CS542200 Parallel Programming

Homework 3: All-Pairs Shortest Path

108062373 蘇勇誠

Implementation

1. CPU (hw3-1)

我在 hw3-1 所使用的 Algorithm 為 Floyd Warshall Algorithm。實作 hw3-1 時，採用了 Pthread，將 Task 平行。平行的方式為以 row 切割（如右圖的第二層 for loop 所示），第 tid 個 thread 會負責計算 D[tid + k * ncpus]（ncpus 為 thread 數量，k = 0, 1, 2, …，tid + k * ncpus < number of Vertex）。而在右圖中最外層 for loop，第 k + 1 個 iteration 的 data 與第 k 個 iteration 的 data 之間有 dependency，因此需使用 barrier 等所有 thread 將第 k 個 iteration 計算完以後，才會進到第 k + 1 個 iteration。

```
void* FloydWarshall(void* args) {

  int* tid = (int*)args;

  for(int k = 0; k < V; k++) {

    for(int i = *tid; i < V; i += ncpus) {
      for(int j = 0; j < V; j++) {
        if(D[i][j] > (D[i][k] + D[k][j])) {
          D[i][j] = D[i][k] + D[k][j];
        }
      }
    }

    pthread_barrier_wait(&barrier);
  }

  pthread_exit(NULL);
}
```

2. Single GPU (hw3-2)

✓ how do you divide your data ?

將 data 切成 64 * 64，每個 block 會負責 size 大小為 64 * 64 的 data。而由於每個 block 之 Thread 的數量為 32 * 32，因此每個 Thread 會負責(64 * 64) / (32 * 32) = 4 筆 data 之運算。

✓ Padding

在實作 Blocked Floyd Warshall 時，若 input Vertex 的數量不是 256 的倍數時，我會將 input Vetex 數量 Padding 為 256 的倍數。如此一來，在 GPU 運算時，能較方便實作，也能避免在 kernel 中有過多的 if else，以減少 branch 的數量，提升效能。

✓ Blocking Factor

由於 Shared Memory 大小的限制，因此將 Block Factor 設為 64（shared memory size 為 49125 bytes，而我使用的 shared memory 之 type 均為 int，且因為計算需求，所以需 3 個 shared memory。因此，Block Factor = sqrt(shared memory size / sizeof(int) / 3)約為 64）。至於為何不將 Blocking Factor 設為 32 或 16，原因為 Blocking Factor 為 64 時可以使 global memory access bandwidth 較 Blocking Factor 設為 32 或 16 時低，以提升 Performance。

✓ Number of Threads

由於每個 block 最多只能有 1024 個 Threads（GPU 硬體限制），因此我就使用了 1024 個 Threads，將資源耗盡做計算。

✓ Number of Blocks

Block 的數量會分別在 Phase 1、Phase 2、Phase 3 的 Implementation 中進行討論。

✓ Implementation – Phase 1

在 phase 1 時，只需要**一個 GPU block** 進行計算 pivot block。實作方式為將 data 先從 global memory load 至 shared memory，每個 thread 會負責 load 4 筆 data 至 shared memory（由於 data size 為 64 * 64，block 數量為 32 * 32，因此每個 thread 負責 load 4 筆 data）。

```
int x = threadIdx.x;
int y = threadIdx.y;

// the address of required data in global memory
int global_x = x + round * BlockingFactor;
int global_y = y + round * BlockingFactor;

// Cache the current data block being calculated
__shared__ int shared_D[BlockingFactor][BlockingFactor];

// load data from golbal memory to shared memory
shared_D[y][x] = D[global_y * V + global_x];
shared_D[y + Half_BF][x] = D[(global_y + Half_BF) * V + global_x];
shared_D[y][x + Half_BF] = D[global_y * V + global_x + Half_BF];
shared_D[y + Half_BF][x + Half_BF] = D[(global_y + Half_BF) * V + global_x + Half_BF];

__syncthreads();
```

等到 block 中所有的 thread 都 load 完 data 後，就開始進行 Blocked Floyd Warshall 計算，計算的 iteration 數量為 blocking factor，如 SPEC 所述。計算 Blocked Floyd Warshall 時，每個 thread 亦會負責 4 筆 data 的計算，計算過程中 Access 的 memory 均為 Shared Memory。而 Block 中每個 thread 在計算完一個 iteration 時，都需要有 barrier 進行 synchronization，因為 iteration 間具有 dependency。

```
// execute phase 1 of Blocked FW
#pragma unroll 64
for(int i = 0; i < BlockingFactor; i++) {
    shared_D[y][x] = min(shared_D[y][x],
        shared_D[y][i] + shared_D[i][x]);

    shared_D[y + Half_BF][x] = min(shared_D[y + Half_BF][x],
        shared_D[y + Half_BF][i] + shared_D[i][x]);

    shared_D[y][x + Half_BF] = min(shared_D[y][x + Half_BF],
        shared_D[y][i] + shared_D[i][x + Half_BF]);

    shared_D[y + Half_BF][x + Half_BF] = min(shared_D[y + Half_BF][x + Half_BF],
        shared_D[y + Half_BF][i] + shared_D[i][x + Half_BF]);

    __syncthreads();
}
```

等到 block 中的所有 thread 都計算完成以後，就會將 data 從 shared memory 寫回 global memory。

```
// store data from shared mem to global mem
D[global_y * V + global_x] = shared_D[y][x];
D[(global_y + Half_BF) * V + global_x] = shared_D[y + Half_BF][x];
D[global_y * V + global_x + Half_BF] = shared_D[y][x + Half_BF];
D[(global_y + Half_BF) * V + global_x + Half_BF] = shared_D[y + Half_BF][x + Half_BF];
```

✓ Implementation – Phase 2

在計算 phase 2 時，需要# **of vertex / Blocking Factor 個 block**，每個 GPU block 會負責計算 pivot column 其中一個 block 與 pivot row 其中一個 block。由於每個 block 會負責計算 pivot column 其中一個 block 與 pivot row 其中一個 block，而計算 pivot column 與 pivot row 時會用到 pivot block，因此須用到 pivot row 其中一個 block、pivot column 其中一個 block、pivot block 中的 data。在 kernel 的一開始，thread 會將 data 從 global memory 移到 shared memory。每個 block 中的 thread 會將該 block 負責的一個 pivot column block 與一個 pivot row block，以及其對應的 pivot block load 到 shared memory。每個 block 會等到自己 block 中所需的 pivot column block、pivot block、pivot row block 都 load 到 memory，才開始計算。

```
__shared__ int shared_Pivot_Block[BlockingFactor][BlockingFactor];
__shared__ int shared_Pivot_Col[BlockingFactor][BlockingFactor];
__shared__ int shared_Pivot_Row[BlockingFactor][BlockingFactor];

// calculate memory access offset
global_x = x + round * BlockingFactor;
global_y = y + round * BlockingFactor;
col_x = x + round * BlockingFactor;
col_y = y + blockID * BlockingFactor;
row_x = x + blockID * BlockingFactor;
row_y = y + round * BlockingFactor;

// load data to shared memory
shared_Pivot_Block[y][x] = D[global_y * V + global_x];
shared_Pivot_Block[y][x + Half_BF] = D[global_y * V + global_x + Half_BF];
shared_Pivot_Block[y + Half_BF][x] = D[(global_y + Half_BF) * V + global_x];
shared_Pivot_Block[y + Half_BF][x + Half_BF] = D[(global_y + Half_BF) * V + global_x + Half_BF];

shared_Pivot_Col[y][x] = D[col_y * V + col_x];
shared_Pivot_Col[y][x + Half_BF] = D[col_y * V + col_x + Half_BF];
shared_Pivot_Col[y + Half_BF][x] = D[(col_y + Half_BF) * V + col_x];
shared_Pivot_Col[y + Half_BF][x + Half_BF] = D[(col_y + Half_BF) * V + col_x + Half_BF];

shared_Pivot_Row[y][x] = D[row_y * V + row_x];
shared_Pivot_Row[y][x + Half_BF] = D[row_y * V + row_x + Half_BF];
shared_Pivot_Row[y + Half_BF][x] = D[(row_y + Half_BF) * V + row_x];
shared_Pivot_Row[y + Half_BF][x + Half_BF] = D[(row_y + Half_BF) * V + row_x + Half_BF];

__syncthreads();
```

計算方式類似 Phase 1 所述。

```
// Calculate Blocked Folyd Warshall
#pragma unroll 64
for(int k = 0; k < BlockingFactor; k++) {

    // Calculate Pivot Column
    shared_Pivot_Col[y][x] = min(shared_Pivot_Col[y][x],
        shared_Pivot_Col[y][k] + shared_Pivot_Block[k][x]);

    shared_Pivot_Col[y][x + Half_BF] = min(shared_Pivot_Col[y][x + Half_BF],
        shared_Pivot_Col[y][k] + shared_Pivot_Block[k][x + Half_BF]);

    shared_Pivot_Col[y + Half_BF][x] = min(shared_Pivot_Col[y + Half_BF][x],
        shared_Pivot_Col[y + Half_BF][k] + shared_Pivot_Block[k][x]);

    shared_Pivot_Col[y + Half_BF][x + Half_BF] = min(shared_Pivot_Col[y + Half_BF][x + Half_BF],
        shared_Pivot_Col[y + Half_BF][k] + shared_Pivot_Block[k][x + Half_BF]);

    // Calculate Pivot Row
    shared_Pivot_Row[y][x] = min(shared_Pivot_Row[y][x],
        shared_Pivot_Block[y][k] + shared_Pivot_Row[k][x]);

    shared_Pivot_Row[y][x + Half_BF] = min(shared_Pivot_Row[y][x + Half_BF],
        shared_Pivot_Block[y][k] + shared_Pivot_Row[k][x + Half_BF]);

    shared_Pivot_Row[y + Half_BF][x] = min(shared_Pivot_Row[y + Half_BF][x],
        shared_Pivot_Block[y + Half_BF][k] + shared_Pivot_Row[k][x]);

    shared_Pivot_Row[y + Half_BF][x + Half_BF] = min(shared_Pivot_Row[y + Half_BF][x + Half_BF],
        shared_Pivot_Block[y + Half_BF][k] + shared_Pivot_Row[k][x + Half_BF]);

}
```

等到 Block 中所有 thread 均計算完畢以後，就會將 data 從 shared memory 寫回 global memory。

```
// store data to global memory
D[col_y * V + col_x] = shared_Pivot_Col[y][x];
D[col_y * V + col_x + Half_BF] = shared_Pivot_Col[y][x + Half_BF];
D[(col_y + Half_BF) * V + col_x] = shared_Pivot_Col[y + Half_BF][x];
D[(col_y + Half_BF) * V + col_x + Half_BF] = shared_Pivot_Col[y + Half_BF][x + Half_BF];

D[row_y * V + row_x] = shared_Pivot_Row[y][x];
D[row_y * V + row_x + Half_BF] = shared_Pivot_Row[y][x + Half_BF];
D[(row_y + Half_BF) * V + row_x] = shared_Pivot_Row[y + Half_BF][x];
D[(row_y + Half_BF) * V + row_x + Half_BF] = shared_Pivot_Row[y + Half_BF][x + Half_BF];
```

✓ Implementation – Phase 3

在計算 phase 3 時，需要# of vertex / Blocking Factor * # of vertex / Blocking Factor – 2 * # of vertex / Blocking Factor + 1 個 block，以計算 pivot row、pivot column、pivot block 以外的 data。不過為了實作方便，我直接用**# of vertex / Blocking Factor * # of vertex / Blocking Factor 個 block**，然後在 blockIdx.x == (round - 1)的 block 以及 blockIdx.y == (round - 1) 的 block 直接 return（不做任何運算）。在 phase 3 時，會將該 block 的 data，以及其所 depend 的 pivot column block、pivot row block 從 global memory load 出來進行加速。

```
// calculate memory access offset
x = threadIdx.x;
y = threadIdx.y;
global_x = x + blockIdx.x * BlockingFactor;
global_y = y + blockIdx.y * BlockingFactor;
col_x = x + round * BlockingFactor;
col_y = y + blockIdx.y * BlockingFactor;
row_x = x + blockIdx.x * BlockingFactor;
row_y = y + round * BlockingFactor;

// load data to non global memory
shared_Pivot_Col[y][x] = D[col_y * V + col_x];
shared_Pivot_Col[y][x + Half_BF] = D[col_y * V + col_x + Half_BF];
shared_Pivot_Col[y + Half_BF][x] = D[(col_y + Half_BF) * V + col_x];
shared_Pivot_Col[y + Half_BF][x + Half_BF] = D[(col_y + Half_BF) * V + col_x + Half_BF];

shared_Pivot_Row[y][x] = D[row_y * V + row_x];
shared_Pivot_Row[y][x + Half_BF] = D[row_y * V + row_x + Half_BF];
shared_Pivot_Row[y + Half_BF][x] = D[(row_y + Half_BF) * V + row_x];
shared_Pivot_Row[y + Half_BF][x + Half_BF] = D[(row_y + Half_BF) * V + row_x + Half_BF];

__syncthreads();

data_0 = D[global_y * V + global_x];
data_1 = D[(global_y + Half_BF) * V + global_x];
data_2 = D[global_y * V + global_x + Half_BF];
data_3 = D[(global_y + Half_BF) * V + global_x + Half_BF];
```

接著，進行 Blocked Floyd Warshall 運算，運算方式與 Phase 1 類似。

```
// calculation of Blocked FW
#pragma unroll 64
for(int k = 0; k < BlockingFactor; k++) {
    data_0 = min(data_0,
        shared_Pivot_Col[y][k] + shared_Pivot_Row[k][x]);
    data_1 = min(data_1,
        shared_Pivot_Col[y + Half_BF][k] + shared_Pivot_Row[k][x]);
    data_2 = min(data_2,
        shared_Pivot_Col[y][k] + shared_Pivot_Row[k][x + Half_BF]);
    data_3 = min(data_3,
        shared_Pivot_Col[y + Half_BF][k] + shared_Pivot_Row[k][x + Half_BF]);
}
```

運算完以後就會將結果寫回 global memory。

```
// store data to global memory
D[global_y * V + global_x] = data_0;
D[(global_y + Half_BF) * V + global_x] = data_1;
D[global_y * V + global_x + Half_BF] = data_2;
D[(global_y + Half_BF) * V + global_x + Half_BF] = data_3;
```

3. Multi GPU (hw3-3)

✓ how do you divide your data ?

將 data 切成 64 * 64，每個 block 會負責 size 大小為 64 * 64 的 data。而由於每個 block 之 Thread 的數量為 32 * 32，因此每個 Thread 會負責(64 * 64) / (32 * 32) = 4 筆 data 之運算。

✓ Blocking Factor

在 Multi-GPU 的實作中，Blocking Factor 設為 64，原因如同 Single GPU 實作所述。

✓ Number of Threads

由於每個 block 最多只能有 1024 個 Threads，因此我就使用了 1024 個 Threads，將資源耗盡做計算。

✓ Number of Block

Phase 1 為 1 個 Block（原因在 Single GPU 的 Implementation – Phase 1 有提到）。Phase 2 為 # of vertex / Blocking Factor 個 block（原因在 Single GPU 的 Implementation – Phase 2 有提到）。Phase 3 所需的 block 數量會在 Multi GPU 的 Implementation – Phase 3 提及。

✓ Implementation – Phase 1 & 2

在 Multi-GPU 的實作中 Phase 1 以及 Phase 2 與 Single GPU 相同。

✓ Implementation – Phase 3

在 Multi GPU 的 Phase 3 計算方式與 Single GPU 大致相同。與 Single GPU 不同的是，在 Phase 3 時，GPU 0 會計算 Graph 的 Adjacent Matrix 之上半部，GPU1 會計算 Graph 的 Adjacent Matrix 之下半部。此時，GPU 各需(# of vertex / Blocking Factor * # of vertex / Blocking Factor – 2 * # of vertex / Blocking Factor + 1) / 2 個 block。而為了方便實作，因此分配**# of vertex / Blocking Factor * # of vertex / Blocking Factor / 2** 給每張 GPU。

✓ Implementation – Two threads with Two GPUs & Communication

在 2 張 GPU 的 Blocked Floyd Warshall 實作中，我是用 OpenMP 開兩個 thread，兩個 thread 分別控制兩張 GPU。在 Blocked Floyd Warshall 每個 round 結束時，會使用#pragma omp barrier 進行同步，並與另一張 GPU Communication，將另一張 GPU 所需的 Data 送過去。由於 GPU 0 的 Phase 3 是計算 Graph 的 Adjacent Matrix 上半部的 data，GPU 1 的 Phase 3 是計算 Graph 的 Adjacent Matrix 下半部的 data（以 row 作為切割），因此 GPU 須將下一個 round 之 pivot row 的 data 從另一張 GPU 取得。而 data 複製的放式是使用 cudaMemcpyPeer。

```
// execute blocked FW on device
for(int round = 0; round < (n / BlockingFactor); round++) {
    blockedFW_Phase1 <<<blockNum1, threadNum>>> (device_dist[cpu_thread_id], round, n);
    blockedFW_Phase2 <<<blockNum2, threadNum>>> (device_dist[cpu_thread_id], round, n);
    blockedFW_Phase3 <<<blockNum3, threadNum>>> (device_dist[cpu_thread_id], round, n, offset);

    #pragma omp barrier

    if(((round + 1) >= offset) && ((round + 1) < (offset + blockNum3.y))) {
        if(cpu_thread_id == 0) {
            int addr_offset = (round + 1) * BlockingFactor * n;
            cudaMemcpyPeer(device_dist[1] + addr_offset, 1, device_dist[0] + addr_offset, 0, BlockingFactor * n * sizeof(int));
        }
        else {
            int addr_offset = (round + 1) * BlockingFactor * n;
            cudaMemcpyPeer(device_dist[0] + addr_offset, 0, device_dist[1] + addr_offset, 1, BlockingFactor * n * sizeof(int));
        }
    }

    #pragma omp barrier
}
```

Profiling Result

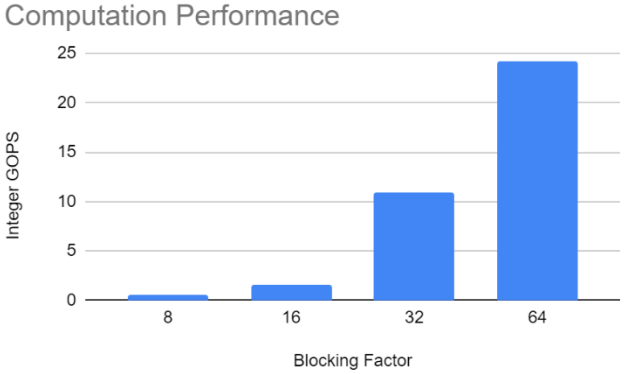
使用 c21.1 進行 Profile，其中執行時間最久與運算量最大的 kernel 為進行 phase 3 運算的 kernel。

	min	max	average
Achieved Occupancy	0.921122	0.9929352	0.923407
Multiprocessor Activity	99.04 %	99.72 %	99.63 %
Shared Memory Load Throughput	2873.4GB/s	3217.9GB/s	3171.0GB/s
Shared Memory Store Throughput	119.72GB/s	134.08GB/s	132.12GB/s
Global Load Throughput	179.59GB/s	201.12GB/s	198.19GB/s
Global Store Throughput	59.862GB/s	67.039GB/s	66.062GB/s

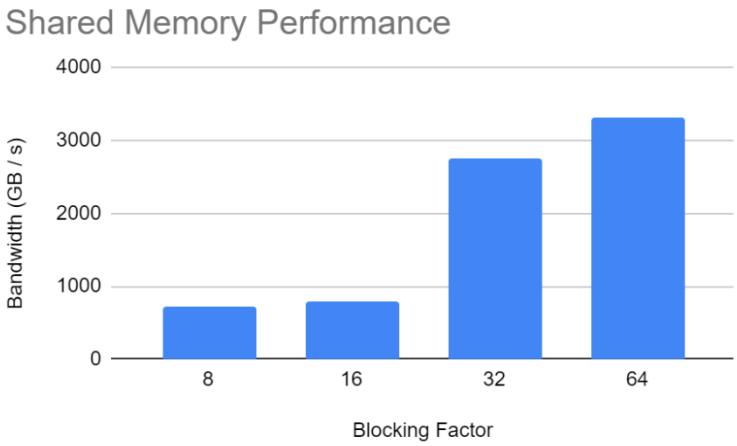
Experiment & Analysis

1. System Spec： 使用 hades server 進行實驗

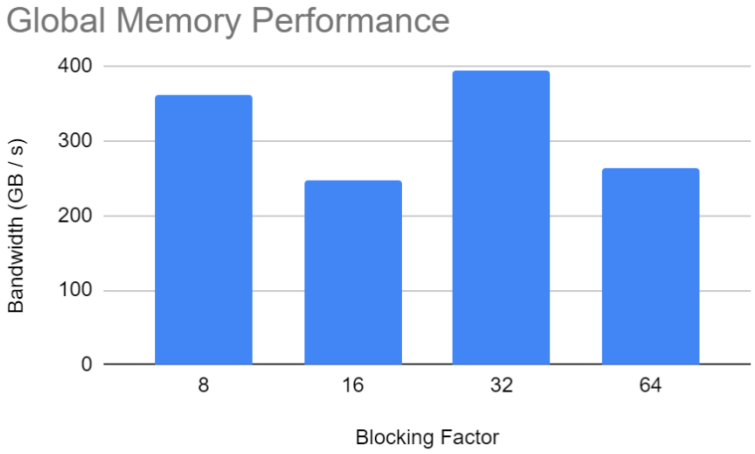
2. Blocking Factor (hw3-2)
   a. Computation Performance：使用 c21.1 進行實驗。Computation Performance 的測量方式為 integer 的 instruction 數量除以執行時間。由 Computation Performance 的圖表可見，當 Blocking Factor 越大時 Integer GOPS 就會越大。可能是因為當 Blocking Factor 增大時，需要 global memory load 和 store 的次數減少(計算過程均 access shared memory)，而使的 GOPS 增加。



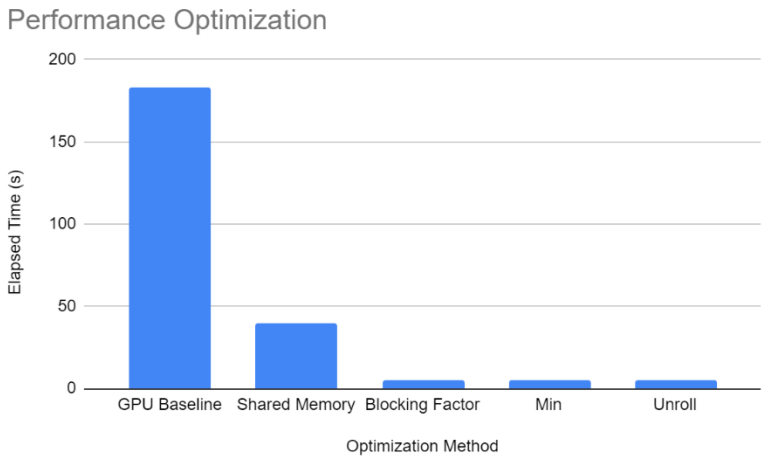
b. Shared Memory Performance：使用 c21.1 進行實驗。Shared Memory 的 Performance 包含 Shared Memory Load 與 Store 的 Performance。由 Shared Memory Performance 的圖表可見，當 Blocking Factor 增加時，Shared Memory 的 Bandwidth 會增加。原因為當 Blocking Factor 增加時，可以提升 Shared Memory 的使用率。



c. Global Memory Performance
當 Blocking Factor 增加時，Global Memory Access 的次數應該要下降。不過當 Blocking Factor 為 32 時 Global Memory 的 Bandwidth 卻上升了。

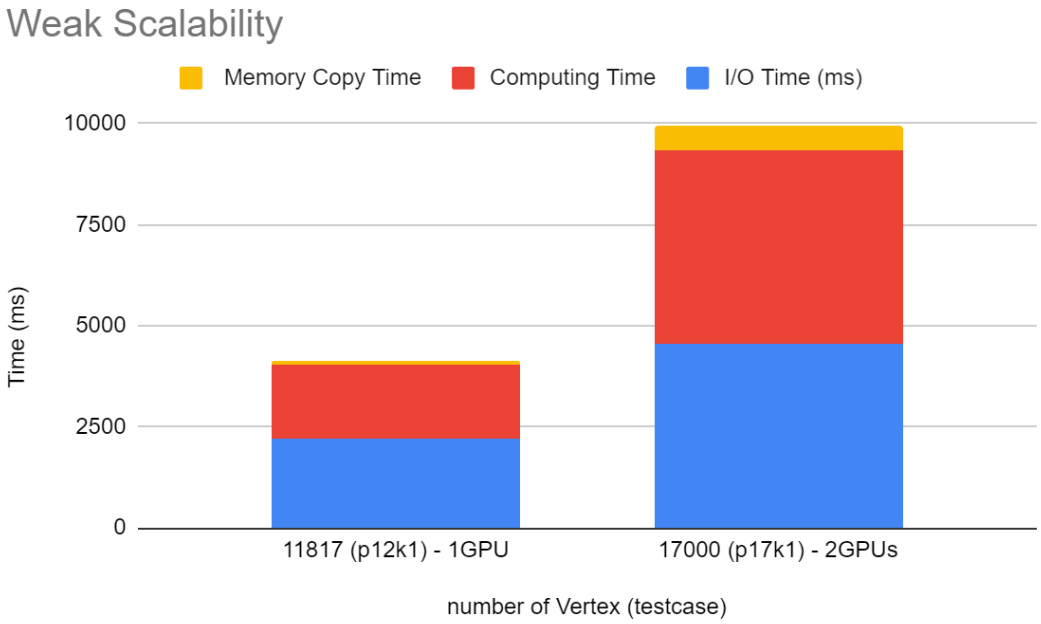


3. Optimization
由 Performance Optimization 的圖表可見，從 Access Global Memory 改為 Access Shared Memory 可以提升 4.5~5 倍的 Performance。此外，再將 Blocking Factor 進行調整，將 Blocking Factor 從 16 改為 64，增加了 4 倍的 Performance。而其他的優化如兩個 expression 要取最小值時，使用 cuda 的 min，避免使用 if-else，以及 unroll loop，只有稍微提升一點 performance。



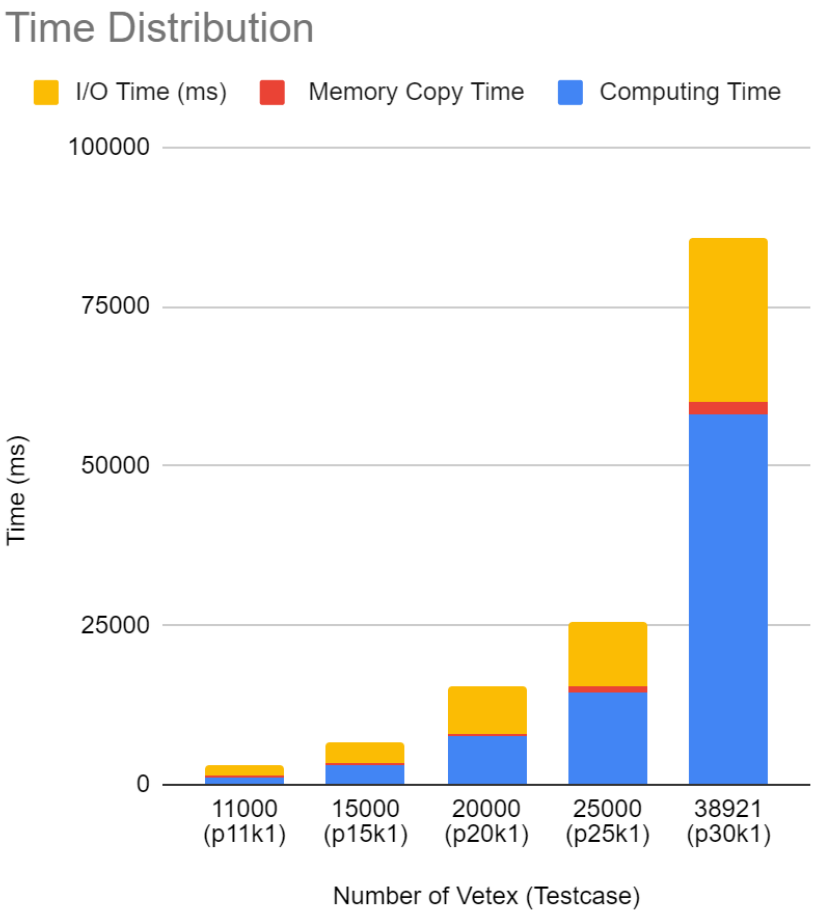
4. Weak Scalability：

   由於有 V 個 Vertex，且 input size 為 V * V 的 Graph，因此在計算 weak Scalability 時，2 張 GPU 的 data size 須為 1 張 GPU 的 data size 之 sqrt(2)倍。故選擇 testcase 12 與 testcase 17 來衡量 weak scalability （test case 17 input size / test case 12 input size = 17000 / 11817 = 1.41，約為 sqrt(2)倍)。由下圖結果可看出 Weak Scalability 的表現並不好，原因可能是當 input size 增加時，所需的 memory copy 時間會隨之增加。而且兩張 GPU 之間計算的 data 具有 dependency（需 synchronization），因此若一張 GPU 算完某個 iteration 的 data 時，需要等待另一張 GPU 也計算完該 iteration，才能進入下個 iteration。

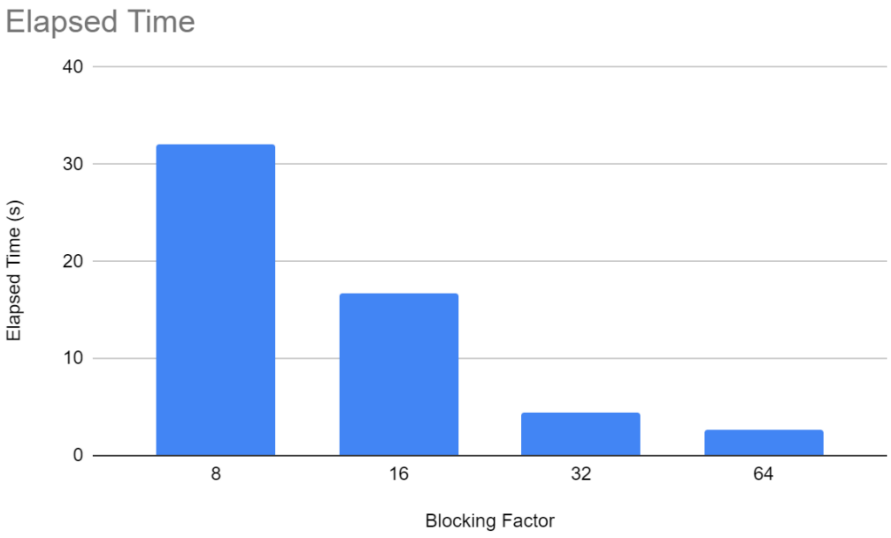


5. Time Distribution：

   在此次實驗中，我使用了不同 input size 的 testcase 進行實驗，並使用 nvprof 取得 Memory Copy 時間以及 Computing Time，使用 c++的 gettime 取得 i/o 時間。由 Time Distribution 的圖表可見，當 vertex 數量增加時，所需的 computing time、i/o time、memory copy time 均會花更多時間。



6. Others – 不同 Block Factor 的 GPU Elapsed Time：由下圖可見當 Block Factor 增加，GPU 的 Elapsed Time 下降，原因為 Data Reuse 次數增加，Global Memory Access 次數下降。



Conclusion

透過這次作業讓我知道 CUDA 程式要優化有多難，必須時常考慮 GPU 架構，需要一直注意不要頻繁 access global memory，要一直想是否可以把 data 放在 shared memory 進行 reuse。更難的是 GPU 程式 Debug 的部分，寫 GPU 程式時沒辦法像寫 sequential code，寫 sequential code 可以透過 print 就能很容易知道程式錯在哪，而 GPU code 很難透過 print debug。