跟避免 `bank conflict` 問題，每個 thread 要負責搬移的點也要仔細思考，對於 `thread(0, 0)` 而言，這個 thread 就必須要搬在 shared memory `(0, 0)`, `(32, 0)`, `(0, 32)`, `(32, 32)` 這四個點的資料，後面要取值時候就可以直接在 shared memory 裡面存取，記得最後每個 thread 搬完資料的時候都要加上 `__syncthreads()` 等待所有的 thread 完成搬移，確保之後的計算存取到的資料都可以正確

```
s[i][j] = dst[place];
s[i][j_B] = dst[place_right];
s[i_B][j] = dst[place_down];
s[i_B][j_B] = dst[place_down_right];
__syncthreads();
```

- 最後就是針對資料進行更新，因為需要的資料都已經 load 到 shared memory，所以只需要在 shared memory 更新完結果再寫回 global memory

```
for (int k = 0; k < B; ++k) {
    s[i][j] = Min(s[i][k] + s[k][j], s[i][j]);
    s[i][j_B] = Min(s[i][k] + s[k][j_B], s[i][j_B]);
    s[i_B][j] = Min(s[i_B][k] + s[k][j], s[i_B][j]);
    s[i_B][j_B] = Min(s[i_B][k] + s[k][j_B], s[i_B][j_B]);
    __syncthreads();
  }
  dst[place] = s[i][j];
  dst[place_right] = s[i][j_B];
  dst[place_down] = s[i_B][j];
  dst[place_down_right] = s[i_B][j_B];
```

Phase 2

- phase2 會把跟 `pivot` 在同個 `row`, `col` 上的所有 blocks 來做更新，所以扣掉 pivot 的話總共會有 2 * (N / Block size) - 1 個 blocks

- 為了妥善運用 shared memory，所以我把 row 跟 col 的 blocks 同時一起更新，也就是說總共只需要 N/Block Size 個 blocks 來做就好

```
int blocks = (N + B - 1) / B;
dim3 block_dim(32, 32);

phase_two<<<blocks, block_dim>>>(dst, r, N);
```

- 所以在 phase 2 裡面，除了要搬需要被更新點的資料，也要負責把 pivot 資料從 global memory 一起搬到 shared memory，每個 thread 一樣要負責四個點

```
__shared__ int s[B][B];
__shared__ int ver[B][B];
__shared__ int hor[B][B];

s[i][j] = dst[diagonal_place];
s[i][j_B] = dst[diagonal_place_right];
s[i_B][j] = dst[diagonal_place_down];
s[i_B][j_B] = dst[diagonal_place_down_right];

ver[i][j] = dst[ver_place];
ver[i][j_B] = dst[ver_place_right];
ver[i_B][j] = dst[ver_place_down];
ver[i_B][j_B] = dst[ver_place_down_right];

hor[i][j] = dst[hor_place];
hor[i][j_B] = dst[hor_place_right];
hor[i_B][j] = dst[hor_place_down];
hor[i_B][j_B] = dst[hor_place_down_right];

__syncthreads(); // 確定大家都已經 load 資料
```

- 最後一樣對這些點進行更新，並且寫回到 global memory 代表這些點都已經更新好了

```
for (int k = 0; k < B; ++k) {
    ver[i][j] = Min(ver[i][j], ver[i][k] + s[k][j]);
```

```
    ver[i][j_B] = Min(ver[i][j_B], ver[i][k] + s[k][j_B]);
    ver[i_B][j] = Min(ver[i_B][j], ver[i_B][k] + s[k][j]);
    ver[i_B][j_B] = Min(ver[i_B][j_B], ver[i_B][k] + s[k][j_B]);

    hor[i][j] = Min(hor[i][j], s[i][k] + hor[k][j]);
    hor[i][j_B] = Min(hor[i][j_B], s[i][k] + hor[k][j_B]);
    hor[i_B][j] = Min(hor[i_B][j], s[i_B][k] + hor[k][j]);
    hor[i_B][j_B] = Min(hor[i_B][j_B], s[i_B][k] + hor[k][j_B]);

    __syncthreads();
  }

  dst[ver_place] = ver[i][j];
  dst[ver_place_right] = ver[i][j_B];
  dst[ver_place_down] = ver[i_B][j];
  dst[ver_place_down_right] = ver[i_B][j_B];

  dst[hor_place] = hor[i][j];
  dst[hor_place_right] = hor[i][j_B];
  dst[hor_place_down] = hor[i_B][j];
  dst[hor_place_down_right] = hor[i_B][j_B];
```

phase 3

- 有了更新好的 row, col, 以及 pivot 的資料之後，phase3 就是用更新好的 blocks 對剩下還沒有更新到 blocks 進行更新，所以總共會有 `(n/round, n/round)` 數量的 blocks

```
int blocks = (N + B - 1) / B;
dim3 block_dim(32, 32);
dim3 grid_dim(blocks, blocks);

phase_three<<<grid_dim, block_dim>>>(dst, r, N);
```

- phase3 花費的時間比其他兩個 phase3 還要久，這是因為大多數的 blocks 都是 phase3 才做更新。
- 一樣會把需要的資料從 global memory load 進來，並且整個 blocks 裡面的 thread 都必須要等到其他 thread 都搬好資料才可以開始計算

```
__shared__ int self[B][B];
__shared__ int a[B][B];
__shared__ int b[B][B];

self[i][j] = dst[self_place];
self[i][j_B] = dst[self_place_right];
self[i_B][j] = dst[self_place_down];
self[i_B][j_B] = dst[self_place_down_right];

a[i][j] = dst[a_place];
a[i][j_B] = dst[a_place_right];
a[i_B][j] = dst[a_place_down];
a[i_B][j_B] = dst[a_place_down_right];

b[i][j] = dst[b_place];
b[i][j_B] = dst[b_place_right];
b[i_B][j] = dst[b_place_down];
b[i_B][j_B] = dst[b_place_down_right];

__syncthreads();
```

- 最後在一起 load 回 global memory

```
for (int k = 0; k < B; ++k) {
    self[i][j] = Min(a[i][k] + b[k][j], self[i][j]);
    self[i][j_B] = Min(a[i][k] + b[k][j_B], self[i][j_B]);
    self[i_B][j] = Min(a[i_B][k] + b[k][j], self[i_B][j]);
    self[i_B][j_B] = Min(a[i_B][k] + b[k][j_B], self[i_B][j_B]);
  }
  dst[self_place] = self[i][j];
  dst[self_place_right] = self[i][j_B];
  dst[self_place_down] = self[i_B][j];
  dst[self_place_down_right] = self[i_B][j_B];
```

4. `Output`

- 最後先把 global memory 的資料更新回 host 上

```
cudaMemcpy(Dist, dst, N*N*sizeof(int), cudaMemcpyDeviceToHost);
cudaFree(dst);
```

- 然後再把資料寫回到 file

```
inline void output(char* outFileName) {
  FILE* outfile = fopen(outFileName, "w");

  #pragma unroll 32
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      if (Dist[i*N+j] >= INF) Dist[i*N+j] = INF;
    }
    fwrite(&Dist[i*N], sizeof(int), n, outfile);
  }
  fclose(outfile);
}
```

Multiple GPU

- 我在 Multiple GPU 的實作跟 Single GPU 的實作一樣，差別只差在 phase 3 以及 device 之間處理資料的傳遞，所以在 Multiple GPU 這邊主要講解怎麼讓兩個 GPU 去處理的實作
- 我的概念是兩個 device 在 phase1 跟 phase2 的會執行一模一樣的操作，然後在花費時間做多的phase3 兩個 device 各做一半，所以在每個 round 結束之後，兩個 device 都需要互相交換資料，保證讓對方接下來的操作以及負責的資料是正確的

1. `malloc memory in different device`
    - 我是使用 OpenMP 讓每個 thread 負責一個 device，然後個別去幫這個 device malloc 所需要的空間

```
// thread set its device and malloc same memory in the device
cudaSetDevice(cpu_thread_id);
cudaMalloc(&dst[cpu_thread_id], N*N*sizeof(int));
```

2. `Enable access to the device`

   - 要設定允許讓另一個 device 可以把資料 copy 到自己的 device 上
   - 另外把在 host 上的資料搬到 device 上
   - 並且要設定 barrier，不然先完成的 device 會先去執行 kernel 導致拿到的資料可能不齊全

```
cudaMemcpy(dst[cpu_thread_id] + dist_offset, Dist + dist_offset, total_byte_r
cudaDeviceEnablePeerAccess(cpu_thread_id_nei, 0);
#pragma omp barrier
```

3. `Copy memory to the other device`

   - 為了要結省 memory 搬移的時間，所以交換資料的時候可以搬越少資料越好
   - 兩個 device 個負責一半資料，只要確定那一半資料的正確性，之後再把這些資料寫回到 host 上面，所以每次傳輸資料的時候，只需要 device 傳一個 `Block Size*Block Size` 大小的資料給另一個 device 就好，因為這樣就可以確保另一個 device 負責的結果一定是正確的。

```
// 先知道哪個 device 是負責現在這個 round*round 位置的 block
unsigned int start_offset_num = r * B * N;

// 把這個 block 裡面的資料傳給另外一個 device
if (r >= start_offset && r < (start_offset + total_row)) {
  cudaMemcpy(dst[cpu_thread_id_nei] + start_offset_num, dst[cpu_thread_id] +
}

// 加上 barrier 兩個 device 的資料都是正確 copy 完成的才開始執行 kernel function
#pragma omp barrier
```

4. `phase 3`

- 因為每個 device 只負責 phase 3 一半的計算量，所以 phase 3 kernel 只需要 single GPU 一半的 block 數量來計算

```
// 每個 device 要計算的起始位置跟總共要負責計算多少大小的資料
unsigned int dist_offset = start_offset * N * B;
unsigned int total_byte_num = total_row * N * B * sizeof(int);
```

- 然後在 phase 3 裡面加上 start_offset，讓 device 計算到要負責的部分

```
// block_y 代表每個 thread 實際要計算的 block 位置
int block_y = blockIdx.y + row_offset;
int i = threadIdx.y;
int j = threadIdx.x;
```

Profiling Results

在 Profiling Result 裡面，我使用 p19k1 測資來測試我的 code 在 `phase3` 所執行的 kernel function

Metrics	Min	Max	Average
Occupancy	0.939905	0.942931	0.941839
Sm Efficiency	99.92%	99.98%	99.98%
Shared Memory Load Throughput	3322.6GB/s	3477.7GB/s	3430.2GB/s
Shared Memory Store Throughput	276.08GB/s	285.98GB/s	283.25GB/s
Global Load Throughput	18.600GB/s	19.011GB/s	18.690GB/s
Global Store Throughput	68.274GB/s	70.805GB/s	69.807GB/s

Experiment & Analysis

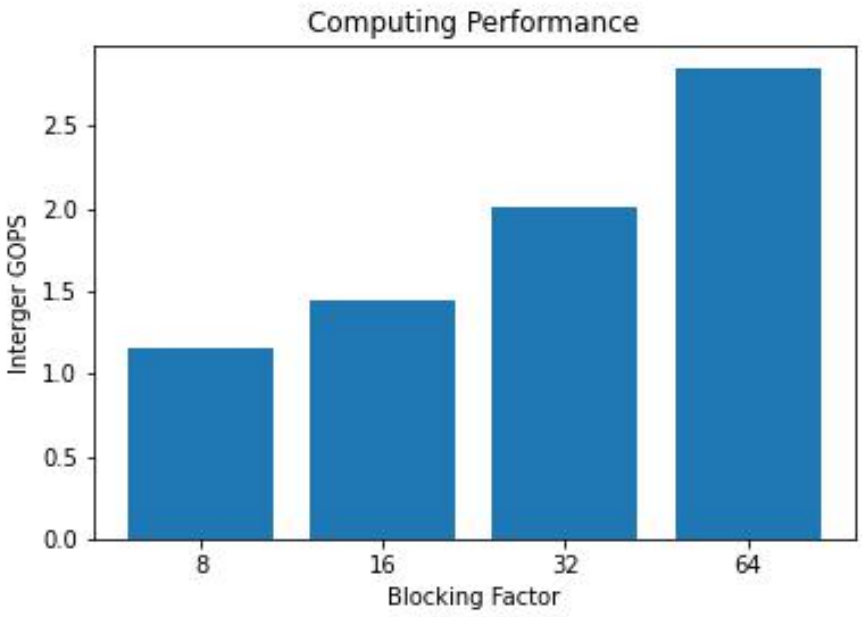
System Spec

使用課程提供的 hades 機器來做測試

Blocking Factor

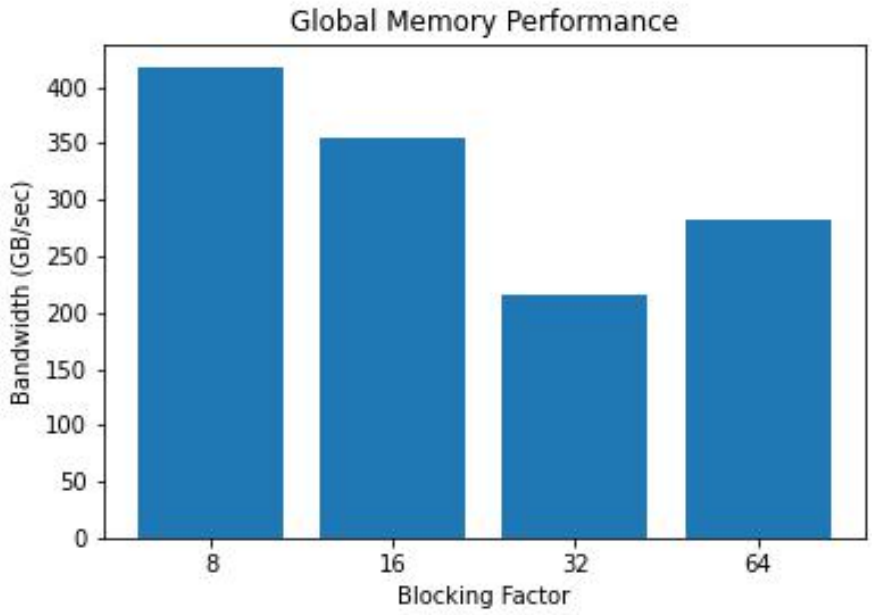
在這邊要計算在不同 Blocking Factor 下 phase3 的 Global/Shared memory 的 Bandwidth 跟 GOPS，實驗中我使用 c20.1 testcase 來進行實驗。

1. `Integer GOPS`
    - 計算在不同 Blocking Factor 下執行的 `integer operation` 數量
    - 下圖可以看出 Blocking Factor 到達 64 的時候 performance 最好，沒有在往上增加的原因是因為 128 大小實作上 shared memory 會不夠使用，因此 `64` 為我挑選出一個最好的 Blocking Factor

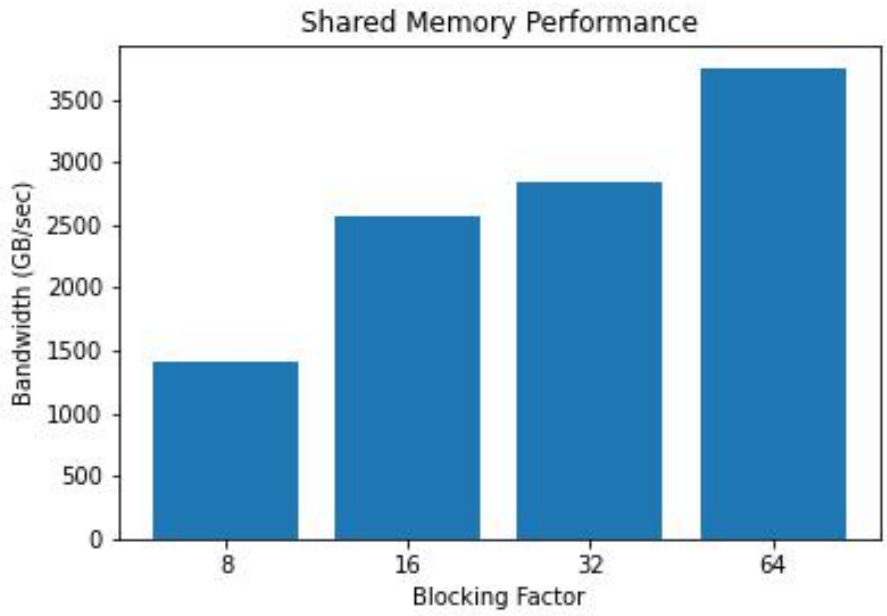


2. `Global Bandwidth`

- 觀察不同 Blocking Factor 下的 Global memory 的 bandwidth，在這邊我把 `global load` operation 跟 `global store` operation 相加的數字當作結果當作 `global bandwidth`
- 下圖因為每個 block 最多只能 launch 1024 個 kernel，為了善用 shared memory，所以我在 Blocking Factor 64 的時候讓每個 thread 負責 load 4 個點的 global memory access



3. `Shared Bandwidth`
   - 觀察不同 Blocking Factor 下的 shared memory 的 bandwidth，跟上面一樣我把 `shared load` operation 跟 `shared store` operation 相加當作 `shared bandwidth`
   - 從下圖可以看得知 Blocking Factor 越大，shared memory 的 performance 就越好



Optimization

在這次作業中，如果沒有使用到一些優化技巧，後面的測資基本上都不會過，所以底下逐一講我在這次作業裡所用到的一些優化技巧，並且用 p11k1 的 testcase 來做測試比較不同優化版本的 performance

1. `No Optimization`

   首先是只有使用 GPU 的版本，一開始先只有從助教提提供的 seq code 改成看以在 GPU 上面執行的版本，在 blocking fatcor 為 32 並且資料運算都是從 Global memory 拿取做運算的情況下，還有很長的一段路要走，但是至少會比 CPU 的 performance 好上非常多

2. `Coalesced Memory`

   接下來考慮到 thread 在 access/load global memory 的時候要考慮到記憶體連續存取的問題，這樣的話可以更好的善用 cache，減少每次都需要去 global memory 拿取資料的狀況

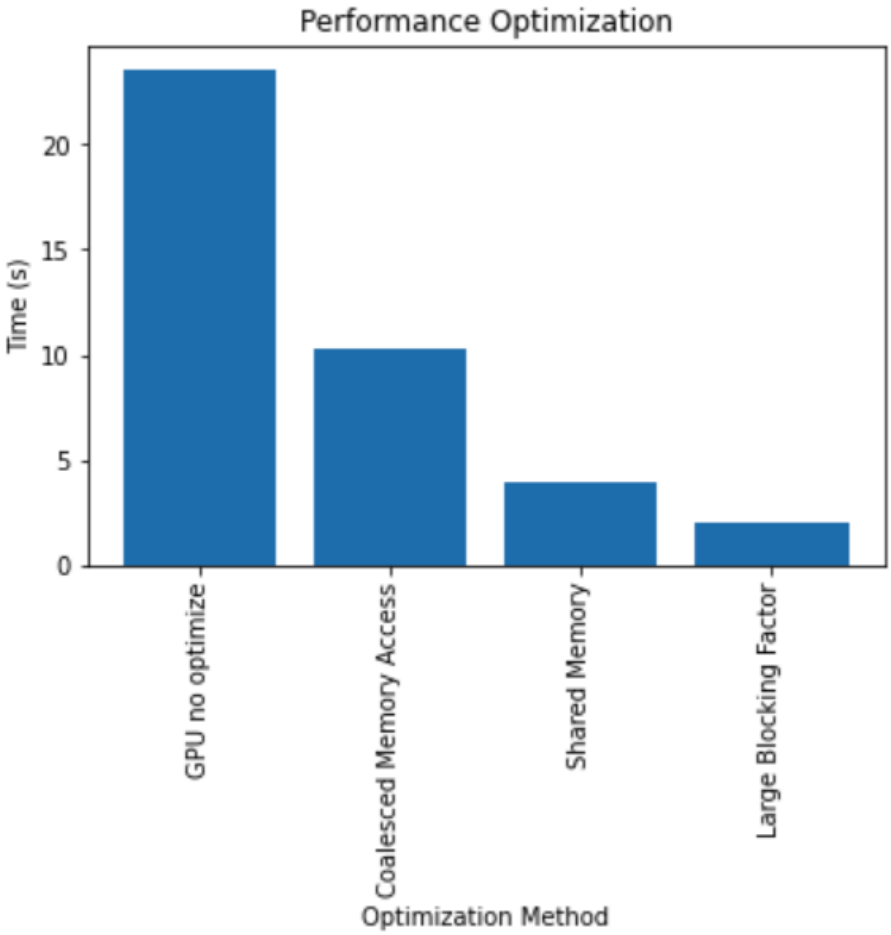
3. `Shared Memory`

因為即使有 cache 的幫助，但是還是會經常 access 到 global memory，所以如果先讓每個 thread 把需要用到的資料 load 到 shared memory 的話，利用 shared memory 的 access 速度比 global memory 更快這一點，可以節省非常多時間

4. `Larger Blocking Factor`

   一開始 Blocking Size 因爲一個 block 最多只能有 1024 個 thread，所以我只有設定成 32*32，但是這樣如果這樣會無法善用 shared memory，因此我將 Blocking Size 設定成 64*64，也就是讓每個 thread 一次要負責 4 個點得運算，這樣一來可以更加善用 shared memory 的空間

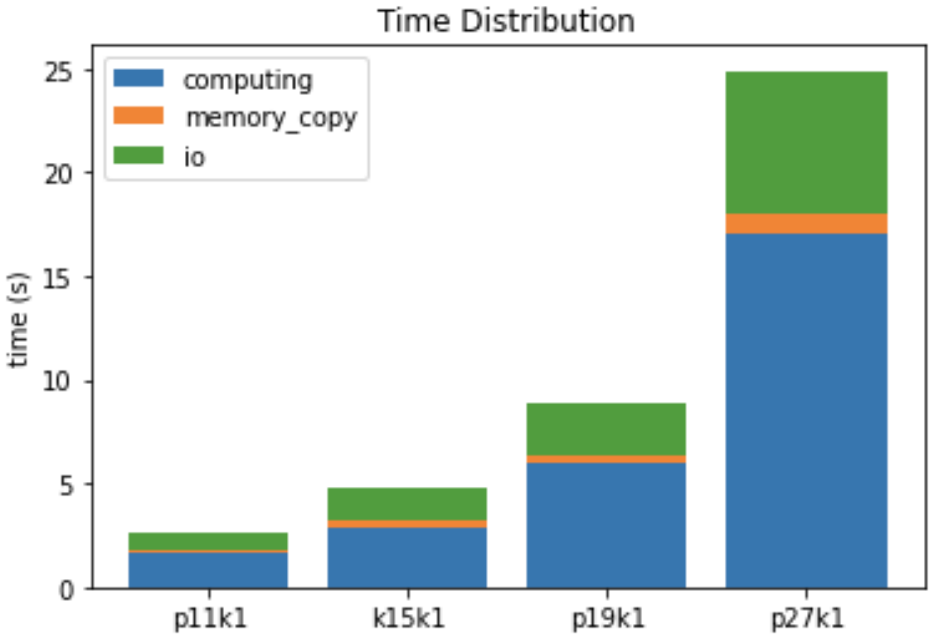
下面的圖時間可以明顯看出每加一個 optimize performance 就變得更好



Time Distribution

不同 size 的 input，然後分別去計算個別 `computing` 、 `memory copy` 、 `I/O` 時間，在計算 `I/O` 的地方我使用 `clock_gettime(CLOCK_MONOTONIC, &start)` 去計算。

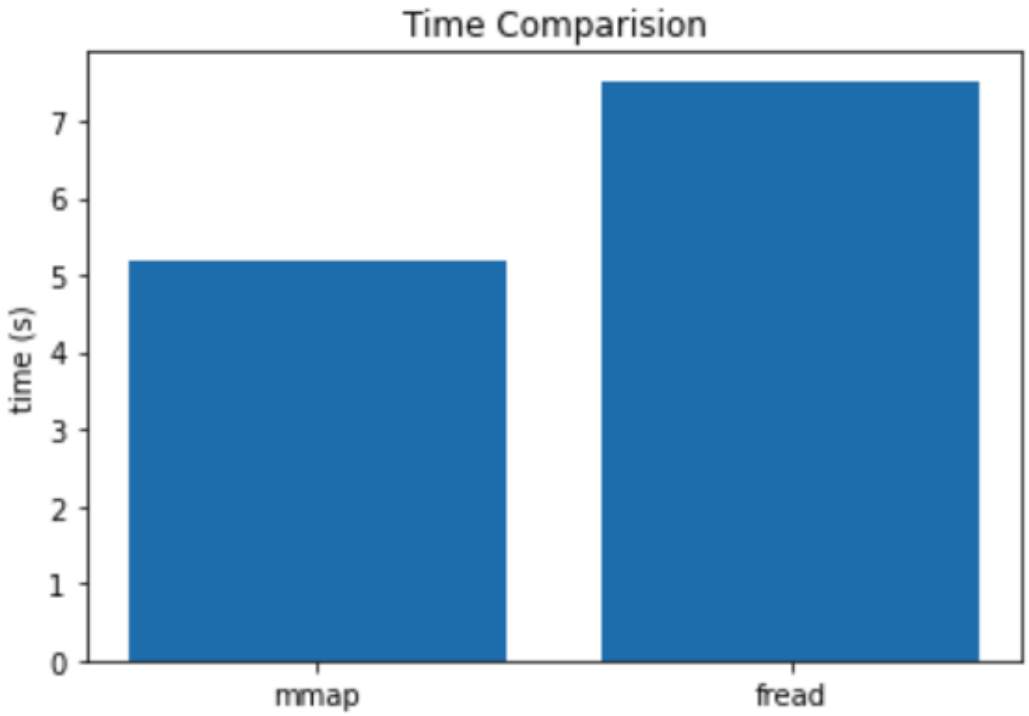
	p11k1	p15k1	p19k1	p27k1
Computing	1.66	2.9	5.94	17.09
Memcpy	0.129	0.278	0.445	0.9
I/O	0.843	1.601	2.45	6.91



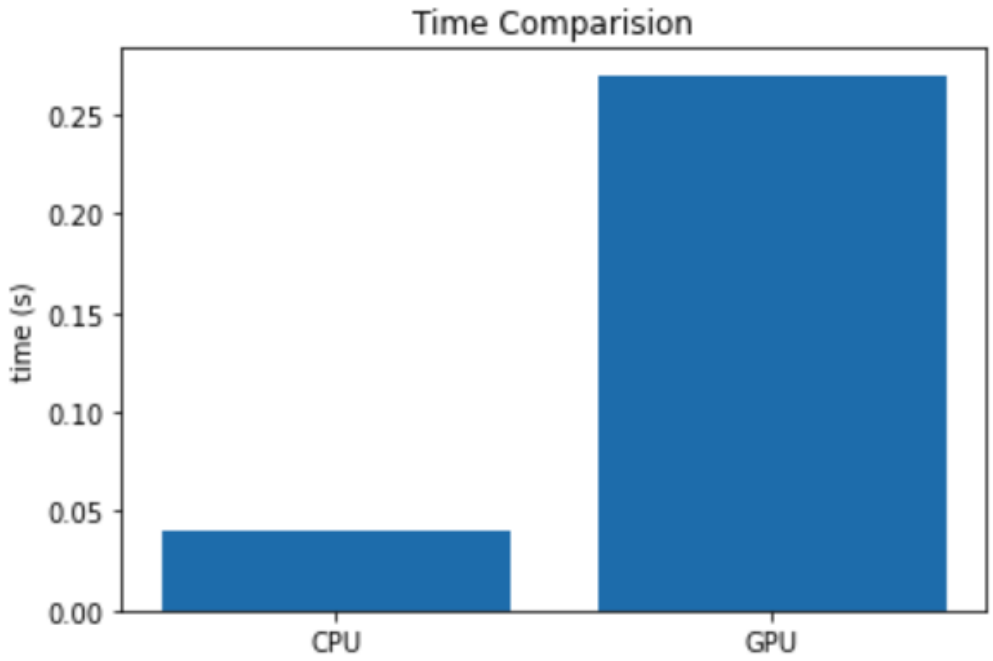
從上面圖跟表可以看到各個花費的時間，資料 size 變的時候，各項所花費的時間也會跟著變大，除了 I/O time 要寫回 file 的時候比較難避免之外，`computing` 基本上就是主要的 bottleneck，所以如何降低 computing 時間就是一個非常重要的目標。

Others

1. `mmap`
    - 這次我在計算 I/O 時間的時候多做了一個比較，把用 `mmap` 的讀檔跟 seq 提供的讀檔的方式在 p27k1 測資上進行比較
    - 因為在 seq 讀取 edge 資訊的時候每次只會讀三筆資料，這樣在 I/O 的時間有可能會因此增加
    - 下圖可以看到在測資越大的情況下時間會有差別



2. `Floyed Warshall`
    - 我發現在 single GPU 前面幾筆測資只要是小測資，因為運算量不大，所以在小測資上， `memory copy` 跟 `kernel function` 呼叫反而會是 bottleneck
    - 所以我在前面幾筆測資都直接使用一般的 CPU 來計算 floyed warshall
    - 我用 `c02.1` 來當作 testcase 比較



Conclusion

在這次的作業中，最令人難忘的部分就是 CUDA 的優化，因為 CUDA 跟平常寫的程式幾乎不太一樣，本來在 seq 上打法的差異在 CUDA 實作上會導致 performance 差非常多，因為 CUDA 對 memory 的使用要非常細心，如果沒有善用 shared-memory，速度差異會非常的大，而且因為 CUDA 優化的方向也非常多種可能，但是想要把各種優化都集結在一起是一件非常困難的事情，像是考慮 shared-memory 要怎麼分配，bank conflict 的問題，這些都是在優化過程中要仔細考慮的點。當然在寫過這次作業之後，不管是對 single GPU，或是 multiple GPUs 的實作都有更深入的了解，也了解了 GPU 資源越多，scalability 不一定更明顯可以提升，因為很可能中間為了要同步化而導致某些機器 idle，希望之後可以了解越來越多關於 CUDA 優化的方法，讓自己對於 CUDA 的架構可以有更好的認識。