Parallel Programming

Homework 4-1 Report: Blocked All-Pairs Shortest Path

108062605 呂宸漢

1. Implementation

Single GPU：hw4-1.cu

a. Blocked Floyd-Warshall 的概念其實就是把原本的 Floyd-Warshall 分割成好幾塊區塊，再分割成 spec 上描述的 3 個 phase 執行，在此先稍微解釋一下每個 phase 的執行方法以方便理解接下來的描述。

phase 1：self-dependent blocks

也就是 pivot block 內自己做 Floyd Warshall，$D_{i,j}^{K}$ 就是比較 vertex i 到 vertex j 的距離與 vertex i 經過 vertex k 到 vertex j 的距離，取最小的距離並更新 $D_{i,j}^{K}$，K 的範圍就是 block 內所有的 vertex id。若將 i 與 j 標準化則會發現其實這個階段就是小規模的 Floyd-Warshall。

phase 2：pivot-row and pivot-column blocks

也就是將 pivot 所在的 row 及 column 的 block 內的 vertex 距離，一樣透過 Floyd-Warshall 計算最短路徑，不過這次是用原路徑與經過 pivot 計算出的路徑取最小的距離當作最短路徑，以 $D_{i,j}^{K}$ 為例，$D_{i,j}^{K}$ 就是比較 vertex i 到 vertex j 的距離與 vertex i 經過 vertex k 到 vertex j 的距離，其中 vertex i 到 vertex j 就是 pivot 所計算的數值，讓 pivot 所在的 row 及 column 的 block 都更新為經過 K 個 vertex 的最短距離。

phase 3：other blocks

用剩餘 block 的位置可以找到該 block 對應的 pivot-row 及 pivot-column，以 $D_{i,j}^{K}$ 為例，$D_{i,j}^{K}$ 就是比較 vertex i 到 vertex j 的距離與 vertex i 經過 vertex k 到 vertex j 的距離，其中 vertex i 到 vertex k 的距離是 pivot-column 所計算的數值，vertex k 到 vertex j 的距離是 pivot-row 所計算的距離。計算完後就完成了這一個 round 的計算。

b. 我是用 GPU block 的 thread 總數當作基本單位分割原本的 distance matrix（Figure 1），如此 GPU 內一個 block 的 thread 就可以一對一 handle 每個 entry，再將 block 組合成 GPU 可以 align 的 2D grid，如此就可以平均分配所有 entry 給所有 thread，雖然會有 thread 因為 padding 的關係被浪費，memory 也會因為 padding 的關係被浪費，不過比起要用判斷式處理而脫慢速度，這樣的方式會比較有效率。

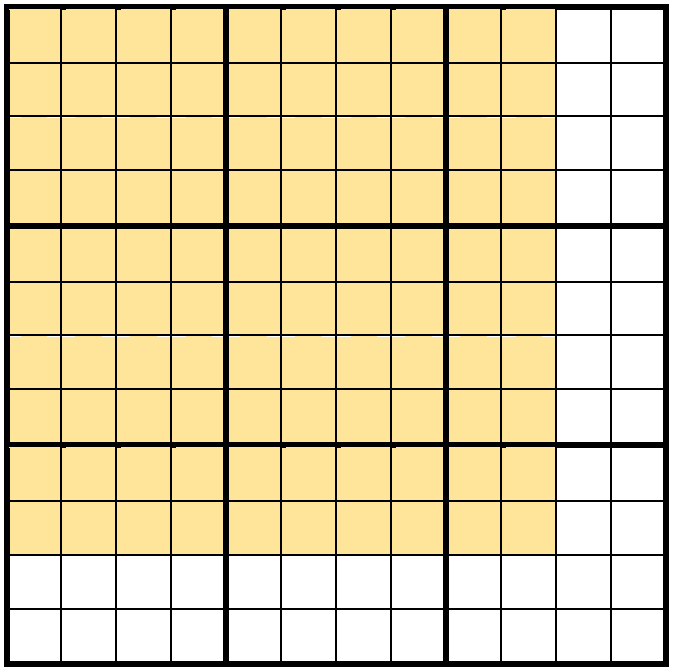


Figure 1. 上圖展示 vertex 數為 10，1 warp = 4 thread，1 block 有 16 個 thread，每個粗線框為一個 block，整個粗線框則是 grid 大小，黃色的部分為 data，白色的部分則是 padding。

c. 一開始我 block factor 就開到最大，因為沒開到最大反而會浪費 warp 內的 thread，降低平行化的程度，以我的實驗結果而言（實驗結果在後面有圖表解釋），的確 block factor 開得較大會比較有效率。不過以上言論是基於我的程式而言，並不確定是不是每個人的程式開滿都會比較快。

2. Profiling Results

由於原本的 case 太大導致測試 Global Load Throughput 時需要花費的時間太長因此我改用測試 correctness 的會後一個 case c21.1 當作 benchmark。

occupancy：

```
==6274== NVPROF is profiling process 6274, command: ./hw4-1 /home/pp19/share/hw4/cases/c21.1 out
==6274== Profiling application: ./hw4-1 /home/pp19/share/hw4/cases/c21.1 out
==6274== Profiling result:
==6274== Metric result:
Invocations                               Metric Name                    Metric Description         Min         Max         Avg
Device "GeForce GTX 1080 (0)"
    Kernel: phase2(int*, int)
        157                   achieved_occupancy                    Achieved Occupancy    0.956302    0.964373    0.960419
    Kernel: phase3(int*, int)
        157                   achieved_occupancy                    Achieved Occupancy    0.977705    0.978261    0.977921
    Kernel: phase1(int*, int)
        157                   achieved_occupancy                    Achieved Occupancy    0.498989    0.499510    0.499375
```

sm efficiency：

```
==8998== NVPROF is profiling process 8998, command: ./hw4-1 /home/pp19/share/hw4/cases/c21.1 out
==8998== Profiling application: ./hw4-1 /home/pp19/share/hw4/cases/c21.1 out
==8998== Profiling result:
==8998== Metric result:
Invocations                               Metric Name                    Metric Description         Min         Max         Avg
Device "GeForce GTX 1080 (0)"
    Kernel: phase2(int*, int)
        157                        sm_efficiency               Multiprocessor Activity      80.05%      94.90%      88.82%
    Kernel: phase3(int*, int)
        157                        sm_efficiency               Multiprocessor Activity      99.87%      99.92%      99.90%
    Kernel: phase1(int*, int)
        157                        sm_efficiency               Multiprocessor Activity       3.89%       4.79%       4.73%
```

shared memory load throughput：

```
==15694== NVPROF is profiling process 15694, command: ./hw4-1 /home/pp19/share/hw4/cases/c21.1 out
==15694== Profiling application: ./hw4-1 /home/pp19/share/hw4/cases/c21.1 out
==15694== Profiling result:
==15694== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "GeForce GTX 1080 (0)"
    Kernel: phase2(int*, int)
        157                    shared_load_throughput             Shared Memory Load Throughput  1879.2GB/s  1945.8GB/s  1918.4GB/s
    Kernel: phase3(int*, int)
        157                    shared_load_throughput             Shared Memory Load Throughput  2044.3GB/s  2048.8GB/s  2046.9GB/s
    Kernel: phase1(int*, int)
        157                    shared_load_throughput             Shared Memory Load Throughput  165.52GB/s  181.91GB/s  180.66GB/s
```

shared memory store throughput：

```
==6305== NVPROF is profiling process 6305, command: ./hw4-1 /home/pp19/share/hw4/cases/c21.1 out
==6305== Profiling application: ./hw4-1 /home/pp19/share/hw4/cases/c21.1 out
==6305== Profiling result:
==6305== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "GeForce GTX 1080 (0)"
    Kernel: phase2(int*, int)
        157                   shared_store_throughput            Shared Memory Store Throughput  60.932GB/s  346.21GB/s  97.400GB/s
    Kernel: phase3(int*, int)
        157                   shared_store_throughput            Shared Memory Store Throughput  66.502GB/s  297.98GB/s  104.46GB/s
    Kernel: phase1(int*, int)
        157                   shared_store_throughput            Shared Memory Store Throughput  2.4070GB/s  12.049GB/s  3.1358GB/s
```

global memory load throughput：

```
==15731== NVPROF is profiling process 15731, command: ./hw4-1 /home/pp19/share/hw4/cases/c21.1 out
==15731== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
==15731== Profiling application: ./hw4-1 /home/pp19/share/hw4/cases/c21.1 out
==15731== Profiling result:
==15731== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "GeForce GTX 1080 (0)"
    Kernel: phase2(int*, int)
        157                            gld_throughput                  Global Load Throughput  59.142GB/s  60.401GB/s  59.825GB/s
    Kernel: phase3(int*, int)
        157                            gld_throughput                  Global Load Throughput  63.834GB/s  63.956GB/s  63.897GB/s
    Kernel: phase1(int*, int)
        157                            gld_throughput                  Global Load Throughput  85.364MB/s  89.451MB/s  87.738MB/s
```

global memory store throughput：

```
==9252== NVPROF is profiling process 9252, command: ./hw4-1 /home/pp19/share/hw4/cases/c21.1 out
==9252== Profiling application: ./hw4-1 /home/pp19/share/hw4/cases/c21.1 out
==9252== Profiling result:
==9252== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "GeForce GTX 1080 (0)"
    Kernel: phase2(int*, int)
        157                            gst_throughput                  Global Store Throughput  19.241GB/s  23.495GB/s  22.378GB/s
    Kernel: phase3(int*, int)
        157                            gst_throughput                  Global Store Throughput  21.232GB/s  24.867GB/s  24.422GB/s
    Kernel: phase1(int*, int)
        157                            gst_throughput                  Global Store Throughput  80.414MB/s  103.63MB/s  101.25MB/s
```

occupancy 及 sm efficiency 可以看出不同 phase 的 GPU 占用及使用率，phase 1 因為只計算一個 block，所以占用率及使用率相當的低，phase 2 因為是計算 pivot-row 及 pivot-column 的 block，在所有 block 中會包含 pivot block，因此使用率大概 80 - 95%，而 phase 3 則是要計算所以剩餘的 block，grid 中的所有 block 幾乎都要計算，因此利用率及占用率來到 99%，不過由此還無法斷定 bottle-neck 的所在，雖然 phase 3 的利用率很高，phase 1 及 phase 2 較低導致整體利用率不平均，可是不同 phase 間因為要獨立計算，因此無法 overlap 彼此的計算時間。

由 global memory load/store throughput 及 shared memory load/store throughput 的數據可以看出，shared memory 的 throughput 皆比 global memory 的 throughput 還要多，至少可以推論 bottle-neck 不可能發生在 shared memory，有可能是 global memory 的原因。

3. Experiment & Analysis

a. System Spec

所有程式皆在課程所提供的 cluster 上進行測試。

GPU imformation：

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 384.81                 Driver Version: 384.81                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 1080    On   | 00000000:4B:00.0 Off |                  N/A |
| 32%   47C    P8    14W / 216W |      2MiB /  8114MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  GeForce GTX 1080    On   | 00000000:4D:00.0 Off |                  N/A |
| 10%   47C    P8    18W / 216W |     11MiB /  8113MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
```

b. Time Distribution

i. computing time

用 nvprof 取得 3 個 phase 的執行時間。

ii. communication time

因為是 single GPU，因此傳輸時間為 0。

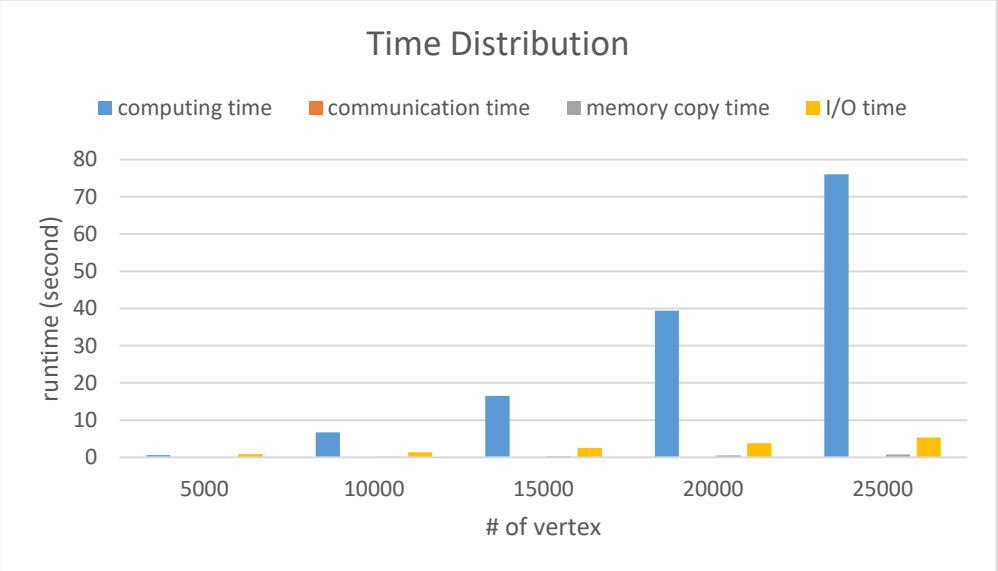
iii. memory copy(H2D, D2H)

用 nvprof 取得 H2D 與 D2H 的執行時間。

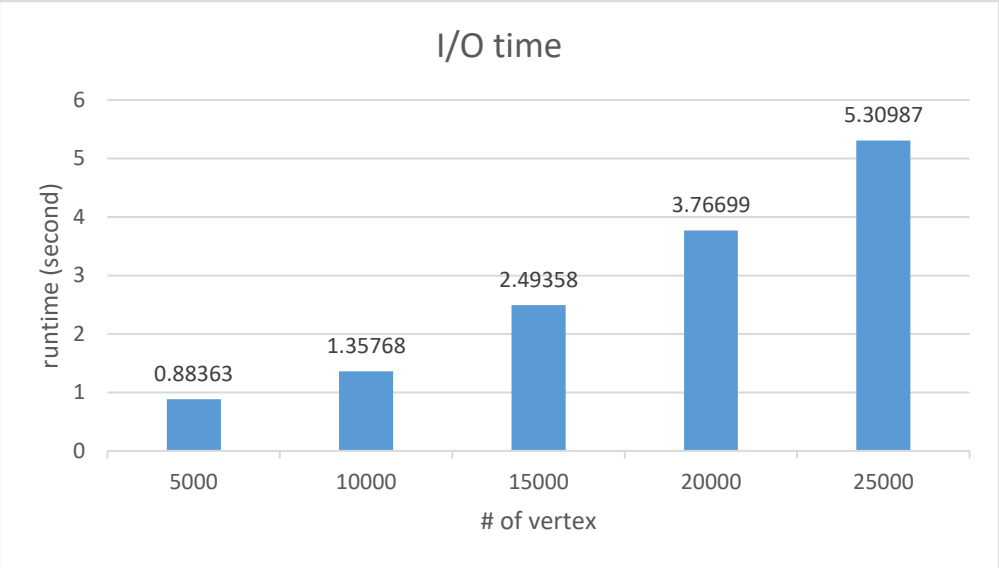
iv. I/O time

用 clock_gettime(…)取 CLOCK_MONOTONIC 的時間，將此函式加在檔案 input/output 的前後取差值相加即可。

由下圖可以看出，雖然 I/O time 的時間仍然很長，computation time 才是整個程式執行時間長短的決定因素，以先前的分析及測試數據而言，也就是 phase 3 最耗費時間，若要降低 runtime 則必須想辦法加快 phase 3 的速度才是最重要的。



由下圖顯示，因為我沒有平行化讀寫檔的程式，導致讀取時間會隨著資料數越大而越長，因此若時限一直是 30 秒的情況下，這不是一個好的 I/O 策略，應該要平行化加速以防 I/O time 佔執行時間越多。



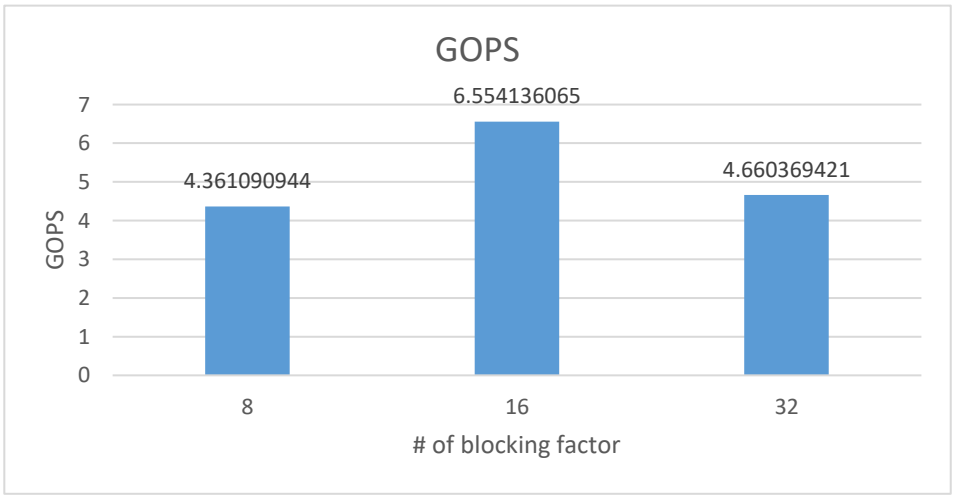
c. Blocking Factor

此測試是用 correctness case c21.1 做 benchmark，原本是使用自己產的 testcases，可是在取得 GOPS 時太花時間，跑了兩個小時多還沒有答案出來，擔心占用 GPU 太久導致損失其他同學的權益，因此使用此 case。

i. GOPS

在執行時加上 nvprof --metrics inst_integer 取得。

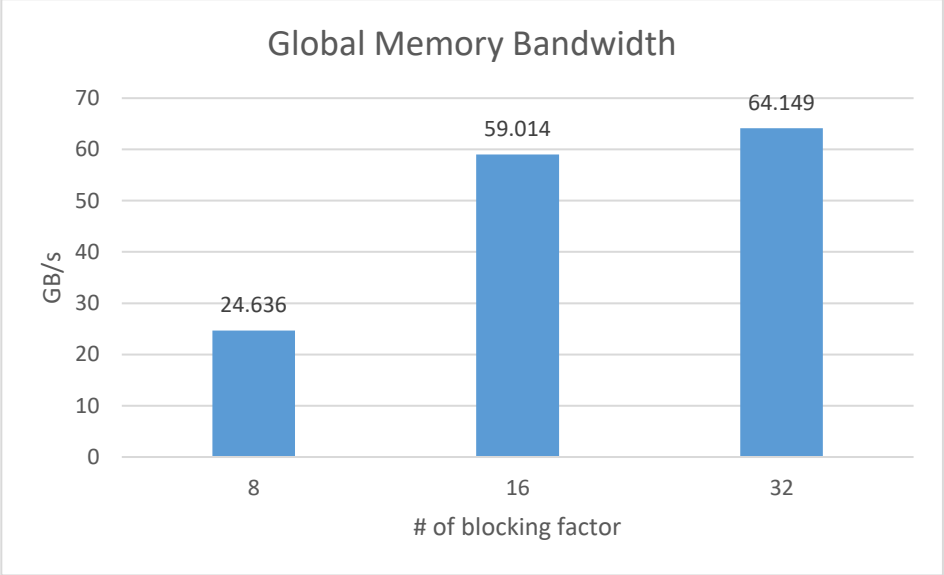
由下圖顯示 GOPS 沒有隨著 block factor 的增加而增加，一開始我認為是實驗做錯，可是後來想想，因為他是計算每個 thread 所計算的 integer 總數的加總在除以時間平均，我人為是因為 padding 的關係，可能是因為 factor 為 8 與 32 的時候，較接近 matrix 的 size，因此需要 padding 的數量較少，才導致 factor 為 8 與 32 的 GOPS 較相近，反而 factor 為 16 的 GOPS 較大。



ii. global memory bandwidth

在執行時加上 nvprof --metrics gld_throughput 取得。

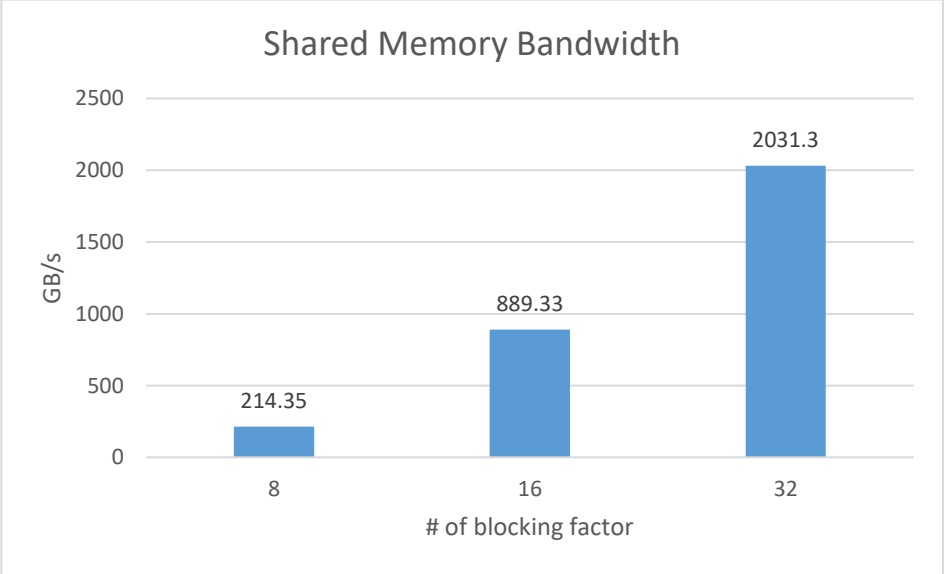
因為 global memory 是在每個 phase 的前後 thread 各自搬動資料到 shared memory 時使用的，因此當 thread 變多，一次需要搬動的資料就變多，因此當 block factor 變大後，需要的 memory bandwidth 就會變大，不過如果 thread 數使數成長的話 bandwidth 的使用應該也是指數成長才對，我認為沒有的原因是因為有些 bank conflict 我沒有解決到，而且 bandwidth 也有極限，因此才停在 64。



iii. shared memory bandwidth

在執行時加上 nvprof --metrics shared_load_throughput 取得。

Shared memory 的 bandwidth 測試結果則在我的預料之內，因為 block factor 變大，thread 數也變多，每個 thread 一次計算的量雖然一樣，可是 thread 數多，因此 shared memory bandwidth 也提高。



d. Optimization

以下列出我有使用的加速方式

i. Share memory

在每次計算前，thread 會把資料從 global memory 搬到 share memory，計算完再搬回去，加速每次 thread 計算時取資料的時間，這也是加速最多的一個方式。

ii. Large blocking factor

如同前面測試所說，我將 block 中的 thread 數開到最大，原本的 block factor 我只用 16 而已，因為當時不確定 share memory 的 size 有沒有辦法 handle 每個 thread 負責一個 entry，而後就開到 32 了。

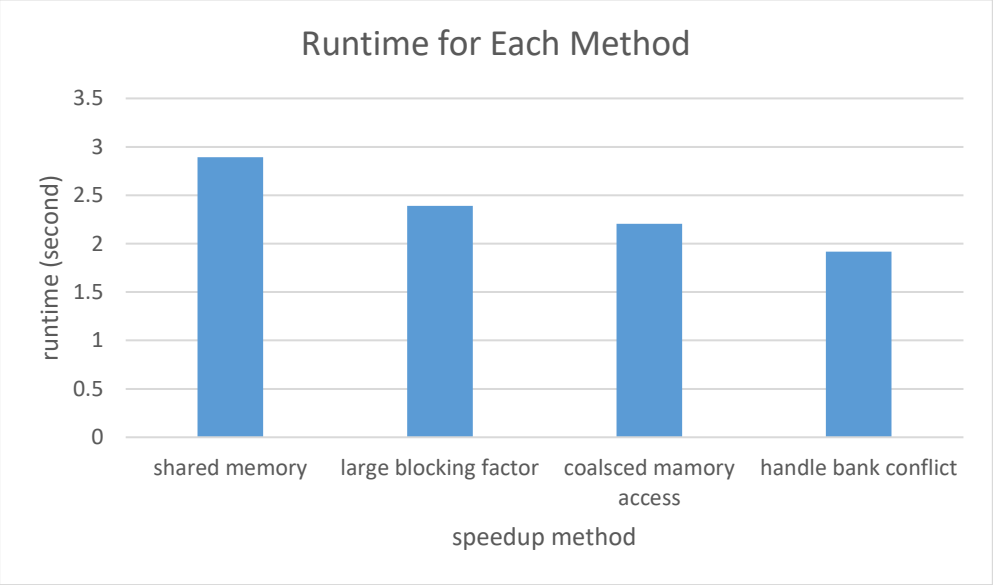
iii. Coalesced memory access

依照 memory 的性質存取 memory。

iv. Handle bank conflict

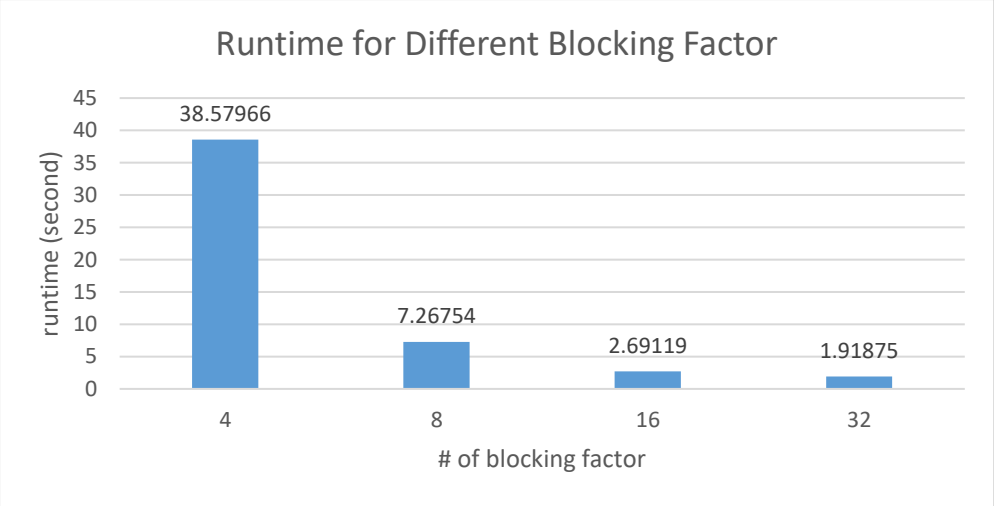
將一個 warp 存取的 memory bank 錯開，不過我只成功改了 phase 1，在改 phase 2 及 phase 3 時速度一直掉下來，而且在 judge 時偶爾會出現 runtime error，我認為是有些小地方我沒有注意到才導致 runtime error，不過只改進 phase 1 也有加速，因此我將此加進來。

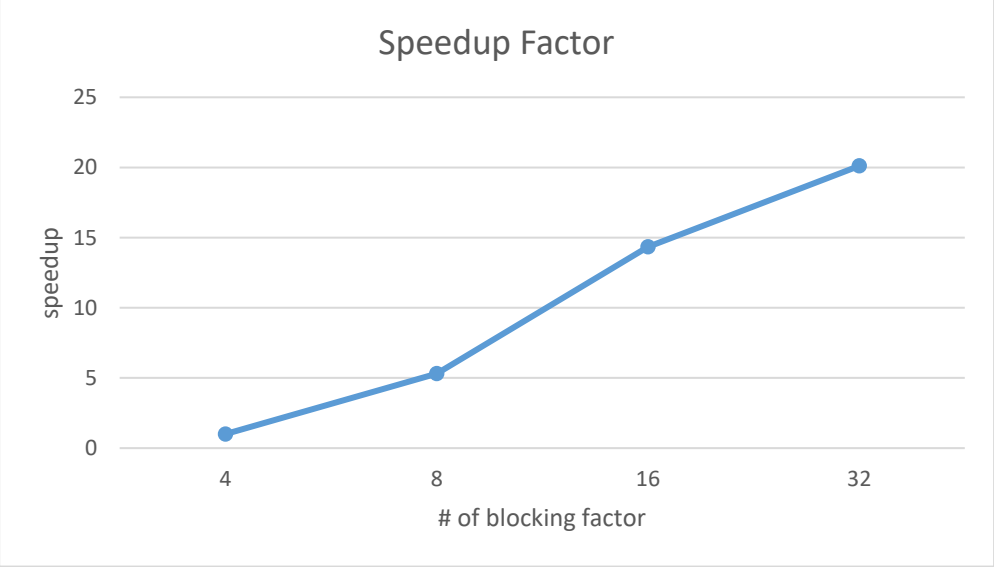
下圖顯示不同加速 method 的執行時間，表格由左到右是有 overlap 的，意即第一個 column 是 shared memory，第二個 column 是 shared memory + large blocking factor，依此類推。



e. Other

這是我額外做的圖表，用以說明不同 block factor 對執行時間的影響及 speedup，由下圖的資料可以看出，此程式的 speedup 與理想的狀況大致相同，都是指數成長，加速狀況良好。





4. Experiences / Conclusion

這次作業的優化很麻煩，一開始先把 sequential code 改成 GPU 可以跑的 code，剛開始還沒有用 shared memory 時 correctness 大部分都會過，雖然有很多都壓在時限前面一些而已，之後改成用 shared memory 時，時間大大的降低，不過在將低後就開使不好優化了，很多時候有找到可以優化的地方，可是當我改上去後，執行時間大都沒有變短，有時候還變長了，就開始猜想是自己優化有錯，還是有些 memory 的 affinity 在一開始寫的時候有有顧慮到，所以改了反而會變慢，在優化的過程中一直遭遇瓶頸，可是又怕優化到最後一刻時來不及寫報告與實驗，因此我選擇優化到一定程度後，就開始寫報告與實驗，以防 GPU 到時候太滿不好做實驗。希望下次的 multi-GPU 版本我可以再試出更好的優化。