Homework 3: All-Pairs Shortest Path

111062613 蔡鎮宇

1. Implementation

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

我使用的是 Floyd-Warshall 的演算法，然後直接把系統能開到的 threads 用到最多，使用 OpenMP 的 parallel for directive，schedule 選 static，讓每個 thread 去搶著運算總共 n^2 的點。重複這個行為做 k 次。

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

在 hw3-2 中，假設有 n 個 vertices，block factor 設 64，代表有一個 n*n 的矩陣要運算，因此會叫 GPU 開啟夠多個 block 去對應這 n*n 個點。每個 block 中的每個 thread 負責 x 個點。

在 hw3-3 中，我將 n*n 個點用 hw3-2 的方式對應成一堆 block，接著把 block 以 row 為單位對切成兩半，GPU 1 負責 n/2 個 row，GPU 2 負責剩下的 row。

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

Blocking factor 皆會設置為 64，因為硬體規格 shared memory 只有 49152 bytes，在 phase 3 中，我們需要用到 3 個 shared matrix，一個 matrix 要用到 64*64*4 = 16384，而 49152 = 16384 * 3。為了不讓 block 用到 global memory，因此以 shared memory 的大小做為選擇 blocking factor 標準。

Block 數的設定根據 n*n 的大小，先假設 n 能被 64 整除，則需要用到 $n^2 / 64^2$ 個 block。因為不是每個 n 都會剛好被整除，所以一開始就要把 n 轉換成一個能被整除的數 n'，其中 n' = n + blocking factor – (n % block factor)。如果沒有生出 n'，程式在 GPU 階段很容易出錯。

Threads 數是 32*32 = 1024，因為硬體規格一個 block 最多用 1024 threads，為了讓平行最大化，就用最多的 threads 數。

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

每次只傳送一個 pivot row 給對方。從 b. 可以知道兩個 GPU 各自負責一半的資料，代表的是他們負責的資料永遠是最新的，也就是正確的答案。假設今天 pivot row 是在 GPU 1 負責的地方，那他會把 pivot row 的資料傳給 GPU 2，GPU 2 接著能用這些資料運算出它需要的 pivot row。在 e.中會詳細解釋

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

介紹 c.：Figure 1 中，blocking factor = 4，n = 6，淺色部分是原本理想上 matrix 的大小，但是因為 block factor 為 4 無法整除 n，因此我們要 allocate 8 * 8 (n' = 8)的空間，這樣才能讓 4 個 4*4 的 GPU block 去一一對應到每個 matrix 的 block。

並且假設每個 block 最多能用 16 個 thread，我也會設定一個 4*4 的 thread block，讓每一個 thread 去負責下圖每一格的計算。

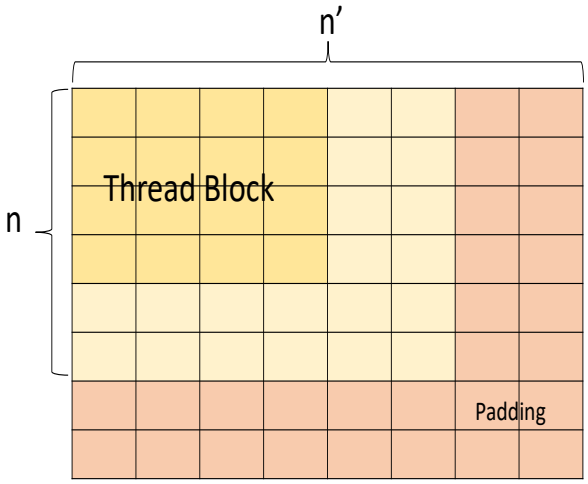


Figure 1. Allocated Matrix and Thread Block

介紹 d.：hw3-3 中的演算法和 hw3-2 幾乎一樣，一樣計算 phase1, 2, 3。差別是一、Phase 3 的時候，每個 GPU 只會計算屬於自己的那一半；二、 phase 都算完後，某一方 GPU 要傳送一個 pivot row 給需要的那方。

假設 n = 8, block factor = 1。Figure 2 顯示現在要進行到 round 1，也就是 $row^1[1]$要被計算成 pivot row。由於在前一輪 phase 3 的時候，GPU 2 並不會計算到 $row^0[1]$的值，也就是 $D^{(0)}(1, j)$ for all j = 0, …, 7 的值並沒有被更新到。因此在 round 0 結束後，GPU 1 會把 $row^0[1]$傳給 GPU 2，GPU 2 在 phase 1 和 2 時，就可以成功計算出 $D^{(1)}(1,j)$的值。因為在 block factor = 1 時看不出為何要計算 phase 1 和 phase 2，可以自行想像每個 row 是一整列的 block，block 裡面還有更多要計算的點。當計算出正確的 pivot row 和 column 後，phase 3 也能成功計算出資料。

Figure 3 進一步顯示 GPU 2 所需要的藍色格子。實際上他的 phase 3 只需要一半 pivot column，且 column 的資料會被 GPU 2 於 round 0 時更新，所以他在 phase 2 不需要使用 GPU 1 的 column（除了 $D^{(0)}(1, 1)$）。因此我們並不需要傳送 pivot column，而是只要傳送 pivot row。

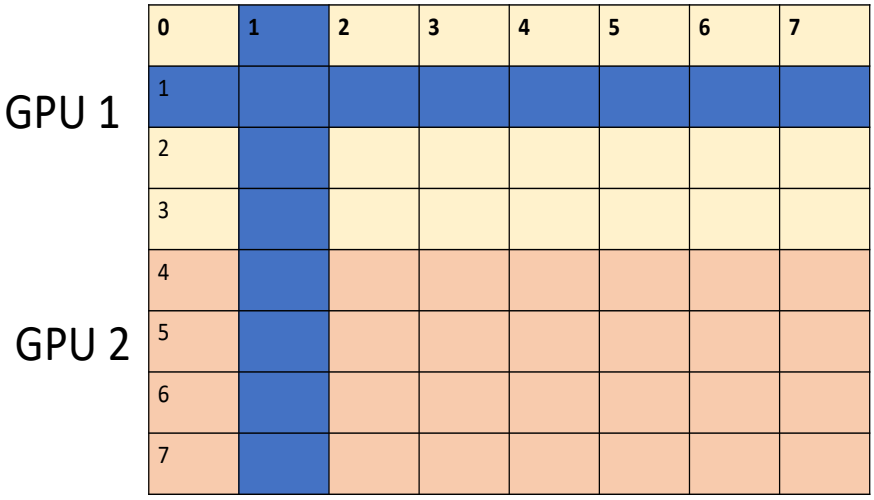


Figure 2. round 1's pivot row and column

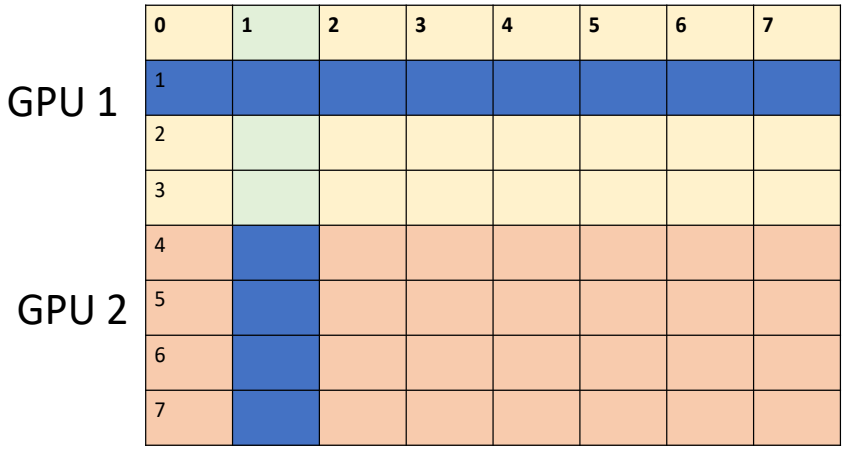


Figure3. GPU 2 only cares about Blue cells

2. Profiling Results (hw3-2)

使用 p11k1 作為測試的 testcase。

i. occupancy

我以為 phase3 的 occupancy 會更高，結果只有約 90%，因為每個 block 的每個 thread 基本上都有對應的點要去計算(除了超出了原本計算範圍卻又還是被我們 allocate 的記憶體位置外)。

Invocations	Metric Name	Metric Description	Min	Max	Avg
Device "NVIDIA GeForce GTX 1080 (0)"					
Kernel: phase3_all(int, int*, int)					
172	achieved_occupancy	Achieved Occupancy	0.906690	0.915608	0.914750
Kernel: test_phase1(int, int*, int)					
172	achieved_occupancy	Achieved Occupancy	0.498174	0.498280	0.498215
Kernel: phase2_combine(int, int*, int)					
172	achieved_occupancy	Achieved Occupancy	0.969867	0.976754	0.973398

ii. sm efficiency

phase3 的每個 specific multiprocessor 幾乎都有 active 的 warp 在運作，平均有 99.92%。

Invocations	Metric Name	Metric Description	Min	Max	Avg
Device "NVIDIA GeForce GTX 1080 (0)"					
Kernel: phase3_all(int, int*, int)					
172	sm_efficiency	Multiprocessor Activity	99.88%	99.94%	99.92%
Kernel: test_phase1(int, int*, int)					
172	sm_efficiency	Multiprocessor Activity	3.33%	4.62%	4.60%
Kernel: phase2_combine(int, int*, int)					
172	sm_efficiency	Multiprocessor Activity	91.85%	96.38%	95.03%

iii. shared memory load/store throughput

在 phase3 我們需要對 shared memory 用到極高的 load 來進行運算。從 throughput 變得非常高可以看出，應該沒有 bank conflict 的問題，每個 thread 都能拿到、存放要用的資料。

Invocations	Metric Name	Metric Description	Min	Max	Avg
Device "NVIDIA GeForce GTX 1080 (0)"					
Kernel: phase3_all(int, int*, int)					
172	shared_load_throughput	Shared Memory Load Throughput	3152.8GB/s	3460.1GB/s	3418.5GB/s
Kernel: test_phase1(int, int*, int)					
172	shared_load_throughput	Shared Memory Load Throughput	106.39GB/s	128.71GB/s	125.82GB/s
Kernel: phase2_combine(int, int*, int)					
172	shared_load_throughput	Shared Memory Load Throughput	2087.5GB/s	2551.5GB/s	2485.0GB/s

```
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: phase3_all(int, int*, int)
        172                    shared_store_throughput           Shared Memory Store Throughput  256.79GB/s  282.05GB/s  278.83GB/s
    Kernel: test_phase1(int, int*, int)
        172                    shared_store_throughput           Shared Memory Store Throughput  35.543GB/s  43.288GB/s  42.281GB/s
    Kernel: phase2_combine(int, int*, int)
        172                    shared_store_throughput           Shared Memory Store Throughput  1456.9GB/s  1737.7GB/s  1690.6GB/s
```

iv. global load/store throughput

相比 iii.的 shared memory load/store，這邊的 throughput 明顯少了很多，phase 3 從 3418GB/s 降到 18GB/s、278GB/s 降到 68GB/s。可以看出我們在使用 global memory 時，速度是真的比 shared memory 慢，這也是為什麼用 shared memory 可以大幅 speedup 的關係。

```
==181305== Profiling application: ./hw3-2 /home/pp23/share/hw3-2/cases/p11k1 out_file
==181305== Profiling result:
==181305== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: phase3_all(int, int*, int)
        172                             gld_throughput                  Global Load Throughput  17.280GB/s  19.092GB/s  18.967GB/s
    Kernel: test_phase1(int, int*, int)
        172                             gld_throughput                  Global Load Throughput  322.94MB/s  392.18MB/s  383.35MB/s
    Kernel: phase2_combine(int, int*, int)
        172                             gld_throughput                  Global Load Throughput  12.694GB/s  15.123GB/s  14.698GB/s
```

```
==93793== Profiling application: ./hw3-2 /home/pp23/share/hw3-2/cases/p11k1 out_file
==93793== Profiling result:
==93793== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: phase3_all(int, int*, int)
        172                             gst_throughput                  Global Store Throughput  65.684GB/s  69.406GB/s  68.929GB/s
    Kernel: test_phase1(int, int*, int)
        172                             gst_throughput                  Global Store Throughput  596.17MB/s  659.81MB/s  649.79MB/s
    Kernel: phase2_combine(int, int*, int)
        172                             gst_throughput                  Global Store Throughput  23.076GB/s  25.258GB/s  24.906GB/s
```

3. Experiment & Analysis

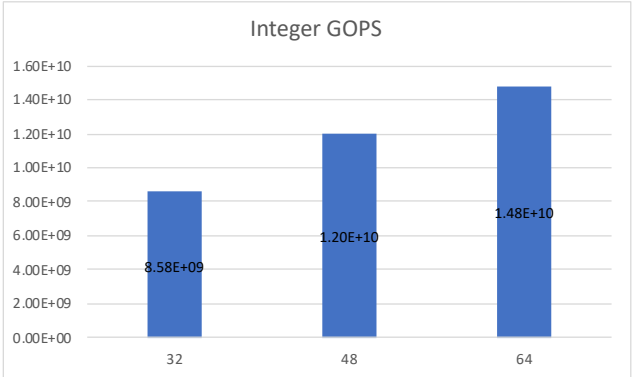
a. System Spec

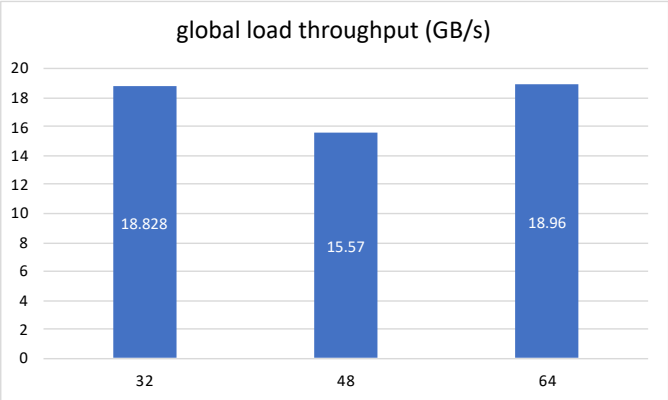
使用的是課程提供的 Hades server 作為測試的電腦。

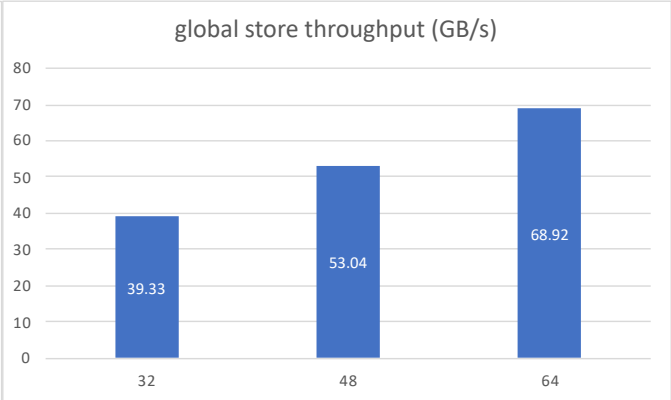
b. Blocking Factor: Observe what happened with different blocking factors, and plot the trend in terms of Integer GOPS and global/shared memory bandwidth

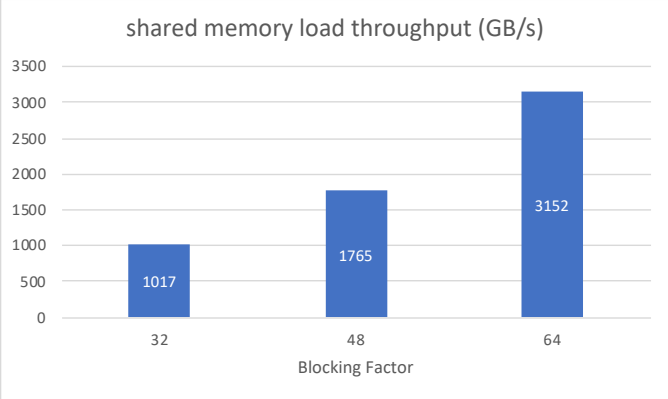
為了避免花過多的時間在 slurm 時 timeout，我只測了 3 個 blocking factor，分別是 32、48、64。因為我們重視的 speedup 在於 phase3，所以以下圖表皆以 phase3 的 avg 呈現。

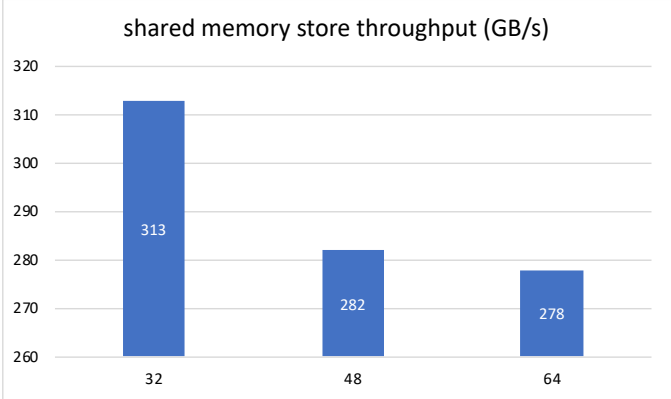
從 Integer GOPS 可以看出 block factor 64 是計算最快的情況。Global load throughput 沒差多少，代表 global load throughput 可以算是 bottleneck。Shared memory load throughput 可以看到當我們把 block factor 開到 64 時，bandwidth 變非常大，也可以說明 shared memory 的優化是很顯著的，才能讓效能提升。











c. Optimization

i. Coalesced memory access

由於一個 block 中的 thread(x, y)的方式是 thread(0, y), thread(1, y), … , thread(x, y)的方式一起做同個指令。所以當 memory access 要讓他們做拿連續的記憶體。如 Figure 4 所示，記憶體位置和 Thread 的顏色對應，也就是該 Thread 要取的格子，當我們用這個方式取記憶體時，效率高很多。但因為 memory 的 x, y 的 index 方式通常和 thread(x, y)的方式有點不同，所以要額外花心力去做。

雖說課堂上說最重要的是 global memory access 要用 Coalesced memory access；shared memory 的 access 也要盡量以 coalesced 的方式來做，會讓執行時間更快。

ii. Shared memory

將 global memory 的資料搬入__shared__中。

iii. CUDA 2D alignment

由於我們已經事先確認好我們的 Block factor 就是 64，所以可以靜態配置 2D 的 Shared Memory (__shared__ var[64][64])。我們本來也使用 2D 的 Thread Block，因此這個速度會比把 Array 攤平成 1D 還要來得快。

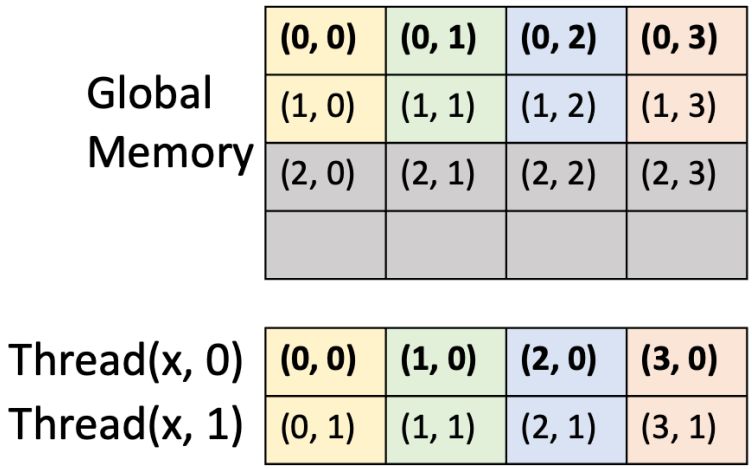


Figure 4. A thread access a memory cell with the same color and specified coordinate

iv. Occupancy optimization

在官網的定義中，Occupancy 代表一個 Stream Processor 中能支援的最大 warp 數中有多少個 active warp。而會讓 Occupancy 降低的原因有可能是因為 unbalanced workload within blocks、unbalanced workload cross blocks、too few blocks launched。而這次程式碼中，我們讓每個 thread 在 block 中負責 4 個點，且我使用了一個 myMin()的 function，可以避免 warp divergence；而不同 block 的都是計算固定 64*64 的數量；最後則是我們已經把所有 block 對應到該計算的 global memory 上，因此已經把最多的 block 開出來了。因此我認為有利用到 Occupancy optimization。

v. Large blocking factor

當 Blocking factor 升到 64 時，我們利用 thread 去把 global memory 的 data 搬入 shared memory，減少 global memory access。

如 Figure 5 左可以看到當 block factor = 1 時，總共要執行 8 rounds。在 phase 3 時（只討論 phase 3 因為這是最花時間的階段），整個 matrix 的格子都要去取 pivot row/column 的記憶體資料。在 round 1 時，單看 1 column，以 column 2 為例， (1, 2)要被整個 column 的 thread 去取資料，一個有 8 個 thread 要搶 global memory。

相對的如果 block factor = 2 時，(2, 4)只會被(0, 4), (4, 4), (6, 4)這三個(方便計算我們估為 4 個) thread 去 access 到。計算左圖的總共 memory access 次數為 8 rounds * 8 access = 64，而右圖則為 4 * 4 = 16，當 block factor 乘 b 時，memory access 次數下降 b 平方！是非常重要的 speedup。

而當我們只能有 1024(32 * 32) threads per block 要用下述方式達到 block 變成 64 * 64：每個 thread block 依舊只有 32 * 32 個 thread，但是每個 thread 要負責 4 個格子的運算。以 Figure 5 右圖為例，假設 thread per block = 1 卻要 block factor = 2，則以深綠色格子而言，有一個 thread 他本來要負責(2, 4)的運算，現在要負責(2, 4), (2, 5), (3, 4), (3, 5)的運算。Block 從原本 1*1 的大小變成 2*2 = 4。

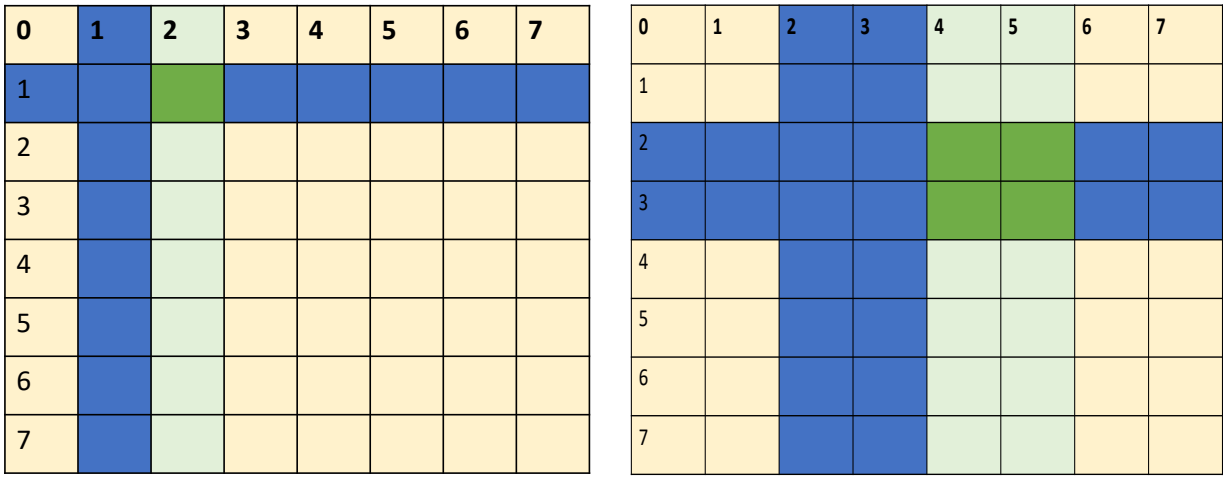


Figure 5. Large Blocking Factor makes less memory access in each round and in total

d. Weak scalability (hw3-3)

Weak Scalability 代表當資料量增加且運算資源增加時，runtime 會隨 core 數增加呈 constant。我選擇 p25k1 和 p31k1 作為 testcases，會讓 1GPU 去計算 p25k1，用 2GPU 運算 p31k1。由於運算量是 input size 的三次方，所以要挑選的兩個 testcases，假設第一個 input size 是 x，則第二個 input size x' 則為 $x'^3 = (x^3)*2$。P25k1 的 input size 為 25000，而 p31k1 為 31000。統計的時間為 Memcpy 和 3 個 phase(Compute) 的運算總和。

依下圖來看，總時間不減反增。我想這是因為兩張 GPU 互相傳遞資料的時間是無法被平行化的，且相較於運算是比較慢的，自然很難做到完美的平行。Phase1, 2, 3 時，當資料量成兩倍時，雖然工作量應該要被兩張 GPU 平分，但是因為我們還沒有計算到每個 block 要去 access global memory 的時間，因此也可能導致了下圖藍色 Compute 的時間增長。

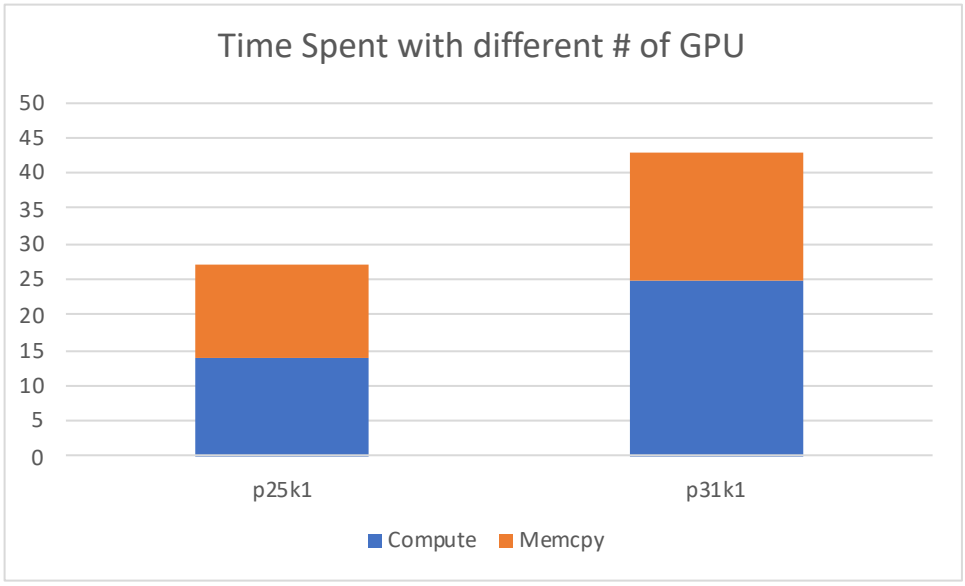


Figure 6. Weak Scalability of CudaMemcpy() and the 3 Phases

e. Time Distribution (hw3-2)

i. computing

用 nvprof 中的 3 個 phase 做加總計算。

ii. communication

Single-GPU 故為 0。

iii. memory copy (H2D, D2H)

用 nvprof 中的 cudaMemcpy 值。

iv. I/O of your program w.r.t. input size.

用 clock_gettime(CLOCK_MONOTONIC, &v)；計算 input read 和 output write 的時間。

使用以下 3 個 testcases 做比較：

	# of Vertex	# of Edges
p11k1	11000	505586
p21k1	20959	987205
p30k1	30000	3907489

由於考量到我們運算的是矩陣，因此當 input size * 2 時，I/O 和 Memory Copy 工作量要乘以 $2^2 = 4$ 倍，運算量則是 $2^3 = 8$ 倍。

以 I/O time 而言，可以看到 p11k1 和 p21k1 差了 4 倍的寫入量，所以 I/O time 也大約是 4 倍；p21k1 和 p30k1 相比差了兩倍多的寫入量，I/O time 也接近 2 倍。

Memory Copy 在 p21k1 和 p30k1 差了約兩倍，也和我們的想法相同。我認為如果在 D2H 的時候如果可以用 Streaming 的 optimization 方法說不定可以減少 I/O time 和 Memory Copy time 的總時間，因為當 D2H 時，CPU 是有時間把 Host 上的資料寫入另外的 Disk 上。

Computing time 隨著 input size 增加而有顯著成長，從 p11k1 到 p21k1 差了快 8 倍，p21k1 到 p30k1 差了 3 倍。在計算 nvprof 的 profile 時，phase3 佔了超過 99%的時間，故可以知道 phase3 的 speedup 是影響效能的關鍵因素。

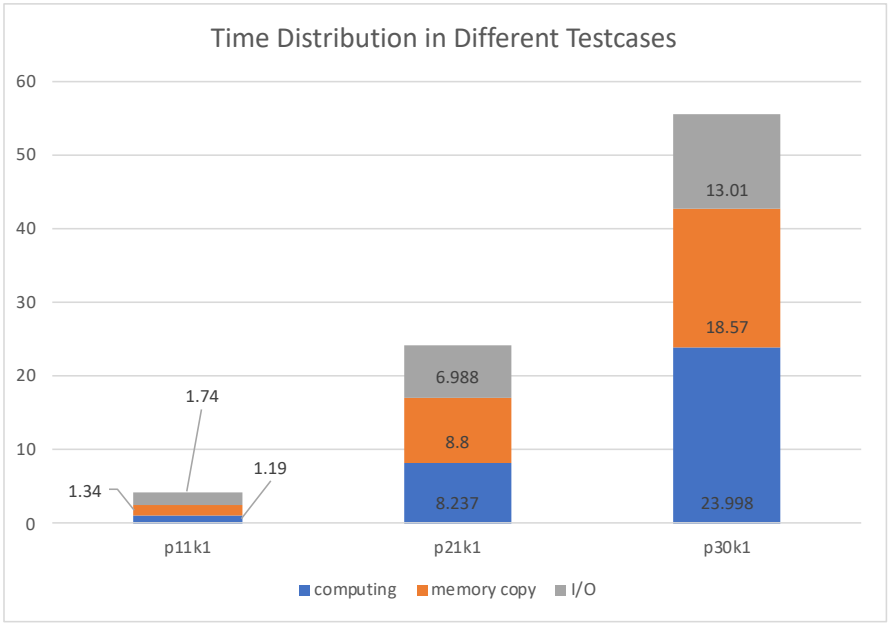


Figure 7. Time Distribution in Different Testcases

f. Others

Figure 8 是累積的優化方法所達成的時間改善圖，最基本的是我們把 global memory 的東西搬到 shared memory 然後計算；接著是把 access global memory 和 shared memory 的方法改成 coalesced 的技巧；然後我們為了減少 global memory access、提升 shared memory 使用率，因此用了 64*64 的 blocking factor。最後是把原本是一維的 Shared Memory 改成 2 維。

出乎意料的是 2D Shared Memory 可以讓時間大幅縮小，我認為是因為 2D 可以讓程式更清楚哪個 thread 要去 access 哪裏，減少 bank conflict 並增加 coalesced 的能力，才可以把前面所使用的 large blocking factor 和 Coalesced Memory 的效益累加的提升。從 Figure 8 的下圖也可以看到 2D Shared Memory 大幅增加了 Load Throughput，證實了這個技巧能充分發揮 Shared Memory 的加速能力。

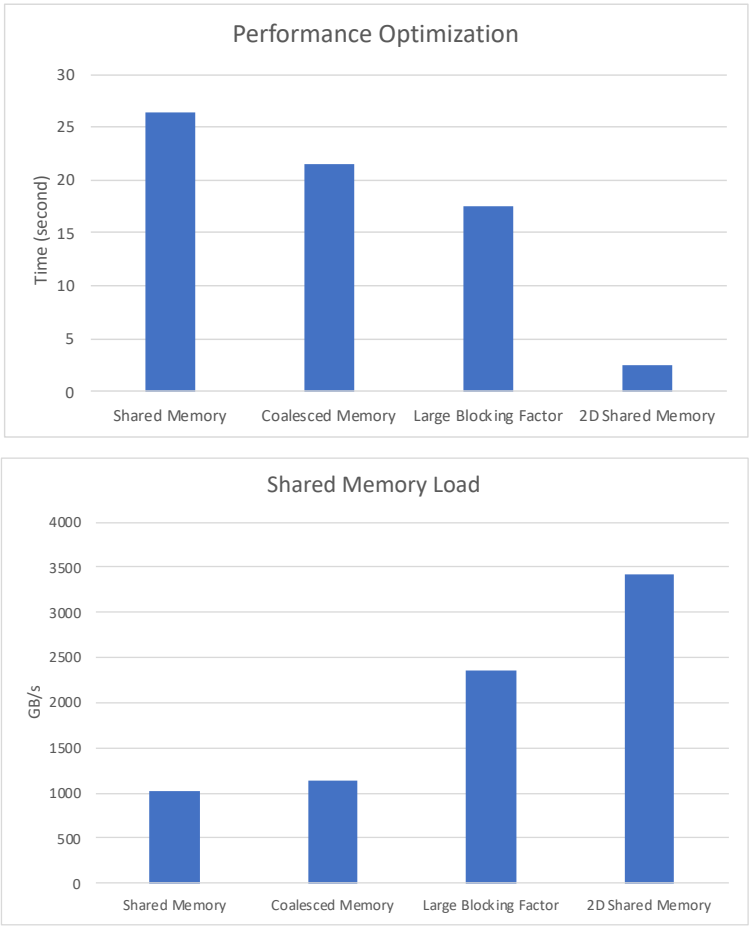


Figure 8. Optimization Method

4. Experience & conclusion

這次的作業學到了很多寫 Cuda 的技巧。上課中覺得許多理所當然的東西如 Coalesced memory access，真的要實際遇到時才發現不好實作。我一開始為了讓 warp 的 x 座標去對應 global memory 的 x 座標花了非常多時間才搞懂，而且更難的是要讓 shared memory 的存取也要 coalesced。而且這些東西除了真的下去測時間以外，實在很難知道到底有沒有寫對。

還有一些小地方例如 I/O 用 linear loop 的寫入方式會比 n^2 的 for loop 寫入方式還快；__synthreads()可以在哪裡少放等等，都會改變速度。我聽說要盡量避免 branch，所以寫了一個 myMin()，這個 function 會增加運算量，有比較多的乘法和加法，但可以避免發生 branch，測了幾次總時間好像沒差太多，所以應該 bottleneck 還是出現在 memory 的地方。

好不容易把 Coalesced memory access 做出來後卻發現 pxxk1 許多 testcase 會超時，那時候真的以為這個 optimization 失敗了。好險後來發現 large factor 和 shared memory 2D alignment 的方法，才讓程式執行得更快。當看到 AC 時，真的覺得一切辛苦都值得了。