CS542200 Parallel Programming

Homework 3: All-Pairs Shortest Path

107062103 王依婷

I. Implementation

a. Algorithm of hw3-1

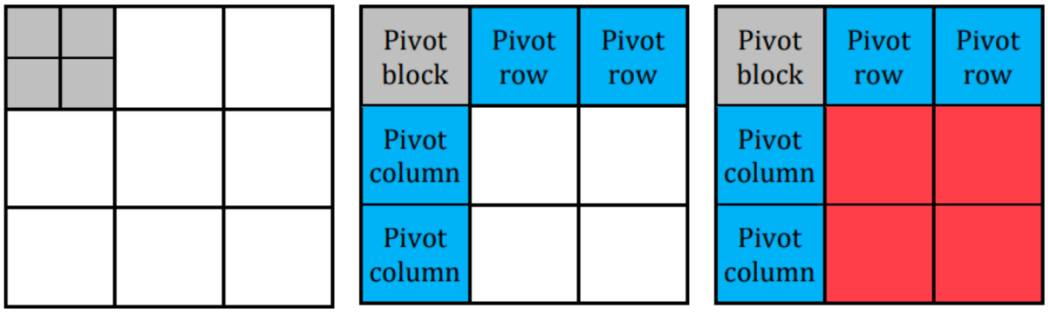
在 hw3-1 部分我使用的是基本的 Floyd-Warshall algorithm，再加上 OpenMp 來做平行。因為最外層的 k for-loop 每個迴圈之間會有 data dependency，因此只對裡面兩層 for loop 進行平行化。

```
49      for(int k = 0; k < v; k++){
50  #pragma omp parallel for shared(D)
51          for(int i = 0; i < v; i++){
52              for(int j = 0; j < v; j++){
53                  if(D[i][j] > D[i][k] + D[k][j])
54                      D[i][j] = D[i][k] + D[k][j];
55              }
56          }
57      }
```

b. Division of data in hw3-2, hw3-3

在 hw3-2 及 hw3-3 中，使用的是 spec 中介紹的 blocked Floyd-Warshall algorithm，會將整個尋找最短路徑的過程分成一塊一塊的 block 來進行，根據 phase 的不同，需要進行計算的 block 數量不同，也會需要不同 block 的值來對當下的 block 做計算。

我讓每個 GPU 中的 thread block 的大小對到一個 block，也就是一個 thread 一次會負責處理一個 vertex。在每一輪中計算中，會依序選定左上到右下對角線上的 block 作為該輪的 pivot block。在 Phase1 時只需要計算 Pivot block，不需其他 block 的數值；Phase2 時要計算和 Pivot 同一行及列的其他 block，需要自己以及在 Phase1 算好的 Pivot block；而 Phases3 時則計算剩餘的 block，需要 Phase2 中對應到自己 row index 和 column index 的兩個 block。



(a) Phase 1 (b) Phase 2 (c) Phase 3

- **hw3-2**

  Single GPU 有整個 distance matrix，只要在計算 block 時拿取所需對應 block 的數值就可以了。

- **hw3-3**

  hw3-3 有兩個 GPU，在計算量最大的 Phase3 時，我會將所有 block 分配到兩個 GPU 上計算，但要注意的是，若原本的 block 數量 (也就是 round) 是奇數，為了計算時取 index 方便以及正確性，會指定由第二個 GPU 多拿一個 block。

```
229     int round_split = (round%2 == 1 && thread_id == 1) ? round/2+1 : round/2;
```

c. Configuration in hw3-2, hw3-3

- **hw3-2**

  GPU block 數量和 thread 數量的部分，對應上一點 divide data 的內容，在 Phase1 時只要計算一個 Pivot block ; Phase2 時共有 round *2-1 個 block (round 為在行/列方向上 block 的數量) ，為了計算時取 index 方便直接開 2*round 個 blocks ; 而 Phase3 要計算(round-1)*(round-1)個 block，一樣為了方便直接開 round * round 個 blocks。每個 block 中 thread 的數量也一樣對照到 algorithm 中每個 block 裡面的 vertex 數量，為 B*B。

  至於 blocking factor，希望一個 cuda GPU 中的 block 能夠剛好處理一個計算時的 block，使用 deviceOuery.cpp 查閱了細

節後(+chap7 講義)得知一個 block 最多能有 1024 個 threads，因此將 blocking factor 設成 32，讓 block 能放滿，裡面有 32*32=1024 個 thread。

	Phase 1	Phase 2	Phase 3
blocking factor (B)	32		
# of blocks	(1, 1)	(2, round)	(round, round)
# of threads	(B, B)		

- **hw3-3**

  blocking factor 同上。

  Phase1 和 2 計算量較小，且為了讓 Phase3 兩個 GPU 各算一半的時候能夠直接開始，不需要再做 data copy，我讓它們也都計算 Phase1 和 Phase2 的所有 block，所以 GPU block 數量和 hw3-2 相同。Phase3 時也是對應到 divide data 所說，兩個 GPU 各有 split * round 個 blocks。

	Phase 1	Phase 2	Phase 3
blocking factor (B)	32		
# of blocks	(1, 1)	(2, round)	(round_split, round)
# of threads	(B, B)		

d. e. Implementation & Communication of hw3-3

- **hw3-2**

  按照作業 spec 上的 block Floyd-Warshall algorithm 方法實作 (以下所附圖片都是 baseline 方法的程式碼)，讀進資料時存成一為的陣列 D，使用 cudaMemcpy 把 D 複製到 device 上，接著

為 Phase1、2、3 各別實作不同的 kernel function 來計算。因為 case 中的 vertex 數量有可能不能整除於設定好的 blocking factor，那這樣在以 block 為單位計算的時候會遇到取到超過陣列範圍的狀況，所以在一開始讀進 vertex 數量時，就會計算出若要湊成 B 的倍數，vertex 還需要再 padding 多少，並將資料陣列開成 B 的倍數，多的 padding 部份補成 $2^{30}-1$，讓計算能夠順利進行，如以下程式碼所示。

```
28  void input(char* fileName){
29      FILE* file = fopen(fileName, "rb");
30      fread(&v, sizeof(int), 1, file);
31      fread(&e, sizeof(int), 1, file);
32      padding = v + B - (v%B);
33      D = (int*) malloc(sizeof(int)*padding*padding);
34
35      // INIT
36      for(int i = 0; i < padding; i++){
37          for(int j = 0; j < padding; j++){
38              if(i == j) D[i*padding+i] = 0;
39              else D[i*padding+j] = INF;
40          }
41      }
42
43      int edge[3];
44  >   for (int i = 0; i < e; ++i) {…
47      }
48      fclose(file);
49  }
```

Phase1 計算一個 Pivot block，根據目前的輪次 r 找到 thread 對應到的 block 中數字，跑 Floyd-Warshall 的三層 for 迴圈計算。而因為不同 k 之間所計算的值有 dependency，所以要加上 __syncthreads()。以下是 Phase1 部分的 code。

```
64  __global__ void Phase1(int *Dist, int B, int r, int v){
65      int i = threadIdx.x + r * B;
66      int j = threadIdx.y + r * B;
67
68      if(i >= v || j >= v) return;
69
70      for(int k = 0; k < B; k++){
71          int k_block = k + r * B;
72          if(Dist[i*v+j] > Dist[i*v+k_block] + Dist[k_block*v+j]){
73              Dist[i*v+j] = Dist[i*v+k_block] + Dist[k_block*v+j];
74          }
75          __syncthreads();
76      }
77  }
```

Phase2 計算和 Pivot block 同 row 或是同 column 的 blocks。我設定當 blockIdx.x=0 時計算 row 方向的 blocks，=1 時計算 column 方向的，根據這兩種情況找到對應的 Pivot block index，取值來計算來計算。以下是 Phase2 部分的 code。

```
__global__ void Phase2(int *Dist, int B, int r, int v){

    //  對應到Phase1的那個block，不用做
    if(blockIdx.y == r)
        return;

    int i = threadIdx.x + blockIdx.y * B;
    int j = threadIdx.y + blockIdx.y * B;
    int p_i = threadIdx.x + r * B;
    int p_j  = threadIdx.y + r * B;

    // blockIdx.x = 0 for row, = 1 for column
    if(blockIdx.x == 0) i = p_i;
    else j = p_j;

    if(i >= v || j >= v) return;
    for(int k = 0; k < B; k++){
        int k_block = k + r * B;
        if(Dist[i*v+j] > Dist[i*v+k_block] + Dist[k_block*v+j]){
            Dist[i*v+j] = Dist[i*v+k_block] + Dist[k_block*v+j];
        }
    }
}
```

最後 Phase3 的部分計算所有剩餘的 block，一樣會按照自己的 blockIdx 和 threadIdx 取得現在要計算的數值位置、還有其他所需要的、Phase2 計算好的數值。

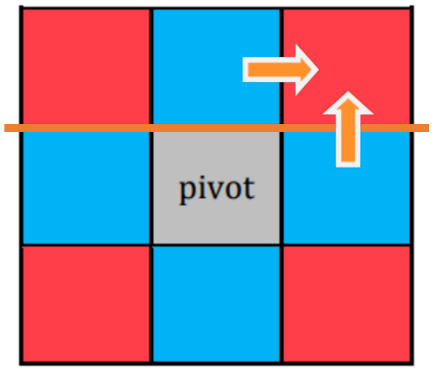
```
__global__ void Phase3(int *Dist, int B, int r, int v){
    // 對應到Phase1 & 2計算的部分，不用做
    if((blockIdx.x == r) || (blockIdx.y == r))
        return;

    int i = threadIdx.x + blockIdx.x * B;
    int j = threadIdx.y + blockIdx.y * B;
    // Pivot-related indices
    int p_i = threadIdx.x + r * B;
    int p_j  = threadIdx.y + r * B;

    if(i >= v || j >= v) return;

    for(int k = 0; k < B; k++){
        int k_block = k + r * B;
        if(Dist[i*v+j] > Dist[i*v+k_block] + Dist[k_block*v+j]){
            Dist[i*v+j] = Dist[i*v+k_block] + Dist[k_block*v+j];
        }
    }
}
```

- hw3-3



(b) Round 2

在 hw3-3 中，按照設定總共使用兩個 core、兩個 GPU 來實作 multi-GPU，也就是一個 core 對到一個 GPU，而我也是使用 OpenMP 來做平行。

在 Phase1 和 2 時兩個 thread 都會呼叫相同的 kernel function，做一樣的計算，這邊都和 hw3-2 相同。Phase3 如同上面所說，將所有要計算的 block 分成一半來處理，為了下一輪的正確性，兩個 GPU 之間需要做一些 data 的複製及溝通。如左圖所示，橘線上方交給 device0，下半由 device1 負責，經過第一輪計算後，他們會各自擁有一半的正確計算結果，當進到第二輪 Pivot block 移動到 device1 那半的位置，若在 Phase3 時 device0 要計算右上 block 的話，需要 device1 所計算好的 row 方向的 block，所以需要把他們複製到 device0 中。統整之後可以發現，所需要的 column 方向 block 上一輪已經計算完畢，不需要複製；只有 row 方向的 block 會因為 Pivot block 移動到了另一個 device 那半的位置，而要做複製的動作。

```
245         int curRow = r * B * padding;
246         if((r >= offset) && (r < offset + round_split)) {
247             cudaMemcpy(device_D[other_thread]+curRow, device_D[thread_id]+curRow, sizeof(int)*B*padding, cudaMemcpyDeviceToDevice);
248         }
249         #pragma omp barrier
```

II. Profile Results (hw3-2)

我使用了 c20.1 來做 profiling，取最大的 Phase3 kernel，下表格為結果：

- **occupancy**：在一個 SM 中 active warp 和最大 active warp 數量的比值，這邊使用 nvprof 中的 achieved_occupancy，它是取所有 SM 在任何時刻的比值平均值。
- **SM efficiency**：平均在 GPU 的所有 multiprocessor 中，至少有一個 warp 是 active 的比例。用 sm_efficiency 來查詢。

	Min	Max	Avg
occupancy	0.894175	0.897251	0.896442
SM efficiency	99.74%	99.82%	99.77%
shared memory load throughput	2696.6GB/s	3048.8GB/s	3029.9GB/s
shared memory store throughput	112.36GB/s	127.04GB/s	126.24GB/s
global memory load throughput	168.54GB/s	190.55GB/s	189.37GB/s
global memory store throughput	56.179GB/s	63.518GB/s	63.122GB/s

可以看到因為做了使用 shared memory 的優化，他的 throughput 比 global memory 的高出許多。而在這次作業較常需要把資料從 memory 中取出來做比較，值較大才做更新，所以 load throughput 都大於 store throughput。

III. Experiment & Analysis

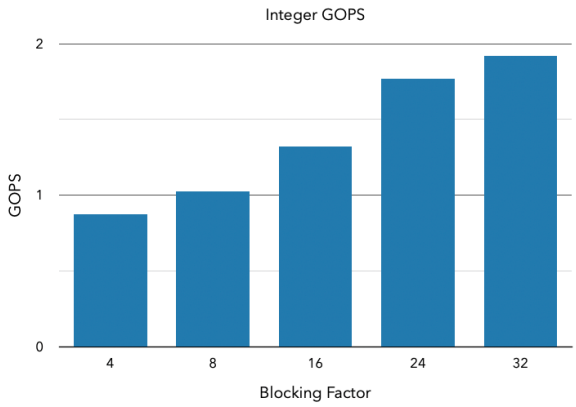
Blocking factor 和 optimization 兩部分也使用 c20.1 來測量；time distribution 再加上其他三個不同 input size 的 case 來做。

a. System spec

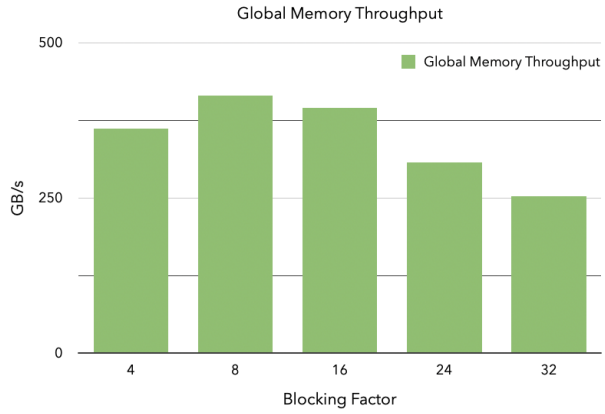
使用課程所提供的 hades，沒有用額外的設備。

b. Blocking factor (hw3-2)

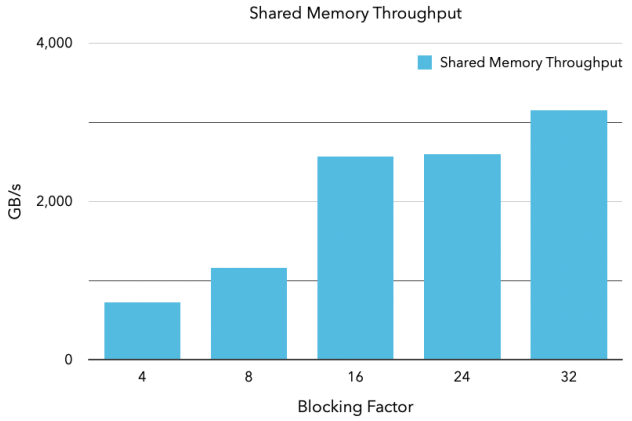
以下是調整不同 blocking factor 所測得的數據柱狀圖，：



使用 inst_integer 來取得數據。從 integer GOPS 的數據中可以看出，隨著 blocking factor 的增加，執行的指令數量也增加，因為使用了更多的 thread，也能夠一次取出更大量的資料來運用、計算，增加效率。



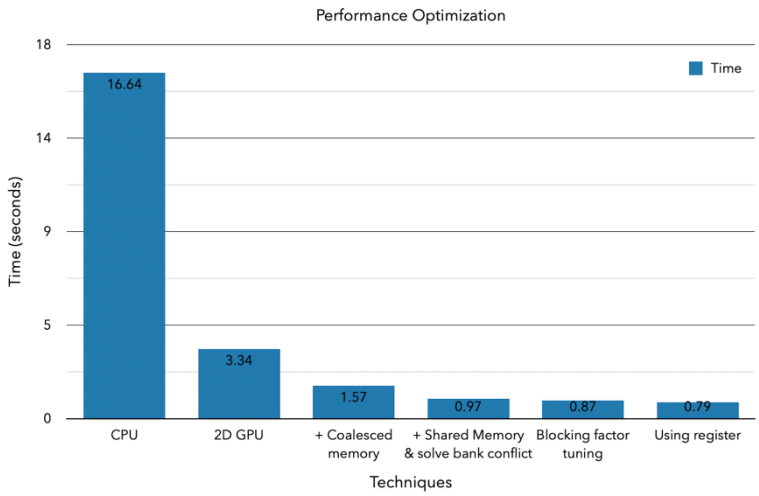
這邊我是使用 throughput，也就是實際上有效的傳輸量作為評量。使用了 shared memory 後，減少了將 access global memory 的次數，只要先把 global memory 裡面的資料搬到 shared memory 就能作後續的計算工作。當 blocking factor 變大，這個搬移資料的次數也變少了，所以數值便下降。

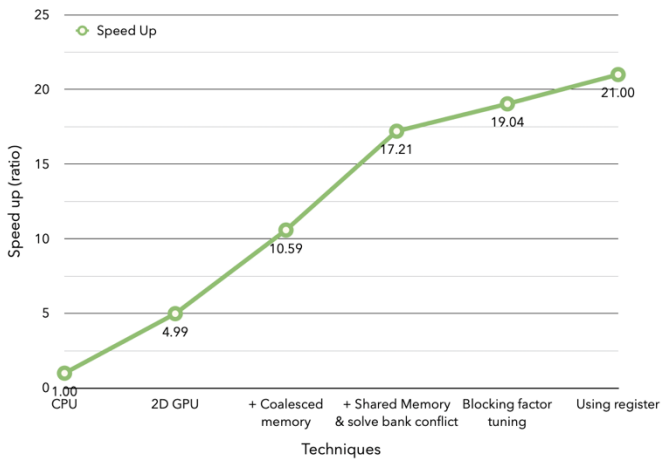


從 Shared memory 的結果中也能看出，因為 blocking factor 變大，一次要搬移到 shared memory 的資料也變多了，因此數值隨著 blocking factor 變大而增加。

c. Optimization (hw3-2)

這邊主要先以整體時間上的減少來做比較，各部分所花的時間會展示在下面的 time contribution 部分。以下是各方法的執行時間和 speedup，並且一一說明優化的方式：





- **2D GPU**

  使用 2D grid 和 2D block，block 的大小會根據 blocking factor 而定。

- **Coalesced memory access**

  為了能讓同一個 warp 中所有 thread 的 data 可以連續，一起被 access，GPU thread 執行的順序要和 data 存取的順序一致。所以我讓 threadIdx.y 是 row 方向，也就是同個 warp 中 threadIdx.y 都相同(最後 blocking factor 最大 tune 到 32，代表一個 block 裡面的 thread 有 32*32 個，也還是可以剛好可以 fit)。

- **Shared memory**

  在每個 phase 中，都會先將計算時需要用到的相關 block 放到 shared memory：Phase1 只需要正在計算的 block；Phase2 還需要 Pivot block，因此也 load 到 shared memory 中；Phase3 需要多 load Pivot row 和 Pivot column。這些所需要的 block 內數值，會由正在計算的 block 中每個 thread 幫忙 load 相同 index 位置。最後用__syncthreads()來將 block 內 thread 的行爲做同步化，確保所有需要的值在計算前都到位。這時的 threadIdx.x 和 threadIdx.y 的順序也和上面相同，同時也避免了 shared memory 中 bank conflict 的可能。

- **Blocking factor tuning**

  Configuration 的部分有提到，因為一個 block 中最多能有 1024 個 thread，因為我是以 blocking factor 的大小來設定 GPU 的 block size，所以將 blocking factor tune 到 32，讓 thread 可以開到最滿。

- **Using registers**

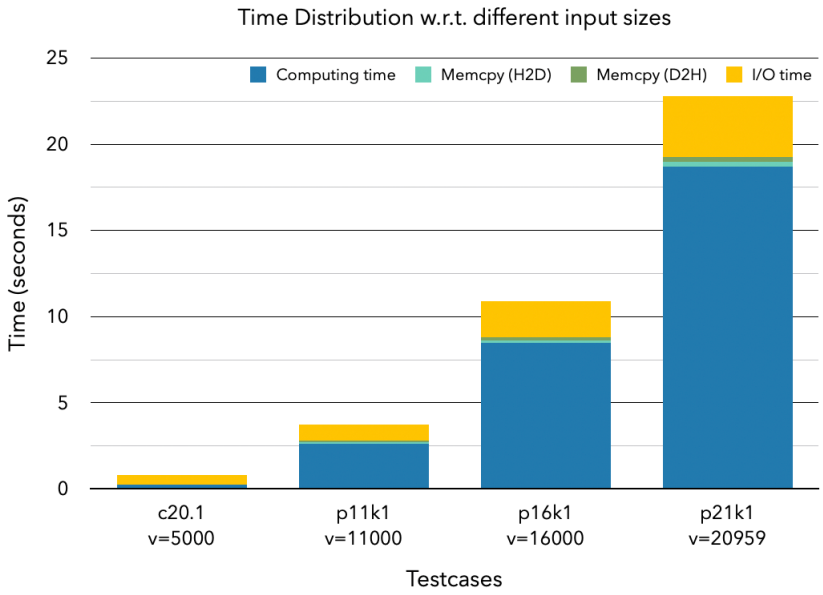
  將過程中會重複計算到的數值，如各個 block 中的 index，都用速度最快的 register 暫存起來，就不需要一直做加減乘除。

```
136     __shared__ int sharedPivot[B*B];
137     __shared__ int sharedSelf[B*B];
138     int tIdxY = threadIdx.y*B;
139     int idx = tIdxY + threadIdx.x;
140     int idxD = i*v+j;
141     int idxPivot = p_i*v+p_j;
142     sharedPivot[idx] = Dist[idxPivot];
143     sharedSelf[idx] = Dist[idxD];
```

d. Weak scalability (hw3-3)

在 Phase3 時會將資料分成一半給兩個 GPU 來計算，因此兩個 GPU 之間還需要做溝通以及同步化，memory access 也有可能變得不連續。最後兩個 GPU 做完一輪後都必須同步，保持整個 matrix 的狀態一致正確性。老師上課時有提到 multi-GPU 以我們目前學到的部分，加速的效果不好、很難，不過還是能夠優化，不過在這個作業中就無法做到。

e. Time distribution (hw3-2)、Others



我挑選了 size 差異儘量差不多的一些 testcase 來做實驗。在 size 較小的 case 中，其實 I/O 所花的時間會比計算還要來得多一些，但到了較大的 case，雖然 I/O 時間也隨之增加，但都會是計算最耗時。而不管 Memcpy 是 HostToDevice 還是 DeviceToHost，所花的時間差不多且並不會影響整體的 performance 太多，不會是問題。

在這次的作業中，其實還有更多優化的方法，但我沒有實作到那麼多，所以可以看到雖然每個 testcase 的 size 差距差不多，在計算上所多花的時間卻是越來越多，最明顯的就是從 c20.1 到 p11k1 之間的差異，vertex 數量是變兩倍沒錯，但計算的時間遠超過兩倍，還有很多可以優化的空間。另外，主要都是針對比較最短路徑時的計算來做優化、改善，像是 I/O 的部分沒有特別更改，尤其是在 size 較小的 case 中，也造成了一定的時間花費。

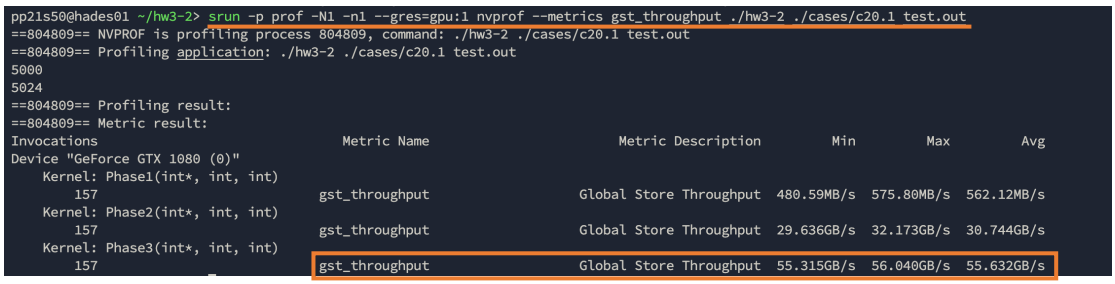
IV. Conclusion

這次作業要實作的部分更多了，優化的方法也因為加入了 GPU 變得更加多元且 powerful，最讓我驚奇的是 coalesced memory 的部分，程式上只是改個執行的方向，在實際 memory access 時卻有很大的差別，節省了不少時間！

另外，在做作業時遇到了一個小疑問，在做 profiling 時，當我使用--metrics [任一 metric]指令，和--metrics [多個 metrics]時，在同一 metric 上所得出的結果會是不同的，甚至差蠻多的，讓我十分疑惑！狀況如下圖所示，可以看到 global store throughput 的結果差異。

```
pp21s50@hades01 ~/hw3-2> srun -p prof -N1 -n1 --gres=gpu:1 nvprof --metrics achieved_occupancy,sm_efficiency,shared_load_throughput,shared_store_throughput,gld_throughput,gst_throughput ./hw3-2 ./cases/c20.1 test.out
==804368== NVPROF is profiling process 804368, command: ./hw3-2 ./cases/c20.1 test.out
==804368== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
==804368== Profiling application: ./hw3-2 ./cases/c20.1 test.out
5000
5024
==804368== Profiling result:
==804368== Metric result:
Invocations                               Metric Name                         Metric Description         Min         Max         Avg
Device "GeForce GTX 1080 (0)"
    Kernel: Phase1(int*, int, int)
        157                        achieved_occupancy                        Achieved Occupancy    0.496467    0.496976    0.496710
        157                                sm_efficiency                   Multiprocessor Activity      3.46%       3.74%       3.66%
        157                    shared_load_throughput             Shared Memory Load Throughput  52.404GB/s  63.535GB/s  60.574GB/s
        157                   shared_store_throughput            Shared Memory Store Throughput  629.43MB/s  10.805GB/s  1.6156GB/s
        157                            gld_throughput                  Global Load Throughput  553.21MB/s  670.72MB/s  639.46MB/s
        157                            gst_throughput                 Global Store Throughput  553.21MB/s  670.72MB/s  639.46MB/s
    Kernel: Phase2(int*, int, int)
        157                        achieved_occupancy                        Achieved Occupancy    0.929182    0.953401    0.942670
        157                                sm_efficiency                   Multiprocessor Activity      88.59%      92.43%      90.41%
        157                    shared_load_throughput             Shared Memory Load Throughput  2015.2GB/s  2412.1GB/s  2299.7GB/s
        157                   shared_store_throughput            Shared Memory Store Throughput  70.801GB/s  489.10GB/s  128.82GB/s
        157                            gld_throughput                  Global Load Throughput  62.007GB/s  74.220GB/s  70.761GB/s
        157                            gst_throughput                 Global Store Throughput  31.003GB/s  37.110GB/s  35.380GB/s
    Kernel: Phase3(int*, int, int)
        157                        achieved_occupancy                        Achieved Occupancy    0.894175    0.897251    0.896442
        157                                sm_efficiency                   Multiprocessor Activity      99.74%      99.82%      99.77%
        157                    shared_load_throughput             Shared Memory Load Throughput  2696.6GB/s  3048.8GB/s  3029.9GB/s
        157                   shared_store_throughput            Shared Memory Store Throughput  112.36GB/s  127.04GB/s  126.24GB/s
        157                            gld_throughput                  Global Load Throughput  168.54GB/s  190.55GB/s  189.37GB/s
        157                            gst_throughput                 Global Store Throughput  56.179GB/s  63.518GB/s  63.122GB/s
```

▲ 同時使用多個 metrics



▲ 單獨使用 gst_throughput 的狀況

V. Reference

- https://docs.nvidia.com/cuda/profiler-users-guide/index.html#metrics-reference