2017 PP Hw4

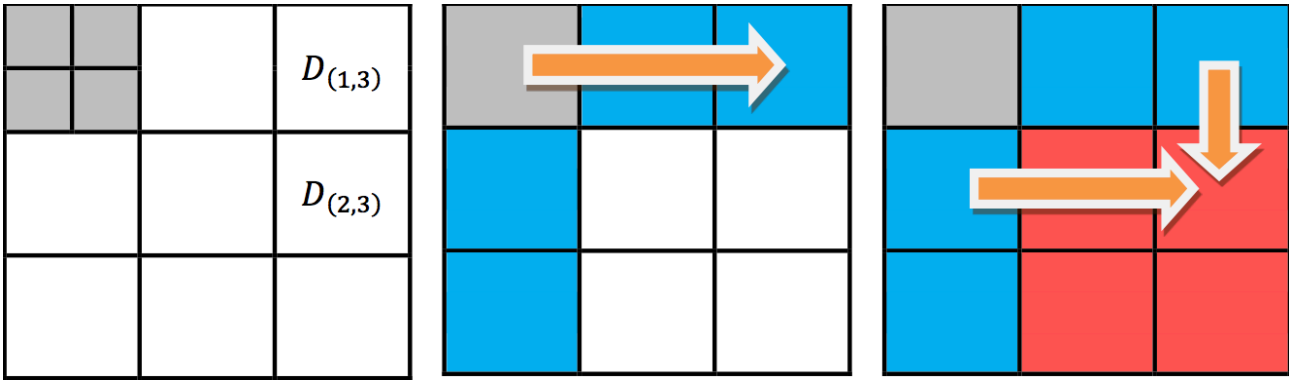
105062570 林子瑜

Implementation

1. Cuda（B = 16）

Shared-memory

根據 Blocked Floyd-Warshall algorithm，每個 phase 的 dependency 關係如下圖所示：

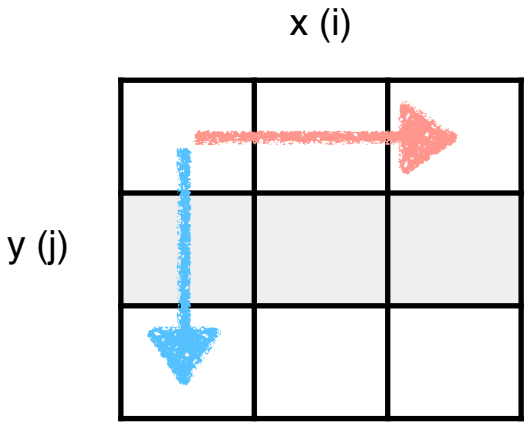


可以得知每個 block 所需的 dependent block 其實最多只會有三個，因此在 shared memory 的配置上只要將相關的這三個 block 讀取進來即可。

那讀進 shared memory 之後需要將 index re-define，這邊 re-define 的方式就是將原本的 index 減掉原本所屬的 (b_i, b_j) * B 後再加上 n*B*B，這邊的 n 決定於當初放進 shared-memory 時的順序。

Sequential address

在將 global memory 讀進 shared-memory 要特別注意，由於GPU thread 開 2 維時 index 的標籤方式與 host 端 graph 的標籤方式不太一樣，如下圖所示：



因此在讀取 global memory 時若是單純使用 i*V+j 的話，就會變成是藍色的箭頭一般跳著讀取 global memory 的值，使得效能低下。當將 index 方式改成 j*V+i 後會變成是粉色箭頭，sequential 讀取 global memory 的值，效能也會突飛猛進。

Memcpy

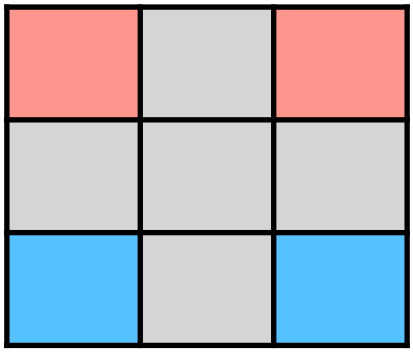
我使用了 cudaMallocPitch / cudaMemcpy2D 來做 device 端記憶體的配置，那在 single 版本中由於 memcpy 並非主要 bottleneck，因此在效能上看不太出差別。

Stream

那由於在 phase2&3 時，會因為有 4 個方向導致有 4 個 kernel call，而每個 kernel 是 sequential 執行的，因此想說若是利用 stream 讓 4 個 kernel call 平行，那或許就能夠增加效能。但是事實證明這樣的想法並不成功，猜想原因應該是 GPU 的資源被吃滿了，就算增加平行度，沒有多餘的運算資源也沒用，後來想想也合理，因為若是單一 GPU 能夠 handle 4 個 kernel 的平行的話，就不需要 multi GPU 惹。

2. OpenMP（B = 16）

openMP 做法與 single version 差不多，差別在於只有在第三個 phase 時利用了兩張卡來加速。切割方式是切割成上下兩半，如下所示：



利用 phase 2 切出來的十字為界，將上下兩半（兩個顏色）切出來，接著將上下兩半在各自對半分，分別給兩張 GPU 卡做運算。而做完運算後的溝通這邊是利用 copy 回 host 再 copy 回自己所屬的 device_ptr 來做到。

Memcpy

那由於數量過多的 memcpy 會成為 bottleneck，因此使用 cudaMallocPitch / cudaMemcpy2D 會使效能上升不少。

Final version

那由於上述的版本實在太慢了，或許是優化不夠吧，因此最後的版本是會看看上下兩塊的大小是否夠大，夠大的話才會進行兩塊 GPU 的同時算，若是不夠大就單張卡各算各的。

3. MPI（B = 32）

MPI 做法與 openMP 的版本一樣，只是在溝通的地方換成 MPI_Isend / MPI_recv。

Profile (# vertex = 10000)

1. Cuba

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   49.21%  3.15164s       624  5.0507ms  22.816us  11.390ms  gpu_phase3_down(int*, int, int, int, unsigned long)
                   48.33%  3.09555s       624  4.9608ms  22.625us  9.9037ms  gpu_phase3_up(int*, int, int, int, unsigned long)
                    0.99%  63.198ms         1  63.198ms  63.198ms  63.198ms  [CUDA memcpy HtoD]
                    0.96%  61.730ms         1  61.730ms  61.730ms  61.730ms  [CUDA memcpy DtoH]
                    0.28%  17.768ms      1248  14.237us  2.7840us  30.112us  gpu_phase2_ud(int*, int, int, int, int, int, unsigned long)
                    0.19%  12.388ms       625  19.820us  18.016us  21.856us  gpu_phase2_lr(int*, int, int, int, unsigned long)
                    0.04%  2.2470ms       625  3.5950us  3.3920us  10.080us  gpu_phase1(int*, int, int, int, unsigned long)
      API calls:   69.66%  4.56960s      3748  1.2192ms     374ns  11.370ms  cudaLaunch
                   27.92%  1.83167s         2  915.84ms  63.130ms  1.76854s  cudaMemcpy2D
                    2.36%  154.55ms         1  154.55ms  154.55ms  154.55ms  cudaMallocPitch
                    0.04%  2.4831ms     21236     116ns      92ns  14.251us  cudaSetupArgument
                    0.01%  691.89us        94  7.3600us     275ns  306.36us  cuDeviceGetAttribute
                    0.01%  510.69us      3748     136ns     102ns  12.485us  cudaConfigureCall
                    0.00%  308.25us         1  308.25us  308.25us  308.25us  cuDeviceTotalMem
                    0.00%  306.78us         1  306.78us  306.78us  306.78us  cudaFree
                    0.00%  75.896us         1  75.896us  75.896us  75.896us  cuDeviceGetName
                    0.00%  5.9020us         3  1.9670us     557ns  4.7640us  cuDeviceGetCount
                    0.00%  1.6760us         2     838ns     372ns  1.3040us  cuDeviceGet
```

可以看到最大的 overhead 是在 phase 3

2. openMP

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   96.92%  12.5781s      2496  5.0393ms  18.048us  11.333ms  gpu_phase3(int*, int, int, int, int, unsigned long)
                    1.46%  190.05ms         4  47.513ms  30.982ms  64.220ms  [CUDA memcpy HtoD]
                    0.95%  122.70ms         4  30.674ms  30.395ms  31.094ms  [CUDA memcpy DtoH]
                    0.64%  83.075ms      4992  16.641us  3.1360us  31.936us  gpu_phase2(int*, int, int, int, int, int, int, unsigned long)
                    0.03%  4.3004ms      1250  3.4400us  3.1360us  10.785us  gpu_phase1(int*, int, int, int, int, unsigned long)
      API calls:   95.47%  12.8110s      2506  5.1121ms  6.0850us  36.773ms  cudaDeviceSynchronize
                    1.80%  241.37ms         1  241.37ms  241.37ms  241.37ms  cudaHostAlloc
                    0.97%  129.86ms         8  16.233ms  8.2450us  129.71ms  cudaStreamCreate
                    0.95%  127.28ms         2  63.640ms  62.959ms  64.321ms  cudaMemcpy2D
                    0.46%  61.283ms         1  61.283ms  61.283ms  61.283ms  cudaFreeHost
                    0.26%  35.228ms      8738  4.0310us  3.3270us  260.37us  cudaLaunch
                    0.05%  7.3535ms     63662     115ns      90ns  251.97us  cudaSetupArgument
                    0.01%  1.6192ms      8738     185ns     127ns  254.28us  cudaConfigureCall
                    0.01%  1.1183ms         2  559.17us  524.20us  594.15us  cudaMallocPitch
                    0.01%  1.0917ms         2  545.86us  415.37us  676.35us  cudaFree
                    0.01%  1.0658ms       188  5.6680us     256ns  249.44us  cuDeviceGetAttribute
                    0.00%  449.16us         2  224.58us  220.32us  228.84us  cuDeviceTotalMem
                    0.00%  115.51us         2  57.754us  51.736us  63.772us  cuDeviceGetName
                    0.00%  82.489us         6  13.748us  5.8490us  27.749us  cudaMemcpy2DAsync
                    0.00%  8.0030us         3  2.6670us     615ns  3.7370us  cudaSetDevice
                    0.00%  4.7950us         3  1.5980us     352ns  3.5830us  cuDeviceGetCount
                    0.00%  2.2430us         4     560ns     311ns  1.2620us  cuDeviceGet
                    0.00%  1.1840us         1  1.1840us  1.1840us  1.1840us  cudaGetDeviceCount
```

很明顯的，memcpy 的 overhead 大幅提升，因為兩個 GPU 需要溝通的緣故

3. MPI

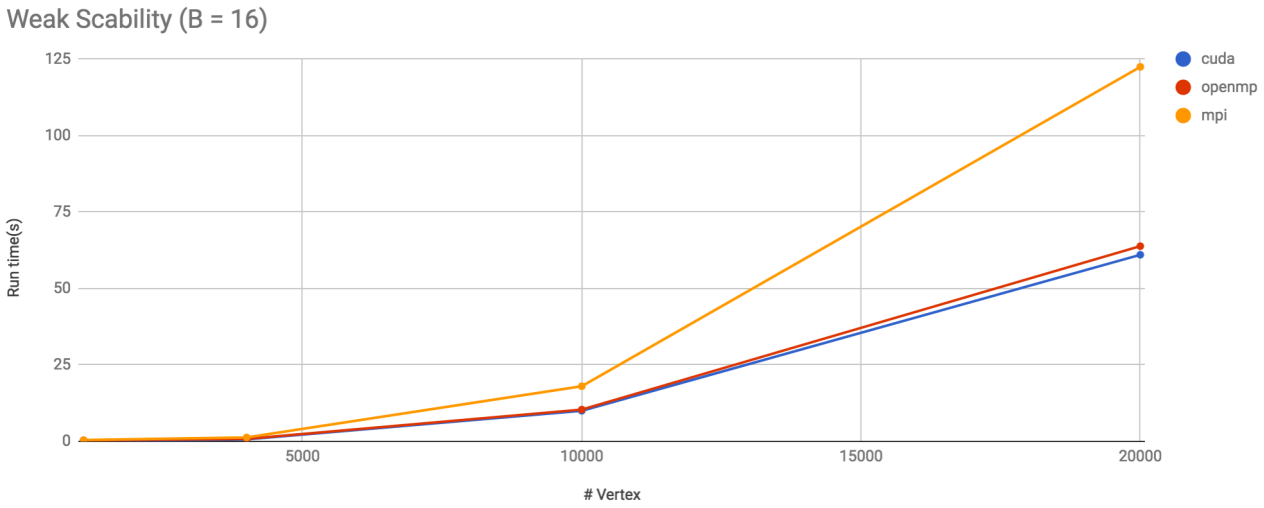
```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   97.17%  10.7485s      1248  8.6126ms  18.272us  17.277ms  gpu_phase3(int*, int, int, int, int, unsigned long)
                    1.50%  166.45ms         2  83.224ms  57.397ms  109.05ms  [CUDA memcpy HtoD]
                    0.91%  101.07ms         2  50.533ms  47.067ms  53.999ms  [CUDA memcpy DtoH]
                    0.39%  43.496ms      2496  17.426us  3.7120us  29.248us  gpu_phase2(int*, int, int, int, int, int, int, unsigned long)
                    0.02%  2.4062ms       625  3.8490us  3.6160us  4.0640us  gpu_phase1(int*, int, int, int, int, int, unsigned long)
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   97.22%  10.9000s      1248  8.7339ms  18.304us  17.474ms  gpu_phase3(int*, int, int, int, int, unsigned long)
                    1.49%  167.25ms         2  83.624ms  56.550ms  110.70ms  [CUDA memcpy HtoD]
                    0.88%  98.586ms         2  49.293ms  44.828ms  53.758ms  [CUDA memcpy DtoH]
                    0.39%  43.449ms      2496  17.407us  4.0970us  29.344us  gpu_phase2(int*, int, int, int, int, int, int, unsigned long)
                    0.02%  2.4217ms       625  3.8740us  3.6480us  4.1280us  gpu_phase1(int*, int, int, int, int, int, unsigned long)
```

Experiment (all on GeForce 1080)

1. System spec

課程提供的平台，使用 GeForce 1080

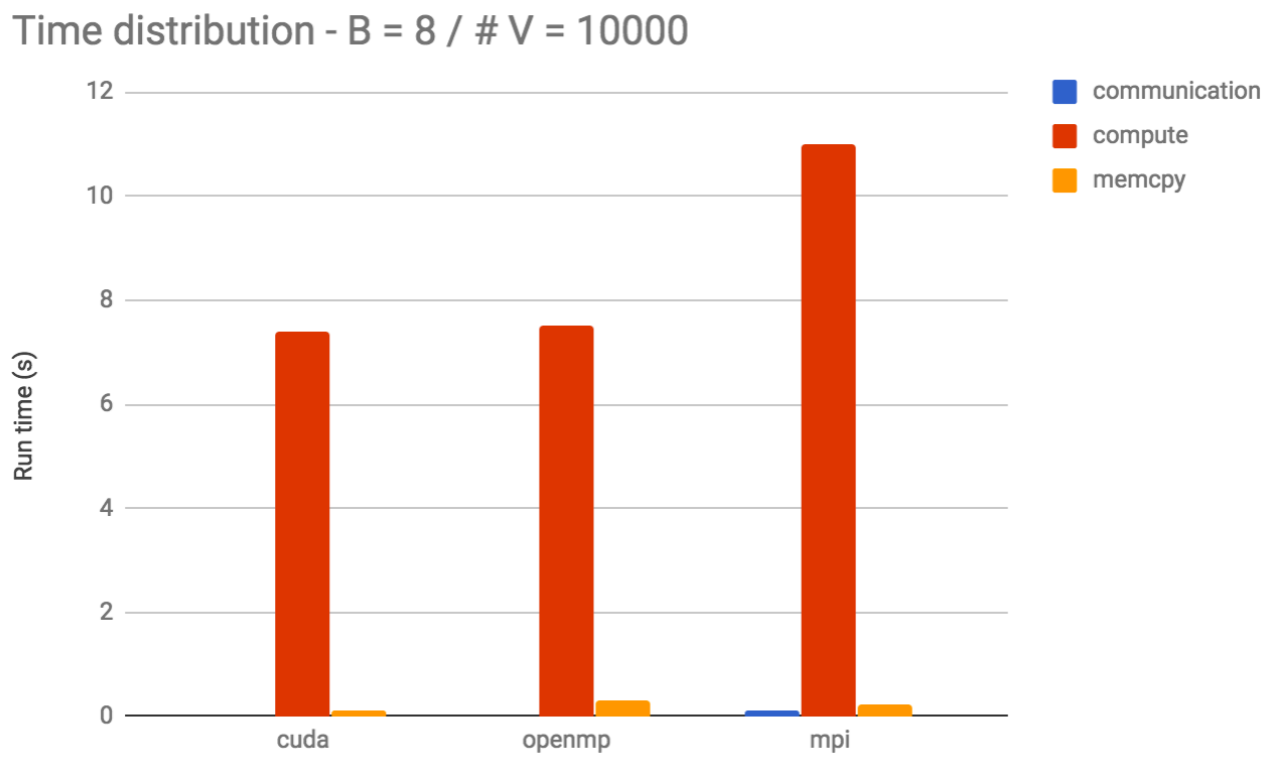
2. Weak scalability

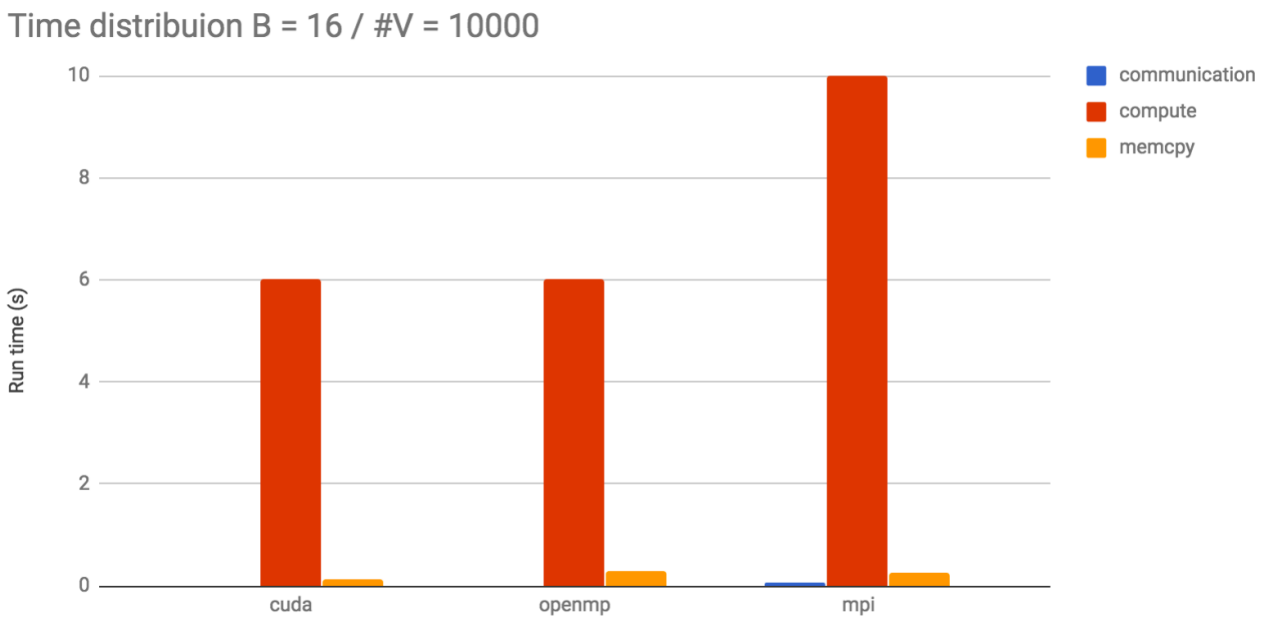


可以看到 openmp 和 cuda 差不多，差別在於某些 round 的 phase3 會分工並且以 memcpy 做溝通，那很明顯的，我的程式並沒有讓兩張 GPU 的優勢顯現出來，反而還因為 mamcpy 的關係造成某些 overhead。

那 mpi 的部分就更誇張了，溝通的成本超級大，我想這應該是程式內有某些邏輯上的 bug 導致的。

3. Time distribution





時間的計算：

compute / memcpy 都是利用 nvprof 的資料，而溝通的部分是利用 gettimeofday 來做計算。

有幾點觀察：

I. 在 B = 8 時 compute time 皆比 B = 16 來得高，這點與 Blocking Factor 的實驗結論相吻合，推測就是平行效果不夠好。

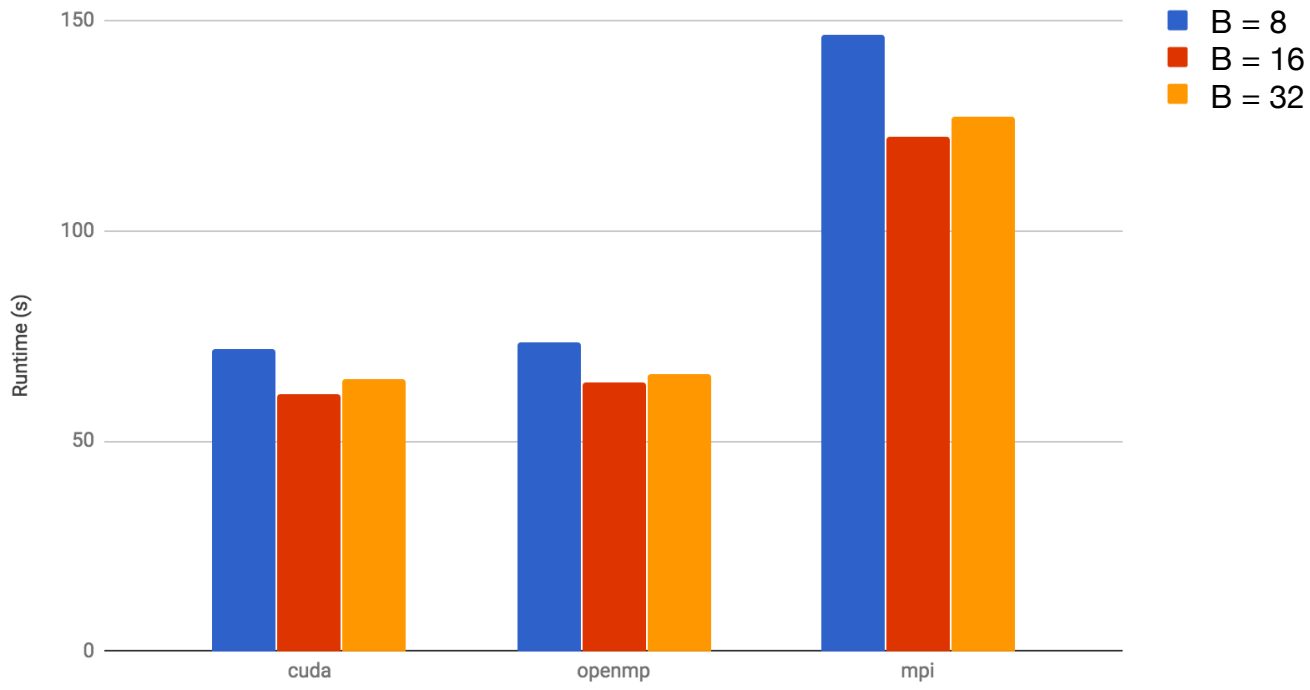
II. 這邊推翻上述所言，mpi 的溝通成本其實只多了一點點，主要是 compute 時間比其他兩者高，這點是因為為了要能夠比較好的傳送資料給別人，因此在 index 上有做一點修改，使得能夠以 row 為單位作傳送，猜測為這點的差別。

III. openmp 以及 mpi 的 memcpy 時間都相對較高，因為需要把 device 的資料 copy 回 host 才能做溝通。

4. Blocking Factor

可以看出在 block factor = 16 時會有最佳效能。那探究其原因的話，在 B = 8 時會使得 round 數過多導致 call kernel 的次數增加，這或許是一個 overhead，因為並沒有達到好的平行效果。

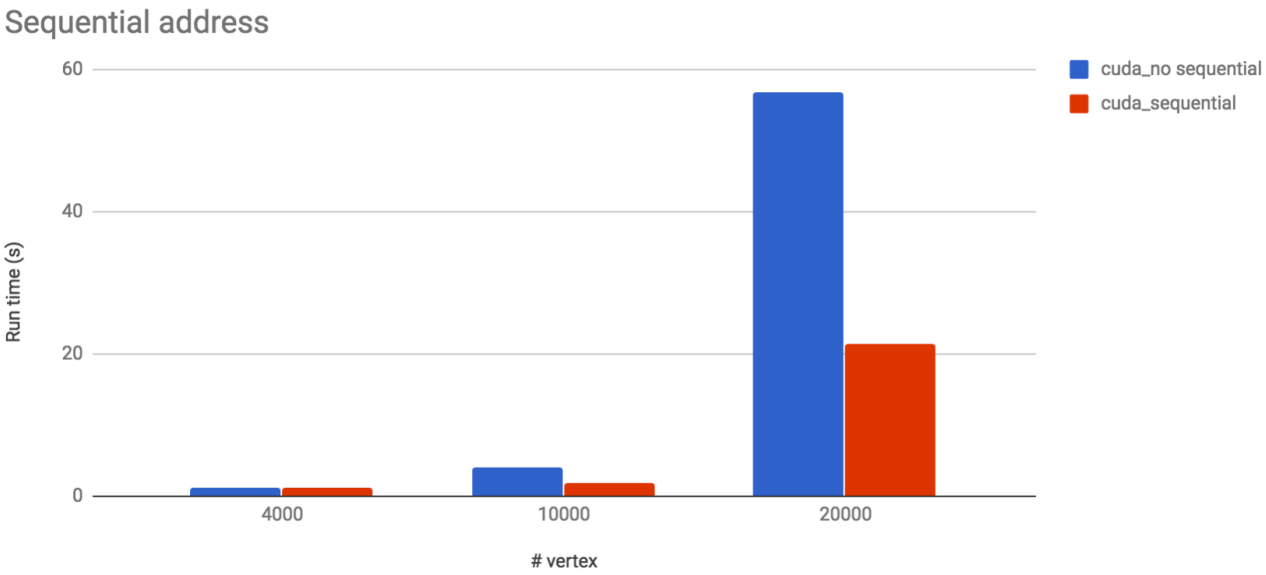
而 B = 32 的時候也會些微變差，這邊猜測是因為當單一 block 裡的 thread 數增加時，因為 kernel 裡有兩層 for 迴圈，而造成更多 branch 導致效能些微下降。



5. Optimization

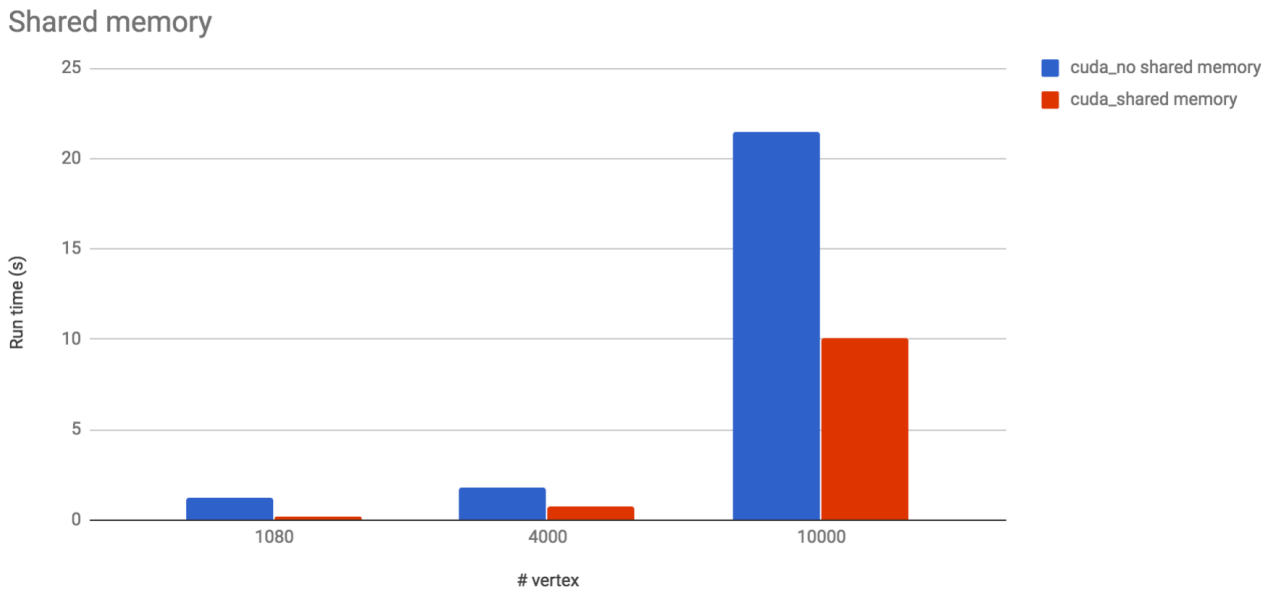
這是很底層的優化，所以三個版本都有做，因此以下只 demo cuda 版

1. Sequential address



看得出來在 Global memory 端是否有 sequential 讀取資料會使效能差距非常大

2. Shared memory



Shared memory的部分就不用說了，當然會提升效能

Conclusion

這次作業要我們 implement blocked Floyd-warshall algorithm on GPU，本身的演算法並不難實作，難的在於要如何加速，而要做到好的加速必須要對 GPU 的整個硬體構造（memory 等）相當了解才行，因此這次作業我認為算是最有難度的一次。

那 openMP 以及 MPI 其實算是屍體，畢竟沒有比單一 GPU 來得快，要如何做到溝通方面的加速也是我至今無法太理解的部分。