平行程式 Parallel Programming – HW4: Blocked All-Pairs Shortest Path

姓名：蕭子聲　　學號：103062372

1、 Implementation

(a) Cuda 的部分，我是使用標準的 32 x 32 的 block 來做計算，沒有做特殊的形狀分割或是任何 stream 優化。總共分成 3 個 phase，第一個 phase 計算 Pivot block，第二個 phase 則是將 row 與 column 合併計算，在 kernel 裡面區分。第三的 phase 計算所有其他的 block，並在 kernel 內判斷是否是 phase 1 或 2 時計算過的 block，是則結束 block。
詳細優化過程請看 4、Optimization。

(b) OpenMP 的部分，我是將一整個完整的 Adjacent matrix 切成 2 x 2 的 block，再將 block 分別平行到兩張 GPU 上，將 Block 分別定義為 G11、G12、G21、G22。則兩個 GPU 執行的流程分別如下：

Stream 0	Stream1
G11	G11
G12	G21
----------sync	
G22	G22
G12	G21
----------sync	
G11	G11
----------sync to host	

利用這個方法，能夠將原本需要每一回合同步的次數縮減到只需要 3 次，雖然仍有大約 2/3 的 block 重疊到，但是已經足以得到比單片 GPU 還佳的效果，不過由於這個方法還是有加速上限而沒辦法達到更理想的數字，因此後續還是會繼續研究其他方法來更加提升速度，但就不會列入這篇 Report。詳細圖解請看 7、Appendix (a)

(c) MPI 方法同 OpenMP，但是因為 MPI 跨兩個 processor，加上還需要在每個階段做同步與資料傳輸，在效能上稍微比 OpenMP 還要慢，小測資也會比單片 GPU 稍微慢，但是當測資足夠大時 (5.in)，還是能夠達到比單片 GPU 快的速度。

2、 Working Items

(a) Single-GPU:

[✓] Achieve better performance than sequential Floyd-Warshall implement.

(b) Multi-GPU implementation with OpenMP:

[✓] Able to utilize multiple GPUs available on single node.

[✓] Achieve better performance than single GPU version.

(c) Multi-GPU implementation with MPI:

[✓] Able to utilize multiple GPUs available on multi node.

[✓] Achieve better performance than single GPU version on single node.

3、 Profiling Results

(a) Cuda version, 5.in, block size = 32

請看 7、Appendix (b)

(b) OpenMP version, 5.in, block size = 32

請看 7、Appendix (c)

(c) MPI version, 5.in, block size = 32

請看 7、Appendix (d)

4、 Optimization

以下的時間計算，均使用測資 5，在 Hades05 上面運行。IO / Read 為將整個 File 讀取到 string buffer 內所花的時間，IO / Write 為總輸出時間，Input Parsing 則是將 string buffer 內的東西讀取到 Matrix 的時間。Cuda Memcpy 為花費在 Cuda HtoD / DtoH 的時間。Calculation 為實際計算所有 APSP 的時間，Phase 3 為 APSP phase 3 所花的時間。Total 則是整個程式執行的時間。

(a) Cuda original version

這個版本是最原始的第一版，使用 Global Memory，沒有做任何優化。

Operation	Time
IO / Read	528.05 ms
IO / Write	3085.99 ms
Input Parsing	12694.20 ms
Cuda Memcpy (HtoD/DtoH)	250.68 ms / 244.88 ms
Calculation	99958.95 ms
Phase 3	99.04 s
Total	116384.371 ms

(b) Cuda shared memory version

所有 Kernel 改用 Shared memory，並且使用 MallocPitch 來配置最佳的 GPU memory 大小。利用 block_size = 32 的優勢以及重新排列 Shared memory 的順序來解決 Bank conflict 的問題。實際是使用 nvprof 的 –events shared_ld_bank_conflict 以及 shared_st_bank_conflict 來確定解決掉 Shared memory bank conflict 的問題。

Operation	Time
IO / Read	531.92 ms
IO / Write	3030.75 ms
Input Parsing	12991.79 ms
Cuda Memcpy (HtoD/DtoH)	251.61 ms / 244.72 ms
Calculation	40529.90 ms
Phase 3	39.75 s
Total	57196.19 ms

(c) Cuda optimize phase 3 / optimized IO

1、 這一版除了優化 Input 速度外，我還發現一件事：Phase 3 計算 APSP 其實不需要每一步都 Sync，因此將 for loop 內的 __syncthreads() 移除掉，得到更快的速度。

2、 另外還有利用 template 的方式，針對 block size 16 / 32 做 Unroll 優化。

3、 Host memory 改成使用 Pinned memory。

4、 將在 Launch kernel 前就可以事先計算好的變數改成用 CPU 計算，並當作參數直接傳入 kernel。

5、 能夠減少 global memory access 次數就盡可能減少。例如下圖。

```
s_l[threadIdx.y][threadIdx.x] = (mc < vert && lr < vert) ? dist[mc * width + lr] : INF;
s_r[threadIdx.y][threadIdx.x] = (rc < vert && mr < vert) ? dist[rc * width + mr] : INF;
```

Operation	Time
IO / Read	529.75 ms
IO / Write	2986.86 ms
Input Parsing	2043.44 ms
Cuda Memcpy (HtoD/DtoH)	246.54 ms / 242.65 ms
Calculation	18152.32 ms
Phase 3	17.55 s
Total	23717.15 ms

(d) Cuda optimize kernel (最後大絕招)

通常做到上一步後，實際 submit 的時間會是 20~22 秒左右。接下來有很多人以為到這邊 phase 3 的 kernel 已經到最簡化了，共存取兩次 global memory (讀 / 寫)來更新自己的 Cell。但實際上其實並沒有需要存取到兩次 memory，大部分時候只要存取一次就可以了。另外，在 access memory 時，藉由重新排序 code 的順序，能夠減少 thread 等待 memory 讀取完成的時間，讓 thread 在等待的過程可以先做其他的計算。這個可以使用 nvprof 的 –matrics stall_memory_dependency 來查看有沒有發生

thread 等待 memory 讀取的狀況。最後簡化的 code 如下：

```
int o = dist[m_cell];
int n;
int mn = s_l[threadIdx.x][0] + s_r[0][threadIdx.y];
for(int k=1;k<block_size;++k){
    n = s_l[threadIdx.x][k] + s_r[k][threadIdx.y];
    if( n < mn) mn = n;
}
if(mn < o)
    dist[m_cell] = mn;
```

可以看出在第一行的地方存取 global memory 放到變數 o，但是變數 o 實際用到是在第八行的地方，這中間 Load global memory 的時間都可以拿來計算 APSP，如果一開始就先將 Global memory load 到變數 mn，然後直接在 for 迴圈內更新就容易發生 memory dependency 的問題，讓 thread 空等在那直到 Global memory 讀取到為止。且在第三行與第五行的地方能夠讓 Shared memory 同時先 Load 到不同的 Reg 來暫存，最後就是判斷有更新時才會再寫回 Global memory，這可以有效減少很多次 Global Memory 存取的次數。最後順利在沒有使用任何 Stream 的情形下，利用一般的 Blocked APSP 方法 submit 時間縮到 18 秒。

除此之外還有再優化過 IO，將 char 的 case 減少剩 3 種，分別為空白、換行，其餘則都當作是數字。

Operation	Time
IO / Read	533.57 ms
IO / Write	3044.53 ms
Input Parsing	1848.17 ms
Cuda Memcpy (HtoD/DtoH)	246.82 ms / 242.89 ms
Calculation	16448.81 ms
Phase 3	15.83 s
Total	21879.06 ms

(e) 總結 (5.in)

下表為每個 Step，Phase 3 kernel 的加速狀況。

Step	Kernel Time	Step Speed up	Cumulative Speed up
Original version	99.04 s	--	--
Shared memory	39.75 s	2.49x	2.49x
Optimize	17.55 s	2.26x	5.64x
Dependency	15.83 s	1.11x	6.26x

下表為整體程式運行的加速狀況，Elapsed time 為上面所有表格的估計時間，因此時間會與實際程式執行時間有一點誤差。

Step	Elapsed Time	Step Speed up	Cumulative Speed up
Sequential	4624.31 s	--	--
Original version	116.384 s	39.73x	39.73x
Shared memory	57.196 s	2.034x	80.85x
Optimize	23.717 s	2.411x	194.97x
Dependency	21.879 s	1.084x	211.36x

(f) OpenMP version

OpenMP 版本使用兩個 Stream 的方式來實現兩片 GPU 的平行，由於上述的平行方式會需要將 Phase 3 拆的更零碎，加上有部分會是平行執行的因此 Kernel 時間會不太正確。另外 Input Parsing 是使用兩個 thread 做 parsing。

Operation	Time
IO / Read	557.37 ms
IO / Write	3067.53 ms
Input Parsing	1318.50 ms
Cuda Memcpy (HtoD/DtoH)	251.34 ms / 121.61 ms
Phase 3	21.62 s
GPU to GPU communication	542.4 ms
Total	17994.79 ms

5、 Experiment & Analysis

除上列之外，我還有另外做了一些實驗，但是加速效果不太明顯 因此就沒有再繼續專研下去。

(a) 增加 Reg 使用量

由於 Reg memory 的存取速度遠遠快於 shared memory，且 GTX1080 的 register per block 有 65536 個，也就是說，若以 32x32 的 block size 來說，每一個 thread 能夠分到 64 個 Reg。因此讓我產生這個想法：如果先將所有 shared memory 結果加起來存到 Reg 之後再進行比較應該會比較快。因此將 kernel 改寫如下：

```
int n[block_size];
int o = dist[m_cell];
for(int k=0;k<block_size;++k){
    n[k] = s_l[threadIdx.y][k] + s_r[k][threadIdx.x];
}
for(int k=0;k<block_size;++k){
    if(n[k] < n[0]) n[0] = n[k];
}
if(n[0] < o)
    dist[m_cell] = n[0];
```

其中 block_size 為 constant 因此 compile 時會自動 Unroll。原本預想的結果為從原本使用 19 個 Reg 會增加到 40、50 個，結果用 nvprof print-gpu-trace 的結果，Reg 數量只從原本的 19 增加為 20，表示這段 Code 被 Compiler 優化後會被轉換為接近原本的 Code 的形式，因此最後實測的結果 Kernel 並沒有獲得加速。

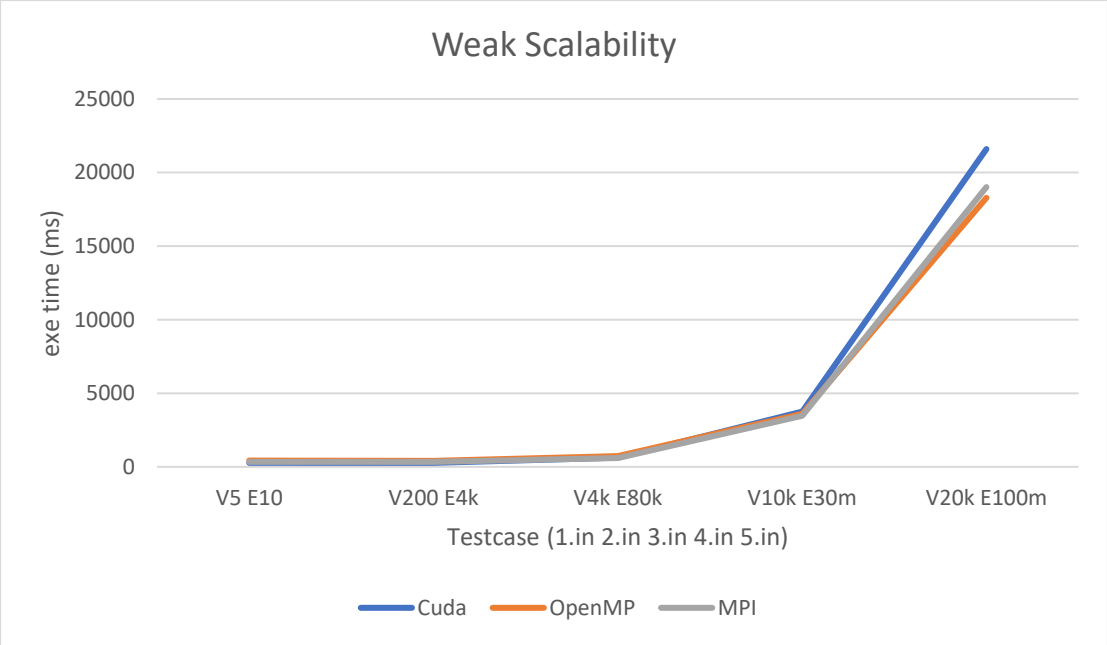
(b) another multi-GPU version

目前還有在設計另外一個 Multi-GPU 的版本，也就是同樣是每一回合 Sync 但是實際上他只需要 sync 一行就可以了，這個方法理論上能夠跑得比這個版本還要快。

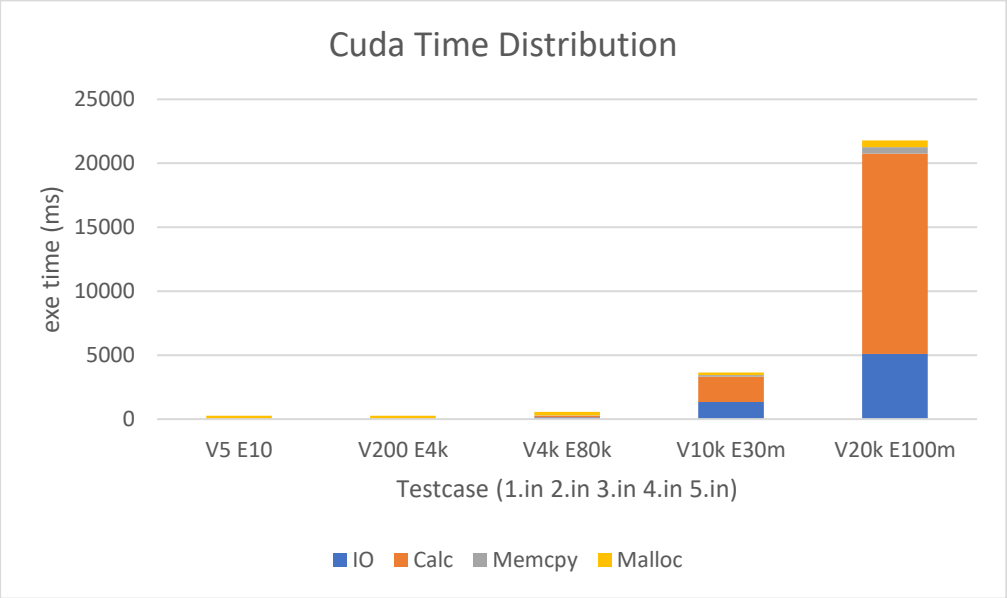
(c) System Spec

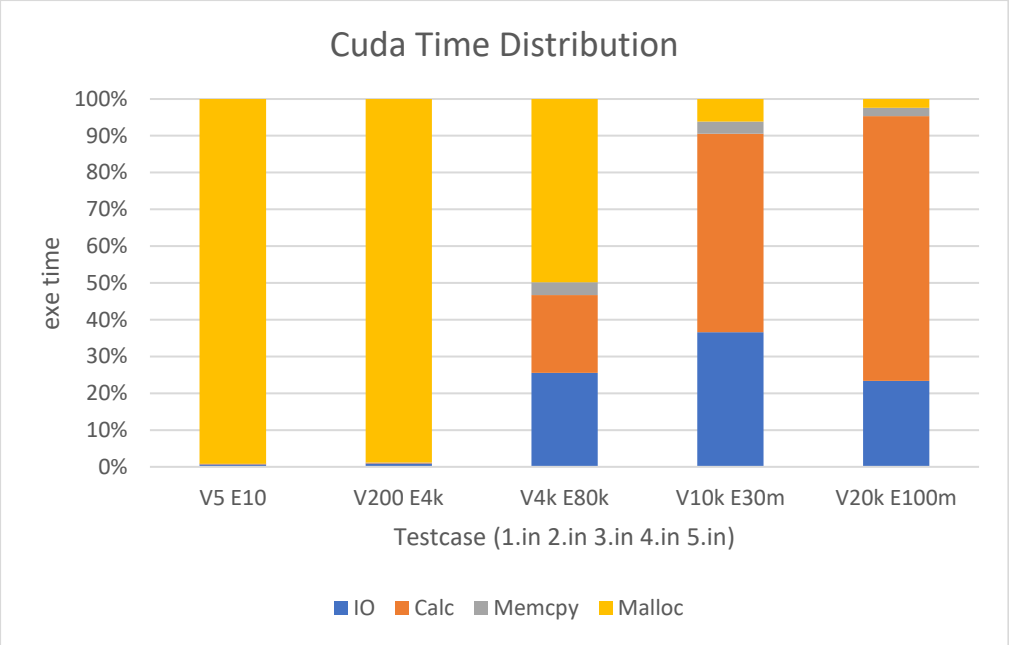
皆使用 Hades 05 測量。

(d) Weak Scalability & Time Distribution

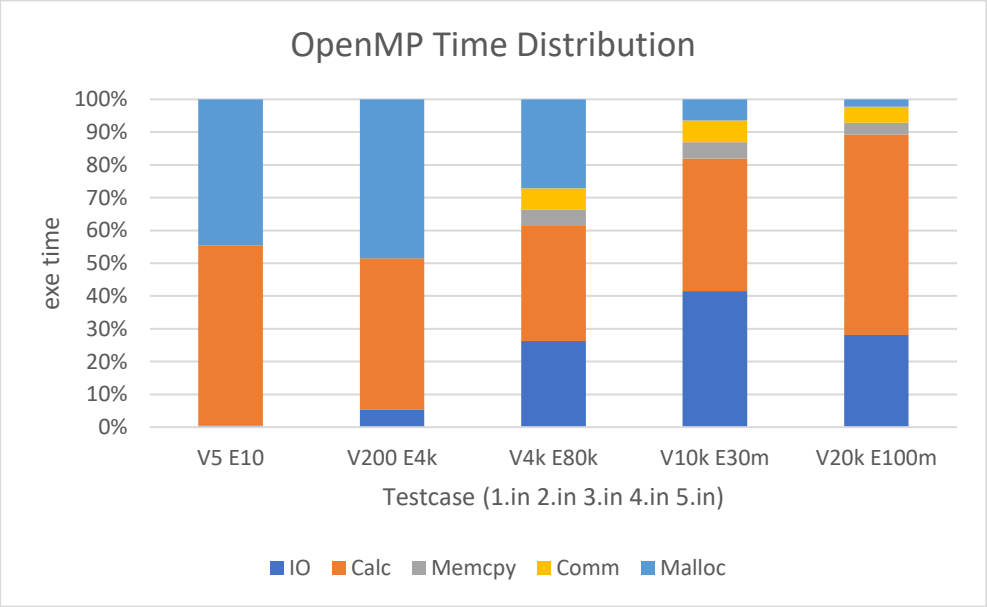


由上圖可以看得出來，OpenMP、MPI 還是有加一點點速，但是以整體效能上比較起來還是表現的非常的差。

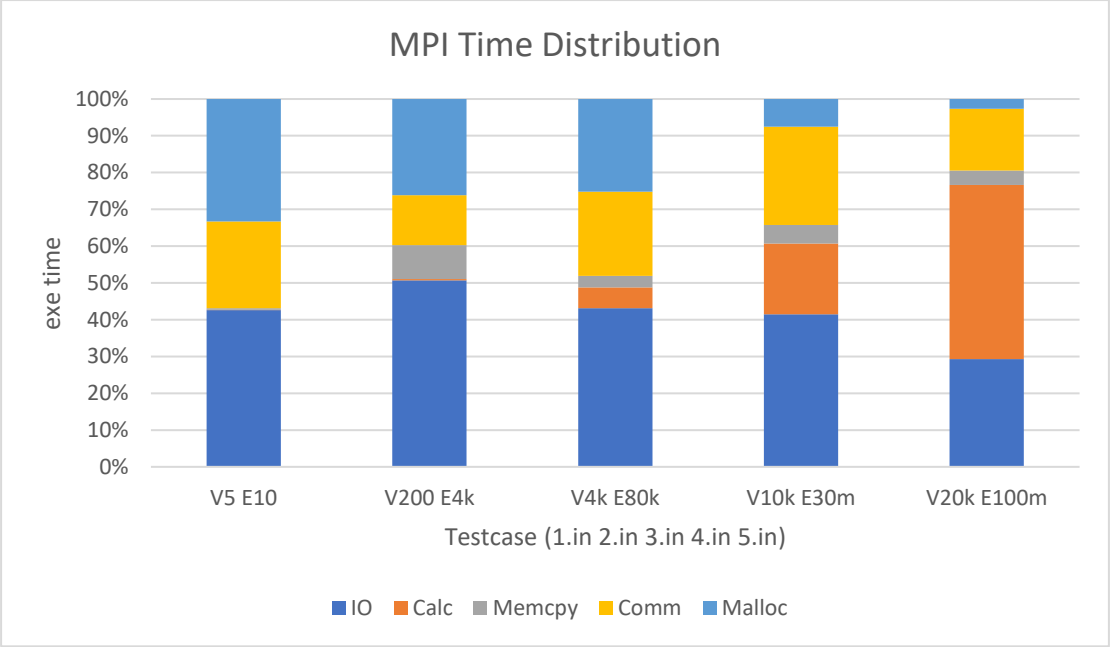




由上面兩張圖可以看的出來，Memory allocate 其實是很花時間的，小測資大部分的時間會花在 Memory allocate 上面。隨著測資漸漸變大，除了 IO 會變長外，計算時間也會成指數倍數上漲，到測資 5 時，IO 所佔的時間比例會變得更少。

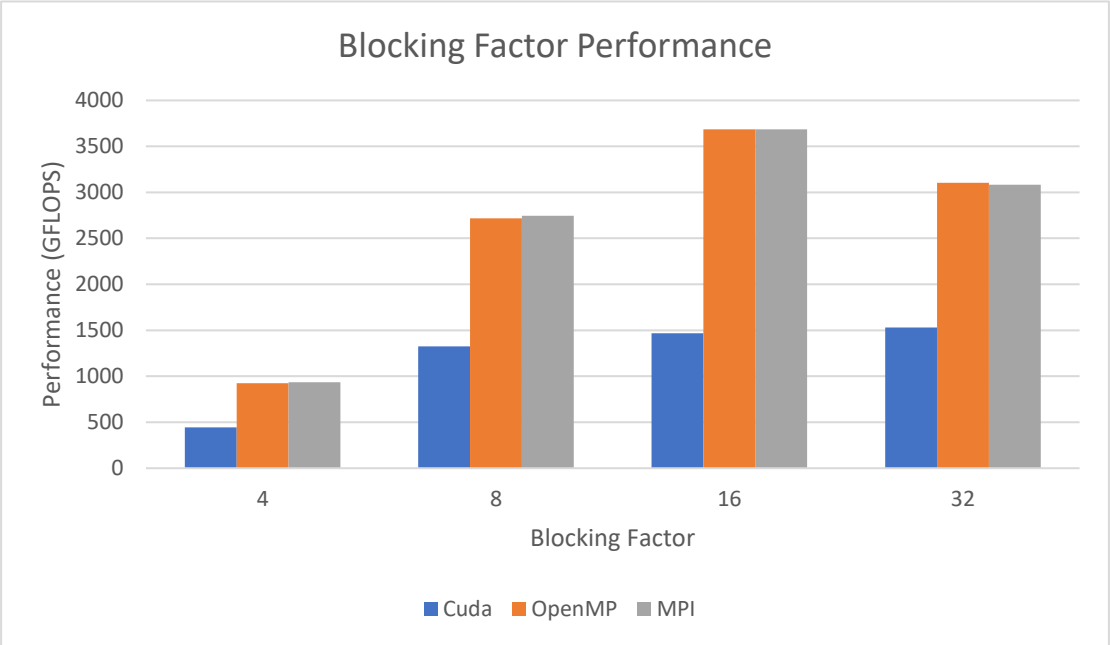


同樣由上圖可以看出，一開最小的測資幾乎會被 Malloc 佔據，隨著測資增大，IO 與計算會佔比例愈多，直到測資 5 時，計算的時間會遠過於 IO 時間。而 Comm 的時間則影響不大。



計算與 IO 的關係可以看到與 Cuda、OpenMP 相同的現象，另外可以看到 Comm 的時間比例佔的會比 OpenMP 的還要多，主要是因為 MPI 是 Process 與 Process 之間的溝通，會比 OpenMP 還要慢，除此之外也可以看出，在不同測資下，Comm 的時間比例基本上是接近於一個固定的值。

(e) Blocking Factor

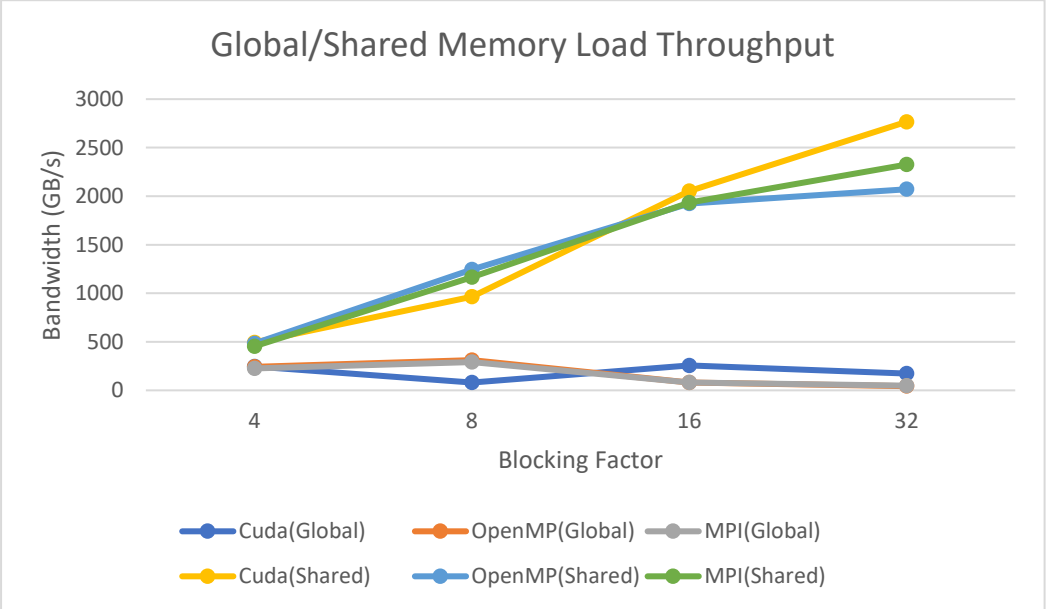


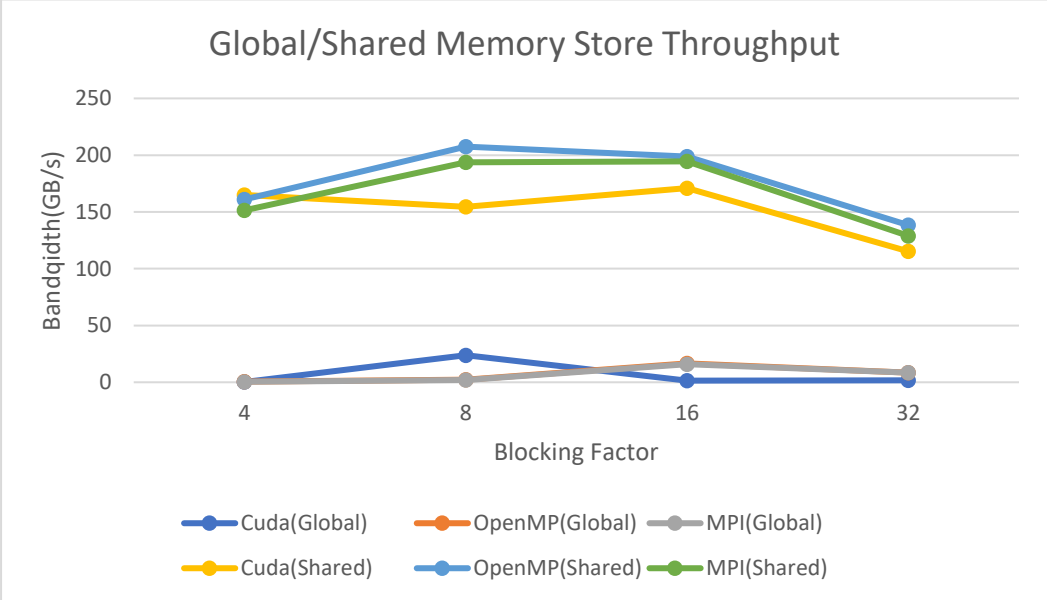
不同 Blocking Factor 的 Performance 如上圖，GFLOPS 的算法為

$$\text{GFLOPS} = \frac{\text{N}}{\text{T}} \times 10^{-9}\ (Gflops/sec)$$

其中 N 為 Total number of integer instructions for all kernels，T 為 Total average exe

time for all kernels 單位換算為秒，之後再除以 10 的 -9 次方換算為 Giga。測量的測資使用 5.in。可以看到 OpenMP 跟 MPI 完全吻合，不過這也是滿廢話的，我兩個版本用一樣的 kernel 當然會一模一樣，Cuda 版本用不一樣的 Kernel 可以看的出來，在 Cuda 版下當 Blocking Factor 為 32 時會有最佳的 GFLOPS，而 OpenMP、MPI 版本則是在 16 時會有最佳的 GFLOPS，但是實測出來卻是 Blocking Factor 為 32 時速度會最快，由此可見，效能的好壞跟 GFLOPS 不一定有關連，可能還有其他 Bottle neck 存在。





6、 Conclusion

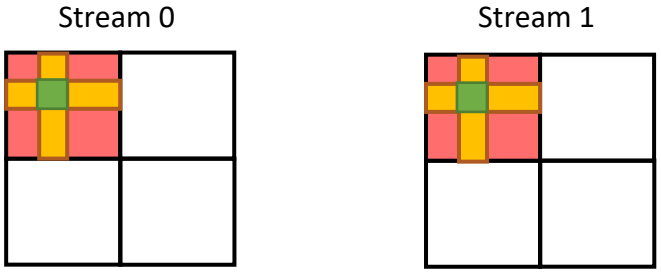
這次的作業非常有趣，想當初一開始拿到 Spec 時完全看不懂 Blocked APSP 到底在做什麼鬼東西，甚至還一度懷疑是不是有些地方有寫錯，後來實際開始 Coding 的時候邊手繪模擬邊絞盡腦汁歸納出規律後才漸漸了解 Blocked APSP 其實是一個很偉大的演算法！當很努力的刻出第一版 (Global Memory) 寫出來發現測資五只有 100 多秒的時候那感覺真的非常低落，但是後來在修改 Code 的過程中，逐漸了解到 Cuda 的運作原理，Blocked APSP 的計算原理後，那個成就感真的是非常大。

過程中遇到的困難像是，一開始要了解 Blocked APSP 的原理以及 Cuda 的運作方式就覺得非常吃力，但是這個就要藉由不斷的觀察、思考解決，另外比較麻煩的是作業期間，一直受到其他科目像是 Deep Learning 之類的課程干擾，沒辦法很專心、投入的去寫這次的作業，對於這件事情我深感非常遺憾。

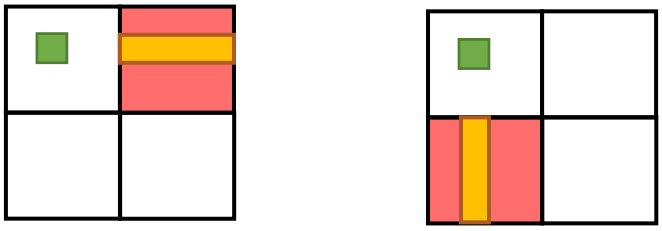
7、 Appendix

(a) 這邊講解 OpenMP 的更新方式：

第一步、兩邊都先對 G11 做一般的 APSP 更新

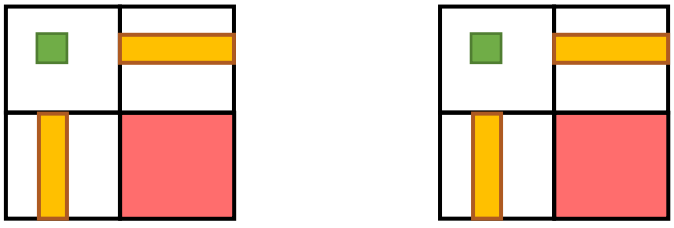


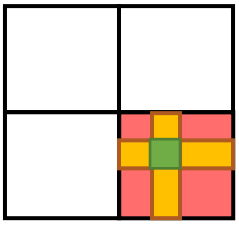
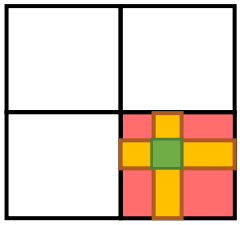
第二步、兩邊各做 G12 與 G21 更新



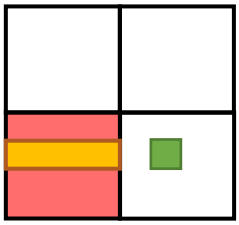
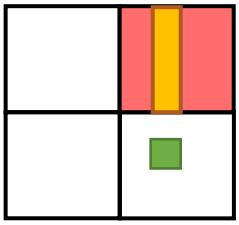
-------同步-------

第三步、同時更新 G22



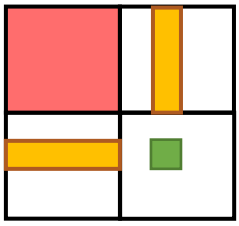



第四步、各自更新 G12、G21

--------同步--------

第五步、更新 G11



-------同步到 Host-------

(b) Cuda nvprof

```
[p103062372@hades01 HW4]$ srun -p pp --gres=gpu:1 -N1 -n1 -c1 nvprof ./HW4_cuda testcase/5.in 5.out 32
==499== NVPROF is profiling process 499, command: ./HW4_cuda testcase/5.in 5.out 32
==499== Profiling application: ./HW4_cuda testcase/5.in 5.out 32
[LOG] Step   0: Timers                          Elapsed time
[LOG] Step   1: block                           16104.960898 ms
[LOG] Step   2: finalize                        0.014097 ms
[LOG] Step   3: init                            4652.768321 ms
[LOG] Step   4: init/init_mat                   119.963624 ms
[LOG] Step   5: init/parse_int                  1852.621393 ms
[LOG] Step   6: init/read_file                  2678.084034 ms
[LOG] Step   7: write_file                      2843.900399 ms
==499== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   96.20%  15.4899s       625  24.784ms  24.451ms  25.622ms  void phase_three<int=32>(int*, int, int, int, int)
                    1.53%  246.56ms         1  246.56ms  246.56ms  246.56ms  [CUDA memcpy HtoD]
                    1.51%  242.60ms         1  242.60ms  242.60ms  242.60ms  [CUDA memcpy DtoH]
                    0.74%  119.72ms       625  191.54us  190.43us  202.85us  void phase_two<int=32>(int*, int, int, int, int)
                    0.02%  2.8300ms       625  4.5270us  4.3840us  7.4560us  void phase_one<int=32>(int*, int, int, int, int)
      API calls:   51.60%  8.52117s         1  8.52117s  8.52117s  8.52117s  cudaDeviceSynchronize
                   42.93%  7.08991s      1875  3.7813ms  3.4000us  25.235ms  cudaLaunch
                    2.96%  489.21ms         2  244.61ms  242.62ms  246.60ms  cudaMemcpy2D
                    2.48%  410.13ms         1  410.13ms  410.13ms  410.13ms  cudaHostAlloc
                    0.01%  1.1294ms         1  1.1294ms  1.1294ms  1.1294ms  cudaMallocPitch
                    0.01%  960.30us      9375     102ns      87ns  3.1700us  cudaSetupArgument
                    0.00%  818.57us         2  409.28us  8.2340us  810.33us  cudaFree
                    0.00%  461.96us        94  4.9140us     108ns  223.29us  cuDeviceGetAttribute
                    0.00%  264.58us      1875     141ns     115ns     868ns  cudaConfigureCall
                    0.00%  108.62us         1  108.62us  108.62us  108.62us  cuDeviceTotalMem
                    0.00%  40.309us         1  40.309us  40.309us  40.309us  cuDeviceGetName
                    0.00%  2.2240us         3     741ns     158ns  1.8520us  cuDeviceGetCount
                    0.00%     643ns         2     321ns     129ns     514ns  cuDeviceGet
[p103062372@hades01 HW4]$
```

(c) OpenMP nvprof

```
[p103062372@hades01 HW4]$ srun -p pp --gres=gpu:2 -N1 -n1 -c2 nvprof ./HW4_openmp testcase/5.in 5.out 32
==530== NVPROF is profiling process 530, command: ./HW4_openmp testcase/5.in 5.out 32
[LOG] Step   0: [thread 1] sp: 10000, Round: 313
[LOG] Step   0: [thread 0] sp: 10000, Round: 313
[LOG] Step   2: Timers                          Elapsed time
[LOG] Step   3: block                           22322.441899 ms
[LOG] Step   4: finalize                        0.014129 ms
[LOG] Step   5: init                            2933.374428 ms
[LOG] Step   6: init/init_mat                   120.037039 ms
[LOG] Step   7: init/parse_int                  2247.496794 ms
[LOG] Step   8: init/read_file                  563.769230 ms
[LOG] Step   9: write_file                      2996.089943 ms
==530== Profiling application: ./HW4_openmp testcase/5.in 5.out 32
==530== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   60.82%  13.6566s      2191  6.2330ms  5.9724ms  6.3236ms  void phase_three<int=32>(int*, int, int, int, int, int, int)
                   17.56%  3.94254s       626  6.2980ms  6.0522ms  6.4428ms  void phase_three_h<int=32>(int*, int, int, int, int, int, int, int, int)
                   17.37%  3.89986s       626  6.2298ms  6.0310ms  6.5135ms  void phase_three_v<int=32>(int*, int, int, int, int, int, int, int, int)
                    2.20%  493.34ms         2  246.67ms  246.43ms  246.90ms  [CUDA memcpy HtoD]
                    1.09%  243.76ms         2  121.88ms  121.55ms  122.22ms  [CUDA memcpy DtoH]
                    0.56%  125.46ms      1252  100.21us  80.896us  107.11us  void phase_two<int=32>(int*, int, int, int, int, int)
                    0.14%  32.378ms       626  51.721us  49.409us  58.081us  void phase_two_v<int=32>(int*, int, int, int, int, int, int)
                    0.14%  32.360ms       626  51.692us  31.713us  58.912us  void phase_two_h<int=32>(int*, int, int, int, int, int, int)
                    0.09%  19.869ms         3  6.6230ms  6.6221ms  6.6243ms  [CUDA memcpy DtoD]
                    0.03%  6.1817ms      1252  4.9370us  3.2960us  19.232us  void phase_one<int=32>(int*, int, int, int)
      API calls:   77.29%  34.9022s      7199  4.8482ms  3.8840us  2.18645s  cudaLaunch
                   21.46%  9.68963s         4  2.42241s  493.27ms  4.39236s  cudaDeviceSynchronize
                    1.17%  527.97ms         1  527.97ms  527.97ms  527.97ms  cudaHostAlloc
                    0.03%  14.383ms         7  2.0547ms  7.8670us  6.3873ms  cudaMemcpy2DAsync
                    0.02%  7.9200ms         3  2.6400ms  8.3280us  7.0717ms  cudaFree
                    0.02%  7.0573ms     47889     147ns      86ns  215.21us  cudaSetupArgument
                    0.00%  2.2442ms         2  1.1221ms  1.0884ms  1.1559ms  cudaMallocPitch
                    0.00%  1.6082ms      7199     223ns     125ns  15.275us  cudaConfigureCall
                    0.00%  1.3123ms         2  656.13us  59.854us  1.2524ms  cudaStreamCreate
                    0.00%  855.66us       188  4.5510us     119ns  199.32us  cuDeviceGetAttribute
                    0.00%  238.91us         2  119.45us  115.63us  123.28us  cuDeviceTotalMem
                    0.00%  80.800us         2  40.400us  38.429us  42.371us  cuDeviceGetName
                    0.00%  37.436us         2  18.718us  16.331us  21.105us  cudaStreamDestroy
                    0.00%  31.019us         4  7.7540us     620ns  26.498us  cudaEventCreate
                    0.00%  9.5910us         2  4.7950us  3.5460us  6.0450us  cudaSetDevice
                    0.00%  9.2920us         3  3.0970us  2.5660us  3.6950us  cudaEventRecord
                    0.00%  3.7910us         3  1.2630us  1.0720us  1.6230us  cudaStreamWaitEvent
                    0.00%  2.2540us         3     751ns     135ns  1.9270us  cuDeviceGetCount
                    0.00%  1.0810us         4     270ns     115ns     632ns  cuDeviceGet
[p103062372@hades01 HW4]$
```

(d) MPI nvprof

```
==571== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   55.35%  5.79802s       939  6.1747ms  5.9188ms  6.2753ms  void phase_three<int=32>(int*, int, int, int, int, int, int)
                   18.40%  1.92746s       313  6.1580ms  5.9732ms  6.1743ms  void phase_three_h<int=32>(int*, int, int, int, int, int, int, int, int)
                   18.25%  1.91173s       313  6.1078ms  5.9374ms  6.1493ms  void phase_three_v<int=32>(int*, int, int, int, int, int, int, int, int)
                    3.62%  379.37ms         2  189.68ms  127.21ms  252.15ms  [CUDA memcpy HtoD]
                    3.48%  364.60ms         3  121.53ms  121.43ms  121.61ms  [CUDA memcpy DtoH]
                    0.58%  61.195ms       626  97.756us  78.112us  99.201us  void phase_two<int=32>(int*, int, int, int, int, int)
                    0.15%  15.391ms       313  49.171us  28.864us  50.272us  void phase_two_h<int=32>(int*, int, int, int, int, int, int)
                    0.15%  15.233ms       313  48.669us  48.257us  49.601us  void phase_two_v<int=32>(int*, int, int, int, int, int, int)
                    0.03%  2.8616ms       626  4.5710us  3.6160us  4.8960us  void phase_one<int=32>(int*, int, int, int)
==570== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   61.97%  7.77746s      1252  6.2120ms  5.9715ms  6.4436ms  void phase_three<int=32>(int*, int, int, int, int, int, int)
                   15.80%  1.98266s       313  6.3344ms  6.0324ms  6.4208ms  void phase_three_h<int=32>(int*, int, int, int, int, int, int, int, int)
                   15.55%  1.95214s       313  6.2369ms  6.0388ms  6.4993ms  void phase_three_v<int=32>(int*, int, int, int, int, int, int, int, int)
                    3.98%  499.67ms         3  166.56ms  123.59ms  252.07ms  [CUDA memcpy HtoD]
                    1.94%  242.93ms         2  121.47ms  121.38ms  121.55ms  [CUDA memcpy DtoH]
                    0.49%  61.601ms       626  98.404us  78.304us  104.35us  void phase_two<int=32>(int*, int, int, int, int, int)
                    0.12%  15.617ms       313  49.894us  29.216us  51.680us  void phase_two_h<int=32>(int*, int, int, int, int, int, int)
                    0.12%  15.554ms       313  49.692us  48.992us  51.264us  void phase_two_v<int=32>(int*, int, int, int, int, int, int)
                    0.02%  2.8662ms       626  4.5780us  3.5200us  5.0240us  void phase_one<int=32>(int*, int, int, int)
      API calls:   52.36%  5.77113s         2  2.88557s  2.87312s  2.89801s  cudaMemcpy2D
                   41.56%  4.58071s      3443  1.3304ms  3.6030us  246.29ms  cudaLaunch
                    4.51%  496.62ms         1  496.62ms  496.62ms  496.62ms  cudaHostAlloc
                    1.51%  166.25ms         1  166.25ms  166.25ms  166.25ms  cudaMallocPitch
                    0.02%  2.6146ms     22849     114ns      89ns  220.86us  cudaSetupArgument
                    0.02%  1.7748ms       188  9.4400us     114ns  414.29us  cuDeviceGetAttribute
                    0.01%  1.0542ms         2  527.12us  8.8960us  1.0454ms  cudaFree
                    0.01%  578.16us      3443     167ns     124ns  12.911us  cudaConfigureCall
                    0.00%  278.11us         2  139.06us  113.51us  164.60us  cuDeviceTotalMem
                    0.00%  186.48us         2  93.242us  41.005us  145.48us  cuDeviceGetName
                    0.00%  96.822us         3  32.274us  17.576us  54.594us  cudaMemcpy2DAsync
                    0.00%  16.601us         1  16.601us  16.601us  16.601us  cudaDeviceSynchronize
                    0.00%  4.7840us         1  4.7840us  4.7840us  4.7840us  cudaSetDevice
                    0.00%  2.6330us         1  2.6330us  2.6330us  2.6330us  cudaGetDeviceCount
                    0.00%  2.4950us         3     831ns     169ns  2.1460us  cuDeviceGetCount
                    0.00%     792ns         4     198ns     107ns     369ns  cuDeviceGet
      API calls:   44.94%  4.60799s      3756  1.2268ms  3.2810us  244.42ms  cudaLaunch
                   28.77%  2.94981s         1  2.94981s  2.94981s  2.94981s  cudaMemcpy2D
                   21.34%  2.18835s         1  2.18835s  2.18835s  2.18835s  cudaDeviceSynchronize
                    4.87%  498.83ms         1  498.83ms  498.83ms  498.83ms  cudaHostAlloc
                    0.03%  3.0413ms     25040     121ns      88ns  225.12us  cudaSetupArgument
                    0.02%  1.6361ms       188  8.7020us     110ns  479.69us  cuDeviceGetAttribute
                    0.01%  1.2184ms         1  1.2184ms  1.2184ms  1.2184ms  cudaMallocPitch
                    0.01%  864.61us         2  432.31us  6.1010us  858.51us  cudaFree
                    0.01%  536.06us      3756     142ns     117ns  4.1890us  cudaConfigureCall
                    0.00%  397.11us         2  198.55us  113.26us  283.85us  cuDeviceTotalMem
                    0.00%  180.08us         2  90.041us  51.935us  128.15us  cuDeviceGetName
                    0.00%  131.04us         4  32.759us  7.7040us  54.552us  cudaMemcpy2DAsync
                    0.00%  4.2340us         1  4.2340us  4.2340us  4.2340us  cudaSetDevice
                    0.00%  2.5000us         1  2.5000us  2.5000us  2.5000us  cudaGetDeviceCount
                    0.00%  2.1610us         3     720ns     171ns  1.8070us  cuDeviceGetCount
                    0.00%     948ns         4     237ns     108ns     497ns  cuDeviceGet
[p103062372@hades01 HW4]$
```