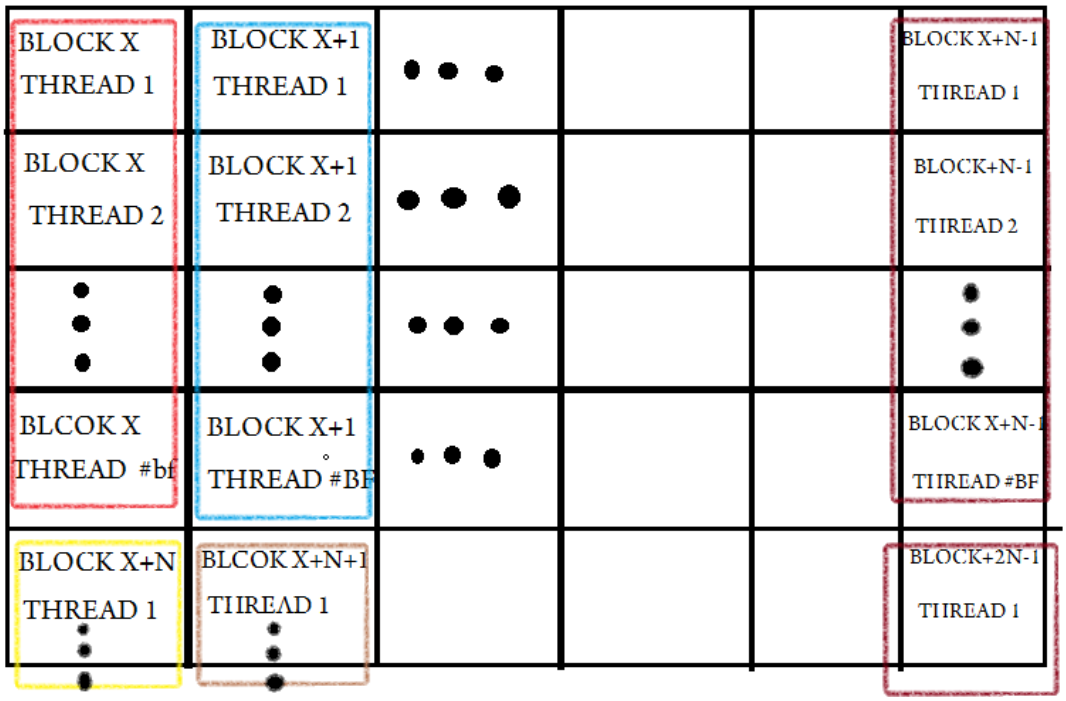
Parallel Programming HW4 Report

102062209 邱政凱

Implementation

實作 Cuda 的第一個要件就在於如何將 Data 之間區分成彼此不互相 Dependent 的單位，充分利用 GPU 的效能。在這次作業中，我們要計算的是一個 N*N 的 Shortest path graph，因此我基本的概念就是讓每個 GPU 裡的 thread 負責更新一個對應的(i,j)距離。然而 cuda 最高的 thread 上限數是 1024，因此我們勢必要切割出許多的 BLOCK。最直覺的做法應該是讓每一個 Blocked APSP algorithm 裡面的 block 對應到一個 cuda 的 block，然而在 Blocked APSP 裡面，同一個 tile block 在同個 phase 裏會需要使用到相依或相同的資料，因此這種作法只支援 Block factor <=32 的實作方法(32*32 = 1024,單一 block thread 數上限)。因此我使用了另一種方法：(前面那種方法我仍然有實作，並在後面的 optimize section 會分析實驗結果)



我的作法是將每個 block 都設定跟 BF 一樣的 Thread 數，接著如上圖一樣，block id 隨著 column id 增加而增加，thread id 隨著 row id 增加而增加，而每 BF 個 ROW 之後又是另一排新的 Block。這種作法在大測資的狀況下會需要多維度的 block dim3，因為 cuda 的 kernel block 數單一維度上限約莫六萬，不過就算放入多維度的 block dim3，我仍然會用轉址的方式把其視為單一維度來處理。

而在 CPU 的部分，我將三個 Phase 分別分成三個 Kernel 來 Launch，原因就在於三個 Phase 需要的 block 數不同，轉址的方式也不同，data dependency 也不同。

```
    for(int K=0;K<FWblockDim;K++){

        for(int i=0;i<bf;i++){
            floyd_warshall_beta_1<<<bf,bf>>>(cuda_graph,K,K*bf + i);
            cudaCheckErrors("phase 1");
        }

        if(FWblockDim>1){
            for(int i=0;i<bf;i++){
                floyd_warshall_beta_2<<<(FWblockDim-1)*2*bf,bf>>>(cuda_graph,K,K*bf + i );
                cudaCheckErrors("phase 2 col");
            }


            floyd_warshall_beta_3<<<blockStr2,bf>>>(cuda_graph,K,K*bf,FWblockDim-1);
            cudaCheckErrors("phase 3");

        }
    }
}
```

可以看到，我在 Phase 1 或 Phase 2 裏，會分成 BF 數個階段來 Launch kernel，這部分是因為在 Phase1 和 2 中，每個 iteration 都會 depend on 上一 Iteration 自身 block 的計算結果，而因為跨 block 的 thread 間無法 synchronize，因此藉由 launch 新的 kernel 確保所有的 thread 抓到的資料都是最新的。

而 Phase 3 因為每次 iteration 都不會相依於上一輪自身計算的結果，而只相依於 phase 2 中的結果，因此可以直接用一個 kernel launch，在裏面直接進行 K 個 iteration，而不用擔心 Synchronization 的問題。

至於關鍵的轉址方式簡單附上如下，大致上就是確保所有 block 的所有 thread 不會計算到相同的(I,j)。

```
__global__ void floyd_warshall_beta_1(int* dist, int k , int kbf  )

    int idx,idy;
    idx = k;
    idy = k;
    int i = cuda_bf * idx + (blockIdx.x%cuda_bf);
    int j = cuda_bf * idy + threadIdx.x;
```

```
__global__ void floyd_warshall_beta_2(int* dist, int k , int kbf  ){

    int idx,idy;
    int temp = blockIdx.x / cuda_bf;
    if( (temp) % 2 == 0 ){
        idx = (temp/2) >= k ? (temp/2+1):(temp/2);
        idy = k;
    }
    else {
        idx = k;
        idy = (temp/2) >= k ? (temp/2+1):(temp/2);
    }

    int i = cuda_bf * idx + (blockIdx.x%cuda_bf);
    int j = cuda_bf * idy + threadIdx.x;
```

```
__global__ void floyd_warshall_beta_3(int* dist, int k , int kbf ,int grid_size ){

    int idx,idy;
    int temp = ((blockIdx.y*gridDim.x) + blockIdx.x) / cuda_bf;
    idx = temp/grid_size  >= k?  temp/grid_size + 1 : temp/grid_size;
    idy = temp % grid_size >= k? temp%grid_size + 1 : temp % grid_size;

    int i = cuda_bf * idx + (blockIdx.x%cuda_bf);
    int j = cuda_bf * idy + threadIdx.x;
```

在 Multi GPU 的版本中，我實作的是可以針對任意數量的 GPU 做計算分配的演算法。因為考量到 Communication 和 Memcpy 的 overhead，我只針對佔據大部分計算量的 Phase3 做跨 GPU 的計算。也就是說，所有 GPU 都在每個回合中自行計算完整的 Phase1 和 Phase2，然後將 Phase 3 的工作量均分。

以兩顆 GPU 的狀況為例，如上圖的 graph 所示，如果目前是 K=1，也就是整個 Blocked APSP algorithm 的第二個 iteration，那上圖紅色和藍色的部分就是 Phase 1/2 要計算的 block，而這是所有 GPU 都要計算的部分。接著在 Phase 3，我會用 row index 當作參考，將扣除 Phase1/2 的 block 後剩下的 blocks 中所有的 row 分成(#GPU)個 Section，如果有無法整除的餘數 Index 的 row 都丟給 ID 最大的 GPU 計算。

計算個別完成之後，在 openmp 的版本中我使用 cudaMemcpy(…,Device to Device)的設定，直接交換所有 GPU 裡面的資料。以上圖為例，GPU1 就是把自己的黃色部分的 BLOCK 的距離計算結果傳給 GPU2，而 GPU2 則把自己的綠色部分傳給 GPU1。交換完彼此的資料之後，設定 K=K+1，繼續 Blocked APSP 的下個 Iteration。在 MPI 的版本中則是每個 Process 先用 cudaMemcpy(…,Device to Host)把自己計算的部分 copy 回 CPU，接著用 MPI_Send 傳給其他所有的 GPU，並同

時接收所有其他 GPU 送來的計算結果，每個 Process 再個別用 cudaMemcpy(…,Host to Device)把其他 GPU 的結果更新到對應於自己的 GPU 中。

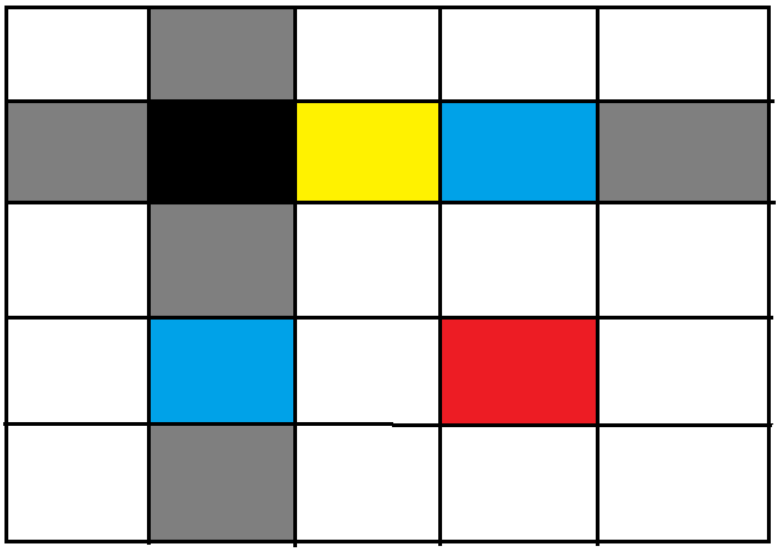
最後一點就是，因為 graph 的節點數不一定能被 Block Factor 整除，因此我的作法是把多補上一些不可能走道的假節點值到 N 可以被 BF 整除為止。

整體而言，這種演算法的好處在於 Block Factor 數大的時候仍可以充分利用 GPU 資源，經實驗結果會發現在 BF = 256 時會達到最好的效果。然而缺點是，因為每個 BLOCK 中只會有 BF 數個 Thread，所以如果 BF 數較小時可能會有太多的 Block，造成 GPU 資源硬體的 Context switch 的 overhead 太多。

Optimization Techniques:

以下則是我另外實作的，加入 Optimization Techniques 的版本。因為為了使用 Shared Memory，這部分的實作是跟上面使用不同的架構。這部分我的架構設計使得 Block Factor 一定要小於等於 32 才行。然而後續測試發現，跟原本的架構比起來，在 Block Factor <=32 時確實會有效能的改善，但是在原本的版本中的 BF=256 時的效能又比這種優化版本的效能還好(但是這個版本 BF 數上限是 32 QQ)。因此後面的實驗章節我使用的還是以上面的架構為主。(不過在相同的 Configuration(Block factor)時這種架構速度是有超過上面的原本架構，所以還是有優化成功的)。

Shared Memory:



圖(一)

使用 shared memory 的關鍵在於把彼此會使用相同資料的節點(以及對應的 threads)放到同一個 block 中，讓它們可以共享資源。因此針對 Blocked APSP 演算法，三個 PHASE 各自有不同的方式。且因為單一 block 的 thread 數上限是 1024，所以 block 的大小最多只能是 32*32。

想像上面的圖每一格 APSP 的 BLOCK 都是剛好對應到一個 Cuda block，而裏面都各自有 32*32 個 thread，而每個 thread 各自都負責更新一個對應的(I,j)距離。而因為 shared memory 的 Life time 跟 block 一樣，因此我們不能像前面一樣每個 iteration 都重新 launch 一個 kernel(在這個版本中也沒必要，因為同一個 APSP Block 的點都在同一個 cuda block 之中，因此可以使用 syncthreads())。下圖就是 shared memory 架構 launch kernel 的方式，跟原本的架構比起來，phase 1 跟 phase 2 並不會分成 BF 次來 LAUNCH，而是只 launch 一次。

```
for(int K=0;K<FWblockDim;K++){
        printf("K=%d phase1\n",K);
    // Phase 1
    floyd_warshall_1<<< 1,threadStr>>>( cuda_graph,K,K*bf);
    cudaCheckErrors("phase 1");


    if(FWblockDim>1){
        floyd_warshall_2<<< (FWblockDim-1)*2 ,threadStr>>>( cuda_graph,K,K*bf);
        cudaCheckErrors("phase 2 col");



        floyd_warshall_3<<<blockStr,threadStr>>>(cuda_graph,K,K*bf);

        cudaCheckErrors("phase 3");

    }
}
```

在 Phase 1 中，每次更新只會使用到同屬同一個 APSP block 的距離，因此在 Phase 1 中，block 中每個 thread 都各自去 Global memory 抓對應於自己(I,j)的那個距離放在 shared memory 中。在 Phase 1 的距離更新也就直接寫到 shared memory 裏，離開 kernel 的時候再寫回 global memory 就行了。(以下三段 code，cuda_bf 代表 block factor，cuda_tempVertex 代表總節點數，I,j 的計算方法就跟原本的架構一樣)

```
__shared__ int D[32*32];
D[threadIdx.y*cuda_bf + threadIdx.x] = dist[i*cuda_tempVertex + j];
__syncthreads();
// Put to shared memory???
```

在 Phase 2 中，假設以圖(一)為例，黑色的 block 是 phase 1 時更新的 block，而黃色的 block 則是目前要更新的，那麼同屬這個 block 的每個 thread 就個別去黑色的 block 和黃色的 block 各自抓取一個對應的距離放到 shared memory 裏來使用和更新。(每個 thread 負責兩個值)

```
__shared__ int D2[32*32*2];
D2[threadIdx.y * cuda_bf + threadIdx.x] = dist[i*cuda_tempVertex + j];
D2[(cuda_bf*cuda_bf) + (threadIdx.y *cuda_bf ) + (threadIdx.x)] = dist[ (kbf+threadIdx.y) * cuda_tempVertex + (kbf +threadIdx.x)];
__syncthreads();
```

在 Phase 3 中，以圖(一)為例，如果當前要更新的 block 是紅色的 block，則每個 thread 就分別去兩個藍色的 block 及紅色的 block 中抓取對應的距離放進 shared memory 中來使用和更新。(每個 thread 負責三個值)。

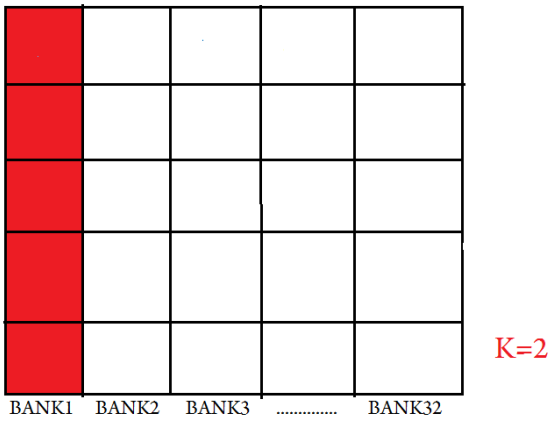
```
__shared__ int D3[32*32*3];
D3[threadIdx.y * cuda_bf + threadIdx.x] = dist[i*cuda_tempVertex + j];
D3[(cuda_bf*cuda_bf) + (threadIdx.y*cuda_bf) + threadIdx.x] = dist[(cuda_bf*idx+threadIdx.y)*cuda_tempVertex + (kbf + threadIdx.x)];
D3[(2*cuda_bf*cuda_bf) + (threadIdx.y*cuda_bf) + threadIdx.x] = dist[(kbf+threadIdx.y)*cuda_tempVertex + (idy*cuda_bf+threadIdx.x)];
__syncthreads();
```

Bank Conflict Avoidance

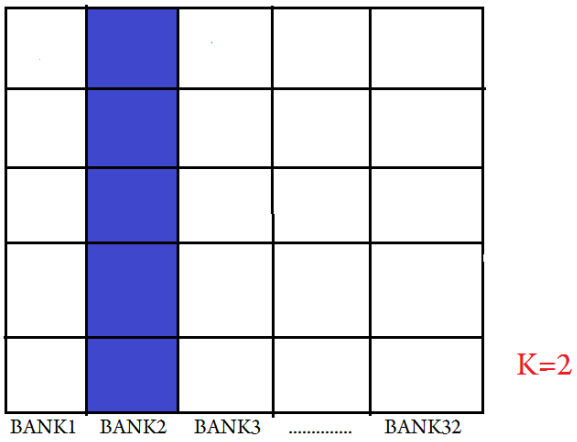
使用 Shared Memory 的另一個關鍵就在於要避免 bank conflict。Bank conflict 發生於同一個 warp 的 threads 存取同一個 bank 不同位址的資料。而在 APSP 演算法中，每回合我們需要去抓三個 shared memory 裡面的值：Dij,Dik,Dkj。以我的狀況來說，因為我一開始的 Indexing 方式如下：

```
int i = cuda_bf * idx + threadIdx.x;
int j = cuda_bf * idy + threadIdx.y;
```

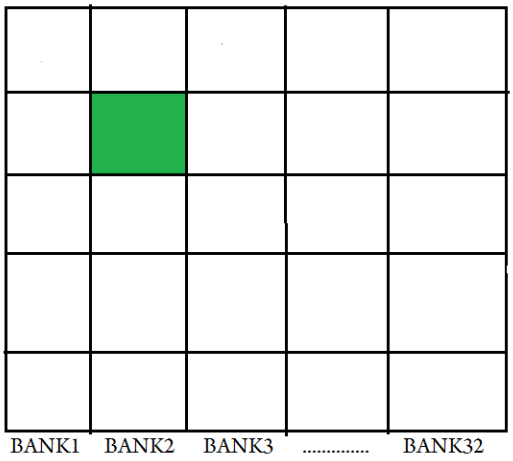
因為 cuda 裏用 2d dim 的結構來做 thread 的 indexing 時，似乎連續的 threadIdx.x 會被當成連續的 thread 而被排在同一個 warp 中。而這樣子的 indexing 方式會導致我的某個 warp 在 shared memory 裡的取值變成如下 (shared memory 為 32*32 的 2D int array，而在我測試的 gpu 中 bank 數為 32，i 是 row index，j 是 column index，假設目前是 K=2 的回合) :



Dij 取值



Dik 取值



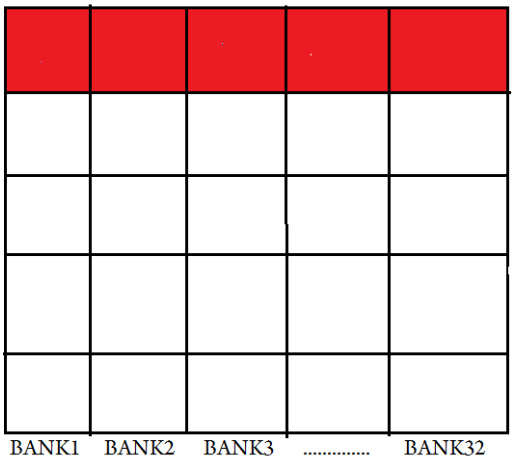
K=2

Dkj 取值

我們可以看到，在 Dij 和 Dik 的取值時，都會發生 bank conflict(Dkj 因為全部都是同一個位置所以不會)。而如果把 thread 跟(I,j) pair 的 indexing 方式改成：(也就是把 kernel 中所有 threadIdx 的 X/Y 互換)

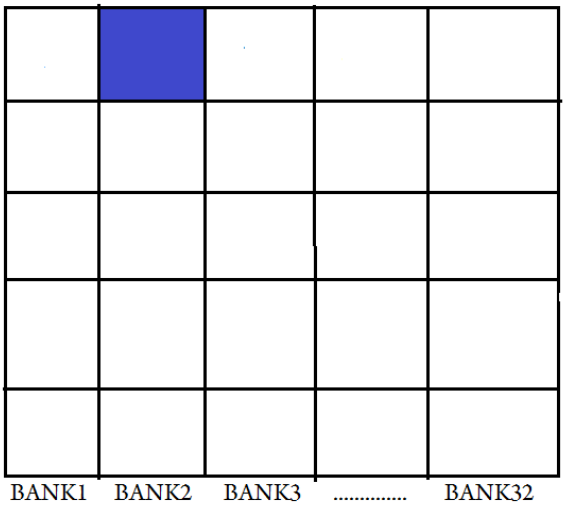
```
int i = cuda_bf * idx + threadIdx.y;
int j = cuda_bf * idy + threadIdx.x;
```

在一個回合中，同一個 warp 的取值就變成如下：

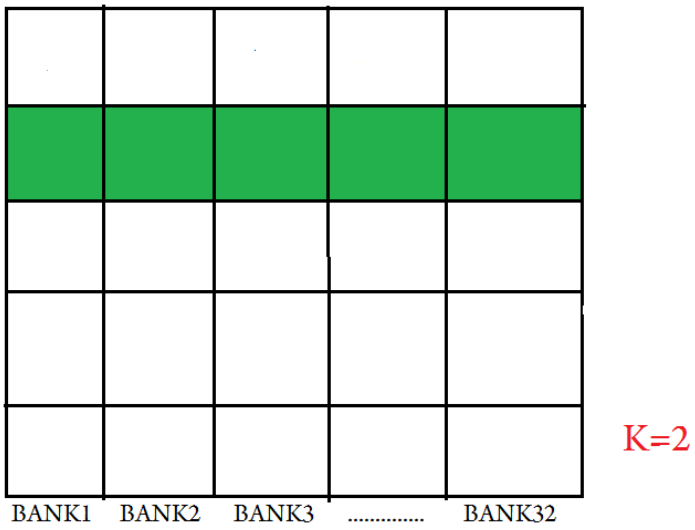


K=2

Dij 取值



Dik 取值



Dkj 取值

可以發現只要把 x/y 的 index 互換，bank conflict 就解決了，在以上三次存取中都沒有發生任何 bank conflict，效率因此獲得提升。

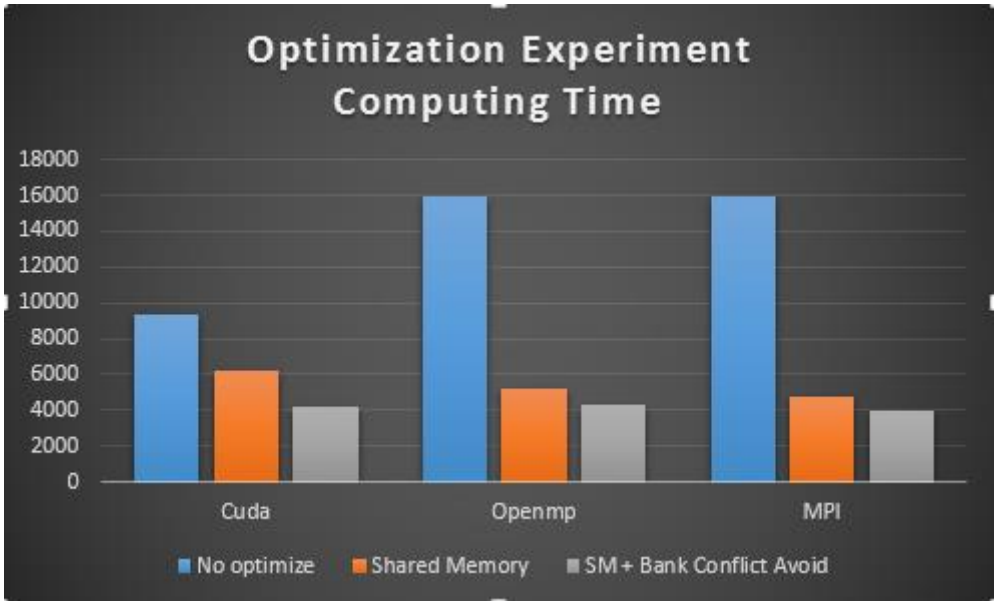
簡單的實驗證明 optimization 的效果如下：

```
cuda: N = 4096
BF = 32, without shared memory and bank conflict avoidance :  11449.762695  / 9402.161133 / 47.701569 / 1942.524780
BF = 32 ,Share memory without bank conflict avoidance :  8362.041321 / 6169.680176 / 51.505432 / 2140.855713
BF = 32 ,Share memory with bank conflict avoidance :    6281.979850 / 4220.048828 / 49.808721  / 2012.122301

openmp: N=6144
BF = 32, without shared memory and bank conflict avoidance : 23839.023438 / 16009.007812 / 3402.279053 / 4219.427734
BF = 32 ,Share memory without bank conflict avoidance : 13032.763149 / 5243.847412 / 3773.474609 / 4015.441128
BF = 32 ,Share memory with bank conflict avoidance :    12078.241717 / 4292.950684 / 3766.252930 / 4019.038103

MPI: N = 6144
BF = 32, without shared memory and bank conflict avoidance : 54012.710938 / 16012.513672 / 12227.518938 / 4521.569824 / 21010.119192
BF = 32 ,Share memory without bank conflict avoidance : 42959.9888582 / 5247.168074  /13215.9201782  /4410.579114  /20086.321492
BF = 32 ,Share memory with bank conflict avoidance : 41070.628547 / 4333.895020  /11941.594012   /4910.402602   /19884.736913

```



這是 Cuda 版本使用 N=4096，Multi GPU 版本使用 N=6144，在 BF=32 的情況下，三種版本對應的 computing time。可以發現 shared memory 會對整體速度有明顯的改善，而 bank conflict avoidance 可以再微幅改善。而在 Multi GPU 的版本中，改善的效過又優於 SINGLE GPU 的版本。

Profiling Result (N=6144, BF=256)

```
==26176== Profiling application: ./HW4_cuda.exe testcase/in5 out 256
==26176== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 68.53%  5.65916s        24  235.80ms  234.43ms  246.91ms  floyd_warshall_beta_3(int*, int, int, int)
 29.68%  2.45102s      6144  398.93us  356.92us  401.72us  floyd_warshall_beta_2(int*, int, int)
  0.66%  54.251ms      6144  8.8290us  6.4310us  10.748us  floyd_warshall_beta_1(int*, int, int)
  0.58%  48.035ms         4  12.009ms     768ns  48.032ms  [CUDA memcpy HtoD]
  0.55%  45.213ms         1  45.213ms  45.213ms  45.213ms  [CUDA memcpy DtoH]

==26176== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 86.44%  7.46935s     12312  606.67us  5.7240us  236.48ms  cudaLaunch
  7.89%  682.22ms         1  682.22ms  682.22ms  682.22ms  cudaDeviceSynchronize
  4.25%  366.98ms         8  45.872ms  1.4140us  366.97ms  cudaEventCreate
  1.09%  93.808ms         2  46.904ms  45.735ms  48.073ms  cudaMemcpy
  0.14%  12.502ms         1  12.502ms  12.502ms  12.502ms  cudaMalloc
  0.09%  7.8271ms     36960     211ns     162ns  10.903us  cudaSetupArgument
  0.04%  3.2180ms     12315     261ns     215ns  3.0020us  cudaGetLastError
  0.04%  3.1773ms     12312     258ns     219ns  1.6290us  cudaConfigureCall
  0.01%  715.68us         2  357.84us  337.74us  377.94us  cuDeviceTotalMem
  0.01%  701.01us       182  3.8510us     318ns  131.70us  cuDeviceGetAttribute
  0.00%  350.98us         3  116.99us  7.6260us  334.76us  cudaMemcpyToSymbol
  0.00%  330.16us         1  330.16us  330.16us  330.16us  cudaGetDeviceProperties
  0.00%  127.84us        12  10.653us  2.5910us  35.648us  cudaEventRecord
  0.00%  73.262us         2  36.631us  31.588us  41.674us  cuDeviceGetName
  0.00%  36.058us         6  6.0090us  3.1520us  13.470us  cudaEventSynchronize
  0.00%  15.200us         6  2.5330us  1.2830us  4.1200us  cudaEventElapsedTime
  0.00%  5.8740us         1  5.8740us  5.8740us  5.8740us  cudaSetDevice
  0.00%  5.3430us         3  1.7810us     531ns  4.2100us  cuDeviceGetCount
  0.00%  3.0880us         6     514ns     339ns     663ns  cuDeviceGet
```

Cuda Profiling

```
==26353== Profiling application: ./HW4_openmp.exe testcase/in5 out 256
==26353== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 44.63%  6.59435s        48  137.38ms  115.31ms  170.40ms  floyd_warshall_beta_3(int*, int, int, int, int)
 38.12%  5.63349s     12288  458.45us  356.85us  521.47us  floyd_warshall_beta_2(int*, int, int)
  8.50%  1.25566s        60  20.928ms     768ns  79.403ms  [CUDA memcpy HtoD]
  7.91%  1.16913s        49  23.860ms  21.072ms  45.677ms  [CUDA memcpy DtoH]
  0.84%  124.17ms     12288  10.105us  6.4690us  13.672us  floyd_warshall_beta_1(int*, int, int)

==26353== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 90.49%  7.95334s        54  147.28ms  3.9280us  213.15ms  cudaEventSynchronize
  3.62%  318.24ms     24624  12.923us  6.2800us  961.31us  cudaLaunch
  2.62%  229.98ms        51  4.5094ms  533.37us  79.895ms  cudaMemcpy
  1.36%  119.53ms         8  14.941ms  1.0150us  119.51ms  cudaEventCreate
  1.01%  88.645ms         2  44.322ms  446.54us  88.198ms  cudaMalloc
  0.46%  40.380ms     73968     545ns     163ns  872.09us  cudaSetupArgument
  0.21%  18.201ms     24677     737ns     248ns  868.06us  cudaGetLastError
  0.19%  16.337ms     24624     663ns     221ns  852.73us  cudaConfigureCall
  0.01%  880.77us       108  8.1550us  2.7420us  52.550us  cudaEventRecord
  0.01%  833.77us       182  4.5810us     328ns  181.48us  cuDeviceGetAttribute
  0.01%  728.18us         2  364.09us  363.29us  364.89us  cuDeviceTotalMem
  0.01%  659.78us       107  6.1660us     575ns  37.298us  cudaSetDevice
  0.01%  569.44us        10  56.943us  8.0920us  321.20us  cudaMemcpyToSymbol
  0.00%  272.83us         1  272.83us  272.83us  272.83us  cudaGetDeviceProperties
  0.00%  127.57us        54  2.3620us  1.0490us  7.7430us  cudaEventElapsedTime
  0.00%  79.559us         2  39.779us  35.409us  44.150us  cuDeviceGetName
  0.00%  23.256us         2  11.628us  5.2290us  18.027us  cudaDeviceSynchronize
  0.00%  7.0870us         3  2.3620us     526ns  5.8690us  cuDeviceGetCount
  0.00%  3.2410us         6     540ns     335ns     764ns  cuDeviceGet
```

Openmp Profiling

```
======== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 46.92%  3.79743s        24  158.23ms  156.74ms  169.59ms  floyd_warshall_beta_3(int*, int, int, int, int)
 39.32%  3.18185s      6144  517.88us  467.89us  521.47us  floyd_warshall_beta_2(int*, int, int)
  6.49%  524.99ms        30  17.500ms     864ns  46.082ms  [CUDA memcpy HtoD]
  6.41%  518.45ms        24  21.602ms  20.738ms  22.640ms  [CUDA memcpy DtoH]
  0.87%  70.079ms      6144  11.406us  8.1840us  13.548us  floyd_warshall_beta_1(int*, int, int)

======== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 97.64%  8.01951s        49  163.66ms  18.800ms  313.75ms  cudaMemcpy
  1.06%  87.106ms         1  87.106ms  87.106ms  87.106ms  cudaMalloc
  0.95%  77.703ms     12312  6.3110us  5.7700us  54.351us  cudaLaunch
  0.13%  10.774ms       182  59.196us     335ns  7.8336ms  cuDeviceGetAttribute
  0.12%  9.9062ms     36984     267ns     158ns  427.52us  cudaSetupArgument
  0.04%  3.1638ms     12365     255ns     216ns  11.657us  cudaGetLastError
  0.04%  2.9751ms     12312     241ns     209ns  10.672us  cudaConfigureCall
  0.01%  907.17us         2  453.58us  434.13us  473.03us  cuDeviceTotalMem
  0.01%  850.85us         2  425.43us  56.580us  794.27us  cuDeviceGetName
  0.00%  209.04us         5  41.808us  7.5670us  170.71us  cudaMemcpyToSymbol
  0.00%  157.65us        27  5.8380us  2.4110us  35.485us  cudaSetDevice
  0.00%  15.206us         1  15.206us  15.206us  15.206us  cudaDeviceSynchronize
  0.00%  7.0690us         3  2.3560us     561ns  5.1700us  cuDeviceGetCount
  0.00%  4.7460us         6     791ns     378ns  1.1600us  cuDeviceGet
```

MPI (process rank=0) Profiling

從 cuda 版本中我們可以看到，phase 3 的 kernel 確實是消耗最多時間((N/BF-1)*(N/BF-1)個 block)，其次是 Phase 2((N/BF-1)*2 個 block)，再來是 Phase 1(1 個 BLOCK)。

另一個有趣的現象我們可以看到 openMP 的總時間和 MPI 的 kernel 執行總時間是差不多的；但是 openMP 的 Kernel function 時間似乎有重疊，從這部份我們可以推論不論哪個 thread 在使用 kernel function，時間都會疊加紀錄上去，而 MPI 版本中因為兩個 gpu 分別是兩個不同的 process 控制，所以 kernel 時間加總大致就是 kernel finction 加總的時間。

另外一點有趣的發現是，儘管 openMP 版本只需要每個 iteration 完呼叫一次 Memcpy(Device to Device)就好，但是實際上可以發現 cuda 似乎還是會先把資料從 GPU 讀回 Host 之後再傳給另一個 GPU，而不是直接傳過去，因此 OpenMP

版本的 Memcpy 時間並不比 MPI 分開兩次(D2H/H2D)來的快。

在 CPU 的方面，可以看到其實 cuda 大部分的 API 其實幾乎都不怎麼花時間的。至於我的 cudaEventSynchronize 在 multi GPU 版本會佔大部分時間主要是因為我在 GPU 之間的溝通都有使用 cudaEventRecord 來記錄時間，因此在傳送完之後會做 cudaEventSynchronize，因此傳送 GPU 資料的時間就會全部加到 cudaEventSynchronize 上頭。

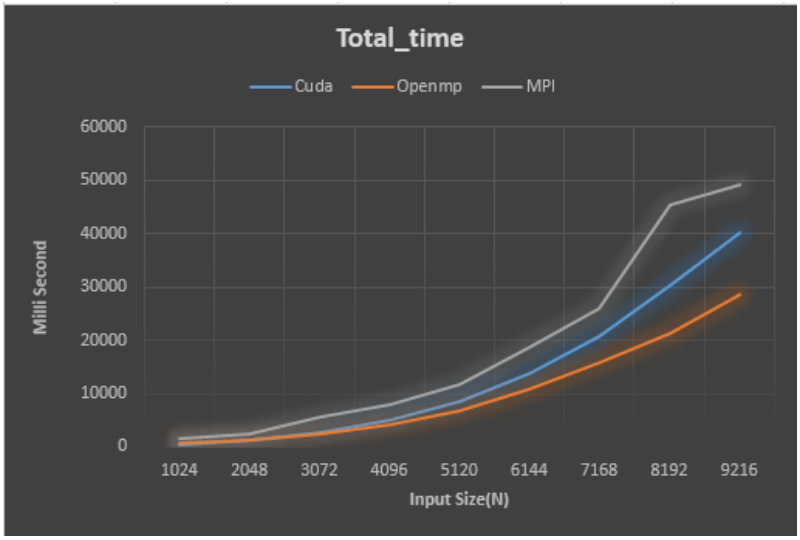
Experiment & Analysis

Weak scalability 的實驗我使用的 Block Factor 固定為 256。MPI 的設定為 Node=2:PPN=1

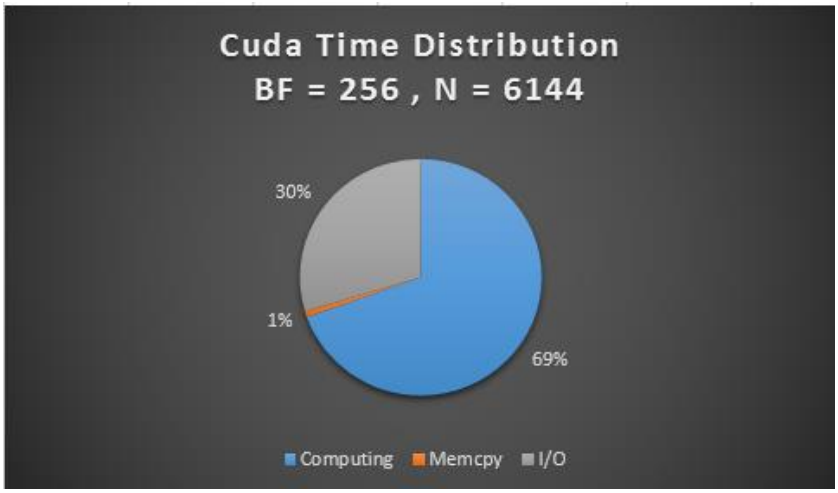
為了避免課程的 cluster 阻塞的問題，我使用的機器則是自己的機器，規格如下：

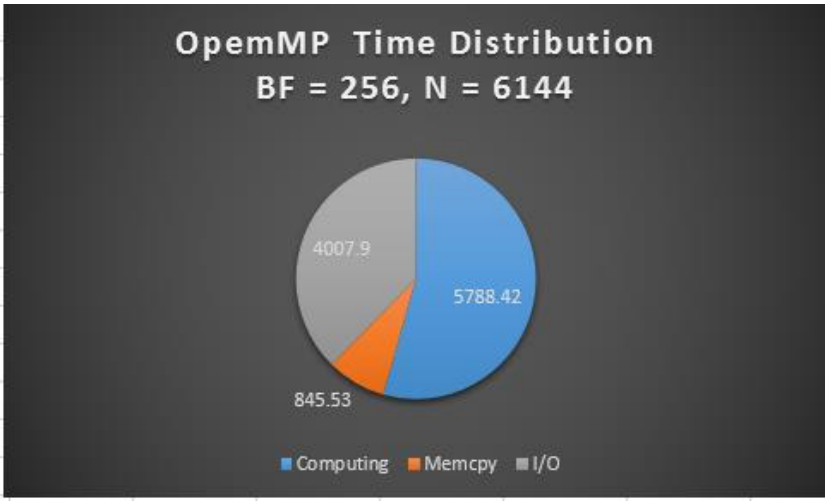
Specifications	NVIDIA GRID K2
Generic SKU reference	•699-52055-0010-000: Airflow intake from bracket •699-52055-0020-000: Airflow exhaust to bracket
Chip	2× GK104
Processor clock	745 MHz
Memory clock	2.5 GHz
Memory size	4 GB per GPU (8 GB per board)
Memory I/O	256-bit GDDR5
Memory configuration	32 pieces of 128M x 16 GDDR5 SDRAM
Display connectors	None
Power connectors	•1x 8-pin PCI Express power connector •1x 6-pin PCI Express power connector
Total board power	225 W

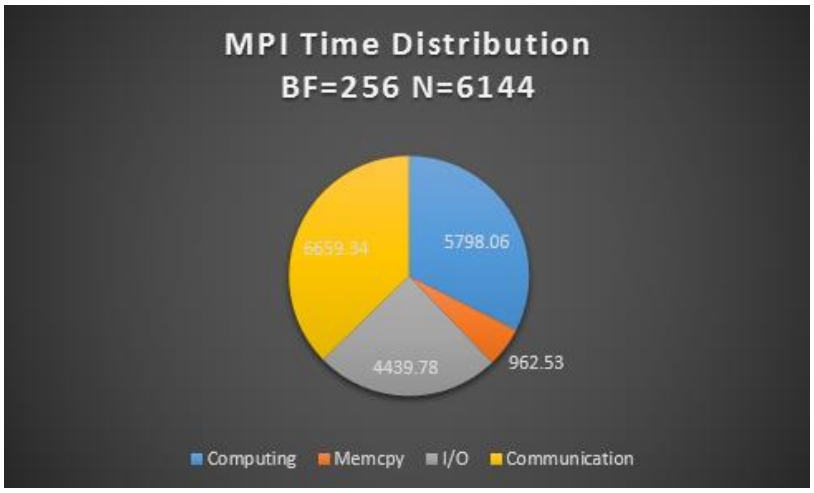
Weak Scalability & Time Distribution：



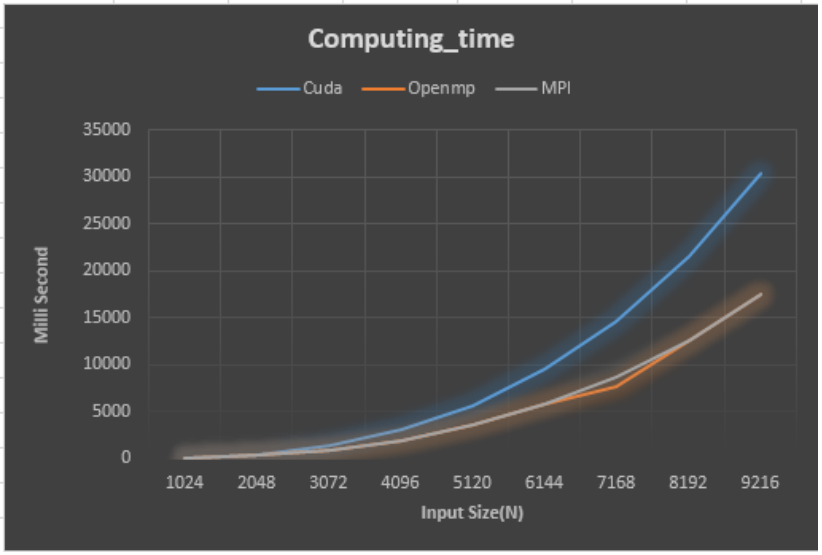
首先是總時間，我們可以看到在我的實作中，OpenMP 的版本是最快的，而 Single GPU 版本次之，最慢的反而是 MPI 版本。這部分的原因後續會分析，主要是因為 MPI 的 Communication 時間太長導致。而 OpenMP 大約可以達到 Single GPU 時 60~70%的時間，這部分主要是因為 Phase 3 的計算拆開由兩塊 GPU 同時執行，但是又因為有 Memcpy 的 Overhead 導致無法達到近 50%的時間。觀察上圖不難發現時間隨著 input size 的增加呈現凹形曲線，可以與 Floyd Warshall Time Complexity(N^3)互相驗證。



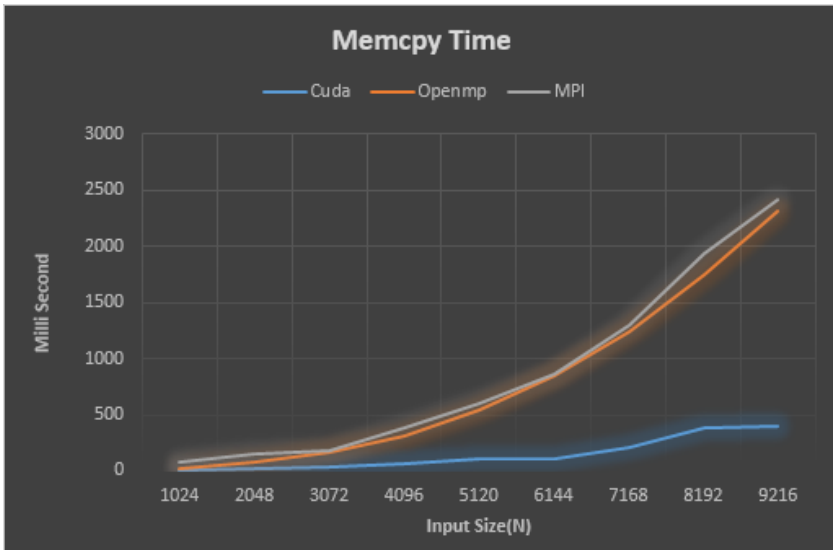




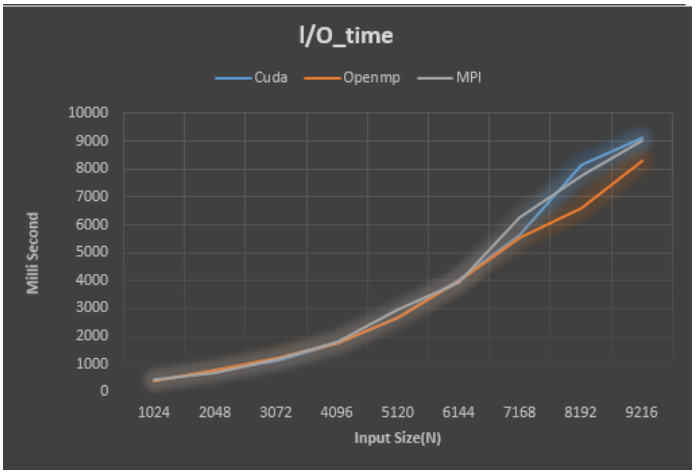
從 Time Distribution 圓餅圖中可以看出，在 Cuda 版本中 Computing Time 佔了大多數的時間且 I/O 也有一定分量；而到了 OpenMP，在 I/O 時間基本上一樣的狀態下，儘管 Computing Time 有減少(兩塊 GPU 同時處理 Phase 3)，Memcpy 佔的比例卻開始變得不容忽視，但是整體還是以 Computing Time 為主。不過在 MPI 版本中，我們可以看到 Communication 反而成了最大的 Bottleneck。其實在 MPI 版本中，Communication 會佔這麼大的比例一部份原因我認為是只有兩塊 GPU 可以用。在 Multi GPU 的實作中，只要有牽涉到資料的傳遞，不論 GPU 是幾塊，總傳輸量是一樣的(有三塊的話每個 GPU 要處理的量就變成原本的 1/3)。所以如果每個 Process 可以使用的 GPU 超過兩塊的話，傳輸總量不變，每個 Process 的計算資源變多，整體而言 MPI 的總時間應該會逐漸呈線性減少。



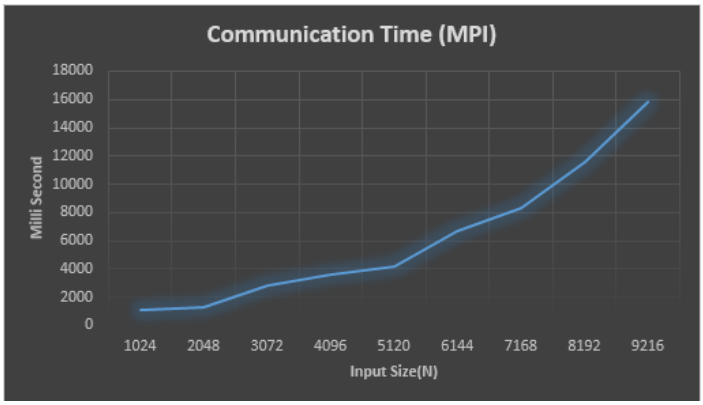
看到 Computing Time 的部分(Kernel runtime)。這部分就幾乎是跟理論值一樣的狀況，在 GPU 有兩塊的狀態下，儘管有分開來計算的只有 Phase 3，但是實際上 Phase3 佔了總計算量的九成以上，所以實際上幾乎使可以達到近兩倍的計算效率。而實際上觀察 Multi GPU 和 Single GPU 的 Computing Time，確實也達到了幾乎兩倍的速度。而 OpenMP 和 MPI 版本的速度幾乎是一模一樣的，這部分也可以驗證前面 OpenMP 和 MPI 版本的總時間差異是來自於 MPI 的 Communication 的假設。



Memory copy 的部分，cuda 因為只需要開始跟結束各自 copy 一次就好，自然是時間最少的，而 OpenMP 跟 MPI 的時間相近的原因我已經在 Profiling section 說明了，不過也許從上圖我們可以推論，呼叫一次 cudaMemcpy(D2D)還是比分開來兩次呼叫 cudaMemcpy(D2H)、cudaMemcpy(H2D)有快一點，也許 CUDA 在這方面有額外的優化？



I/O 時間的部分則跟預期的完全一樣，因為不論是哪種版本要讀和寫都是由 CPU 單獨執行，無法平行化。而時間隨著 Input Size 大致呈現平方倍的成長，這點則是因為 input 和 output 都是 N*N 的 graph。



Communication 時間是我的 MPI 版本的 Bottleneck，可以看到，因為每次的傳輸都是 N*N 的資料量，整體曲線試呈大致平方倍成長的。也許是因為我使用的機器 Message passing 的速度本身就很慢的關係？

Blocking Factor effect

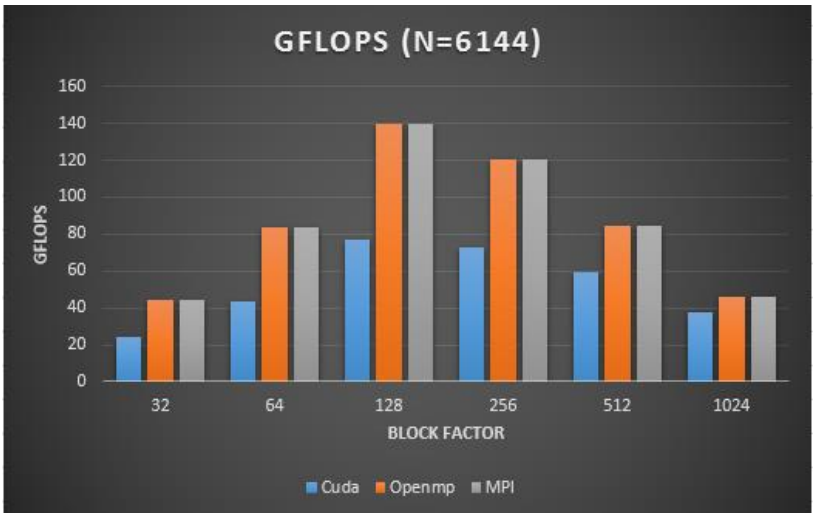
在 GFLOPS 的計算中，我們要計算的是每秒的總共有幾十億個運算被執行，考量到 Floyd Warshall 的核心計算：

```
4   while(x<bf){
5       temp = d[i][k] +d[k][j];
6       if(d[i][j]>temp){
7           d[i][j] = temp;
8       }
9       x++;
10  }
```

是四個運算，再加上每個 kernel 的前置 index 轉換等計算，我計算 GFLOPS 的公式是

*GFLOPS = (10+ 4 * (BF))*(N/BF) * N^2 / (10^9 * computing Time)*

紅色是單一 kernel function 本身的運算次數，藍色是同一個(I,j)pair kernel 會被呼叫的次數，綠色則是總共會有多少個 thread 會同時執行這個 function。



以上則是 GFLOPS 的計算結果，可以發現 GFLOPS 一開始會隨著 BF 的增加而逐漸增加，這部分我認為是因為在小的 BF 時我的架構可能沒辦法有效榨乾 GPU 的運算資源(因為總 Block 數會太多，可能無法善用每個 SM)，而 BF 超過 128 後，GFLOPS 又會開始下降，我認為則是因為 block 裏的 thread 數(我的架構每個 block 裏的 thread 數是跟 bf 一樣的)過多，硬體的 context switch 會太頻繁導致效率下降。因此可以看出針對不同實作方法與切資料的方法選用適當的 Block factor 是很重要的。

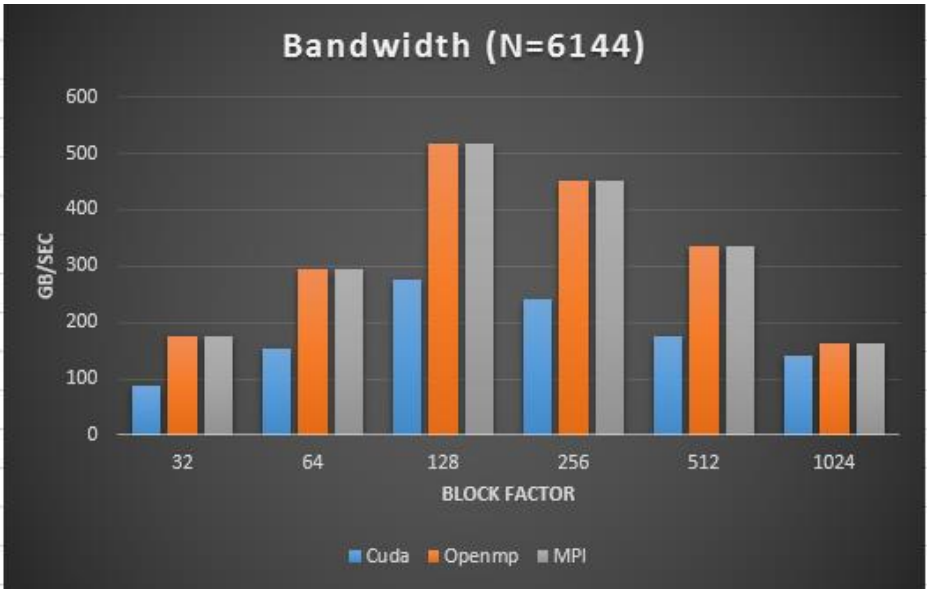
至於 Bandwidth 的計算，根據 Nvidia 官方手冊的寫法：

$$BW_{Effective} = (R_B + W_B) / (t * 10^9)$$

RB 跟 WB 代表的則是在所有被 launch 過的 GPU 運算中讀寫的 byte 的總數。因為在 Floyd Warshall 算法中，每個回合我們都需要(1)讀取 Dik (2)讀取 Dkj (3)可能把結果寫回 Dij 中，如果假設寫回 dij 的機率是 0.5，則當成每個回合總共有 2.5 次 Read/write。因此，我計算 Bandwidth 的方法如下：

*Bandwidth = 2.5 * 4 * BF * (N/BF) * N^2 / (10^9 * computing Time)*

其中 4 表示 int 有 4 個 byte。紅色代表每次 kernel function 總共的 read/write 次數，藍色是同一個(I,j)pair kernel 會被呼叫的次數，綠色則是總共會有多少個 thread 會同時執行這個 function。



看的出來 Bandwidth 隨著 Block factor 呈現的趨勢其實跟 GFLOPS 很像，同樣都是逐漸增加，到了 128 時最大，其後再下降。不過另一點比較特別的是，依照我的機器規格，我算出來我的理論 Bandwidth 最佳值應該是 200GB/SEC 左右，但是上圖中許多結果都超出了理論最佳值。我認為最可能的原因大概是因為 SIMD 的架構下不同的 threads 可能很多都在某個相同的指令去存取同一個 Global Memory 的位置，因此可以用 broadcast 的方式，讀取一次再 broadcast 給所有需要使用該值的 processor。不過這部份牽涉到很多 GPU 內部運作的細節，不太容易計算。

Experience & Conclusion

在這次的作業中，充分練習到了 Cuda Programming 以及如何設計資料結構和切割資料的方式來有效運用 GPU SIMD 架構的特性，以及一些優化 Cuda Program 的技巧。另外就是了解到了如何運用 Multi GPU 進一步加速運算，以及可能遇到的問題。

如果要說這次作業最困難的地方，大概就在於一開始沒有搞清楚 Blocked APSP 特性以及不同階段的資料相依的性質，導致一開始寫的時候怎麼寫結果都是錯的；也許可以當成借鏡，知道再要寫平行程式前，充分了解問題特性以及運算資源的特性，設計適當的演算法是非常重要的。