

# Homework 4: Blocked All-Pairs Shortest Path

107062612 熊祖玲

## I. Implementation

### A. Single-GPU

在這個問題中，需要使用一個 GPU 實作 Blocked All-Pairs Shortest Path algorithm 從給定的 directed graph with non-negative edge weight 中，找到所有 vertex 中任兩點的最短路徑。

在這個問題中，我選擇用一維陣列來儲存 Adjacency Matrix，因此資料大小為所有 Vertex 數量 ( $V$ ) 乘上所有 Vertex 數量 ( $V$ )，並將不存在的 edge 的權重設為無限大。

#### 1. 資料切割

在開始計算之前，必須決定一個 Block 的大小，我根據 GPU 中一個 Block 最多可使用的 Thread 數量 ( $T$ ) 為依據，將一個 Block 設定為  $\lceil \sqrt{T} \rceil \times \lceil \sqrt{T} \rceil$ ，若  $V$  不為  $\lceil \sqrt{T} \rceil$  的倍數，則將  $V$  做 Padding，使得  $V + \lceil \sqrt{T} \rceil > V_{padding} \geq V$ ， $V_{padding} = \lceil \sqrt{T} \rceil \times B$  ( $B$  為一個 column 中的 Block 數量)。

如下二圖所示：原本的 Vertex 數量為 14，GPU 中一個 Block 最多可使用的 Thread 數量為 9，所以將 Block 設定為  $3 \times 3$  的大小，因為 14 無法整除 3，所以對原本的 Adjacent matrix 做擴充，成為  $15 \times 15$  的大小，使得新的 Adjancent matrix 可以剛好被分成相同大小的 25 個 Block。

	0	1	2	3	4
0					
1					
2					
3					
4					

圖一、原本的 ADJACENCY MATRIX

	0	1	2	3	4
0					
1					
2					
3					
4					

圖二、PADDING 後的 ADJACENCY MATRIX

會使用上述方法做資料切割是因為要實作的演算法是以 Block 為單位做運算，若有一些 Block 的大小和不同，會需要使用條件式做判斷，然而在 CUDA 中使用條件式會導致降低程式的效能。此外，在 Launch kernel 時，向 GPU 要資源的時候也是以 Block 作為基礎，所以我希望讓 Block 中的一個路徑對應到一個 Thread，因此我使用 `cudaGetDeviceProperties(prop, 0)` 和 `prop.maxThreadPerBlock` 取得  $T$ ，並將一個 Block 設定為  $\lceil \sqrt{T} \rceil \times \lceil \sqrt{T} \rceil$ 。

## 2. 執行演算法

因為 Blocked all-pairs shortest path algorithm 分為三個階段，所以我也分成三個階段，每個階段分別 Launch 一個 kernel，根據 B 做為需要執行的 Iteration 數量。

### a) Phase 1

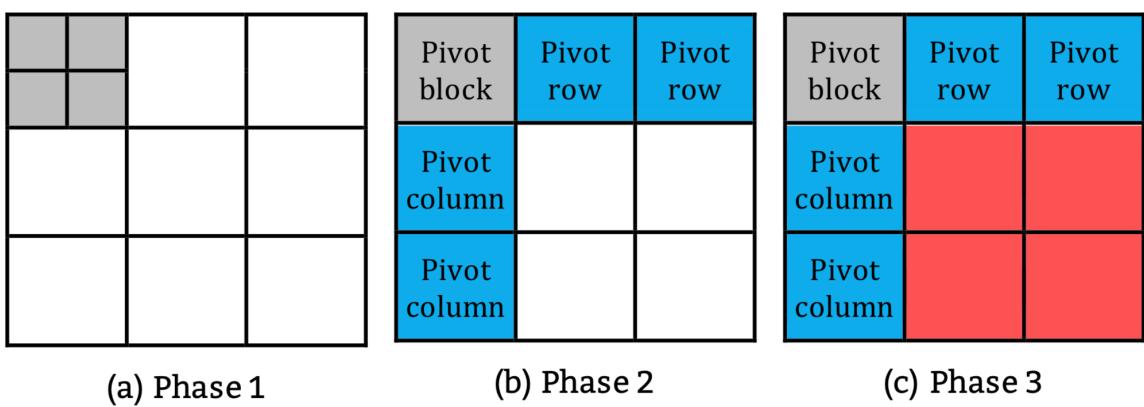
在第  $r$  個 Iteration 時，Launch 一個 Block 數量為 1 且 Block 中 Thread 為 2D 矩陣 ( $\lceil \sqrt{T} \rceil, \lceil \sqrt{T} \rceil$ ) 的 kernel，對 Block ( $r, r$ ) (如圖三中灰色區塊) 做 Floyd-Warshall algorithm 運算。

### b) Phase 2

在第  $r$  個 Iteration 時，Launch 一個 Block 為 2D 矩陣 ( $B-1, 2$ ) 且 Block 中 Thread 為 2D 矩陣 ( $\lceil \sqrt{T} \rceil, \lceil \sqrt{T} \rceil$ ) 的 kernel，對所有在第  $r$  行、第  $r$  列的 Block (如圖三中藍色區塊) 做 Floyd-Warshall algorithm 運算，在做這些 Block 的運算時，會需要 Block( $r, r$ ) 的資訊，因為我有使用 Shared memory 所以需要額外將 Block( $r, r$ ) 的資訊複製到 Shared memory 中，然而，這裡必須注意的是：Shared memory 只能是一維的陣列，我接續需運算的區塊，將此資訊放在 Shared memory 的陣列中。

### c) Phase 3

在第  $r$  個 Iteration 時，Launch 一個 Block 為 2D 矩陣 ( $B-1, B-1$ ) 且 Block 中 Thread 為 2D 矩陣 ( $\lceil \sqrt{T} \rceil, \lceil \sqrt{T} \rceil$ ) 的 kernel，對所有不屬於第  $r$  行、第  $r$  列的 Block( $i, j$ ) (如圖三中紅色區塊) 做 Floyd-Warshall algorithm 運算，在做這些 Block 的運算時，會需要 Block( $i, r$ ) 及 Block( $r, j$ ) 的資訊，所以需要額外將這兩個 Block 的資訊複製到 Shared memory 中。

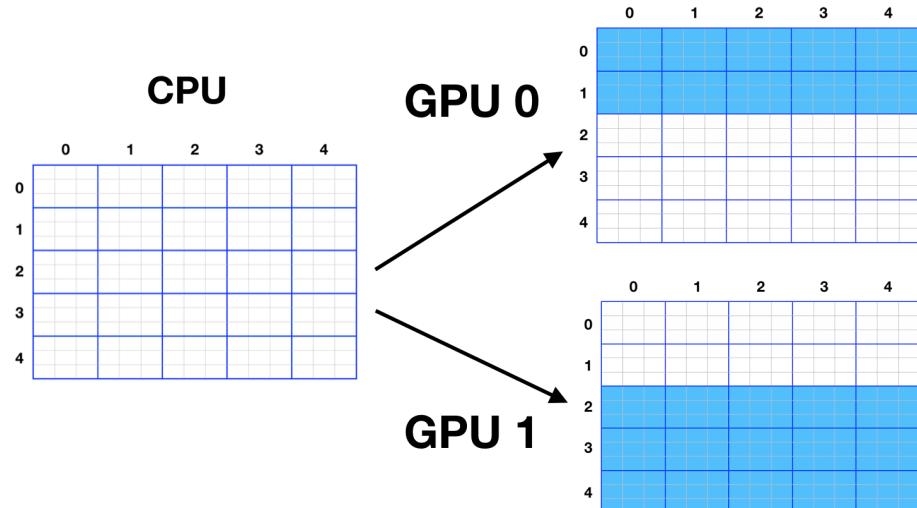


圖三、每個階段運算區塊示意圖

## B. Multi-GPU implementation in the single node

### 1. 資料分割

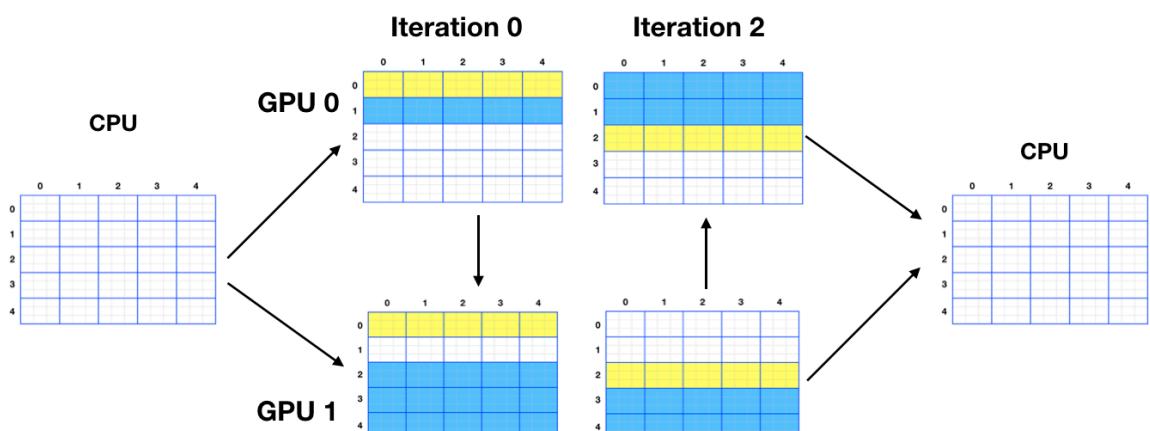
這個部分的資料切割方法大致與上述相同，先將 Adjacent matrix 做好資料分割，並且以 Block 數量為單位分成上下兩部分，Thread 0 被分配到上半部的資料，而 Thread 1 被分配到下半部的資料，作為 GPU 運算的資料。



圖三、MULTI-GPU資料分配示意圖

### 2. 執行演算法

首先，在第  $r$  個 Iteration 開始前，必須將第  $r$  個 row 的資訊透過 DeviceToDevice 的方式由 GPU  $i$  傳給 GPU  $j$ ，接下來進入三個階段的運算。如圖四為例，在 Iteration 0 的時候，GPU 0 將第 0 列的 Block 中所有資訊傳給 GPU 1，再開始執行三階段運算；然而，到了 Iteration 2 時，GPU 1 將第 2 列的 Block 中所有資訊傳給 GPU 0，再執行三階段運算，所有 Iteration 都執行完畢後，GPU 0 (1) 再將上(下)半部的資料複製到 Host 端。



圖四、MULTI-GPU資料溝通示意圖

a) Phase 1

在第 r 個 Iteration 時，2 個 GPU 都必須 Launch 一個 Block 數量為 1 且 Block 中 Thread 為 2D 矩陣 ( $\lceil \sqrt{T} \rceil, \lceil \sqrt{T} \rceil$ ) 的 kernel，對 Block (r, r) 做 Floyd-Warshall algorithm 運算。

b) Phase 2

在第 r 個 Iteration 時，Launch 一個 Block 為 2D 矩陣 (B-1, 2) 且 Block 中 Thread 為 2D 矩陣 ( $\lceil \sqrt{T} \rceil, \lceil \sqrt{T} \rceil$ ) 的 kernel，對所有在第 r 行、第 r 列的 Block 做 Floyd-Warshall algorithm 運算。

c) Phase 3

在第 r 個 Iteration 時，Launch 一個 Block 為 2D 矩陣 (B/2, B-1) 且 Block 中 Thread 為 2D 矩陣 ( $\lceil \sqrt{T} \rceil, \lceil \sqrt{T} \rceil$ ) 的 kernel，對所有不屬於第 r 行、第 r 列且 r 是屬於當前 GPU 的範圍 的 Block(i, j) (如圖三中紅色區塊) 做 Floyd-Warshall algorithm 運算。

## C. Multi-GPU implementation with MPI

此方法大致上與 Multi-GPU implementation in the single node 的方法相同，只有在通的時候不同，原本在每個 Iteration 開始前，兩個 GPU 透過 DeviceToDevice 的方式交換資訊，然而在 MPI 版本中，必須先從 Device 複製到 Host，再透過 Send、Recv 交換兩個 node 的資料，最後透過 HostToDevice 的方式再複製到 Device 上。

## II. Profiling Results

### A. Single-GPU

```
[ditto@hades01 test]$ srun -n 2 --gres=gpu:1 -ppp nvprof .../multi_node ./1414.in ./a.out
==22538== NVPROF is profiling process 22538, command: .../multi_node ./1414.in ./a.out
==22536== NVPROF is profiling process 22536, command: .../multi_node ./1414.in ./a.out
==22536== Profiling application: .../multi_node ./1414.in ./a.out
==22538== Profiling application: .../multi_node ./1414.in ./a.out
==22538== Profiling result:
          Type  Time(%)    Time   Calls    Avg     Min     Max   Name
GPU activities:  98.23% 55.2469s   663  83.329ms  82.805ms  89.107ms FW3(int*, unsigned long, int, unsigned long, int, int)
               0.74% 416.61ms   333  1.2511ms  417.92us  277.48ms [CUDA memcpy HtoD]
               0.74% 414.61ms   332  1.2488ms  412.83us  270.88ms [CUDA memcpy DtoH]
               0.29% 162.61ms   663  245.26us  242.56us  256.74us FW2(int*, unsigned long, int, unsigned long)
               0.01% 3.8759ms   663  5.8450us  5.6640us  7.8080us FW1(int*, unsigned long, int)
==22536== Profiling result:
          Type  Time(%)    Time   Calls    Avg     Min     Max   Name
GPU activities:  98.22% 55.2147s   663  83.280ms  81.926ms  88.125ms FW3(int*, unsigned long, int, unsigned long, int, int)
               0.74% 416.86ms   333  1.2518ms  412.03us  272.31ms [CUDA memcpy DtoH]
               0.74% 416.59ms   332  1.2548ms  417.83us  277.94ms [CUDA memcpy HtoD]
               0.29% 162.67ms   663  245.36us  242.11us  256.71us FW2(int*, unsigned long, int, unsigned long)
               0.01% 3.8629ms   663  5.8260us  5.6320us  6.9440us FW1(int*, unsigned long, int)
API calls:      98.17% 28.8396s   665  43.368ms  434.21us  356.90ms cudaMemcpy
               1.77% 519.81ms   1  519.81ms  519.81ms  519.81ms cudaHostAlloc
               0.04% 12.233ms   1989  6.1500us  2.9900us  31.881us cudaLaunch
               0.01% 2.2078ms   1  2.2078ms  2.2078ms  2.2078ms cudaMalloc
               0.00% 1.0732ms   2  536.60us  13.329us  1.0599ms cudaFree
               0.00% 1.0433ms   8619  12lns   75ns  3.7630us cudaSetupArgument
               0.00% 673.41us   94  7.1630us  122ns  331.94us cuDeviceGetAttribute
               0.00% 482.19us   1989  242ns   105ns  1.8220us cudaConfigureCall
               0.00% 155.78us   1  155.78us  155.78us  155.78us cuDeviceTotalMem
               0.00% 42.445us   1  42.445us  42.445us  42.445us cuDeviceGetName
               0.00% 9.7400us   1  9.7400us  9.7400us  9.7400us cudaSetDevice
               0.00% 2.1160us   3  705ns   193ns  1.4650us cuDeviceGetCount
               0.00% 1.9430us   2  971ns   204ns  1.7390us cuDeviceGet
               API calls:  98.16% 29.0999s   665  43.759ms  431.04us  357.02ms cudaMemcpy
               1.78% 527.57ms   1  527.57ms  527.57ms  527.57ms cudaHostAlloc
               0.04% 12.804ms   1989  6.4370us  3.0420us  41.523us cuLaunch
               0.01% 1.6530ms   1  1.6530ms  1.6530ms  1.6530ms cudaMalloc
               0.01% 1.6342ms   2  817.12us  11.449us  1.6228ms cudaFree
               0.00% 1.0040ms   94  10.680us  103ns  538.67us cuDeviceGetAttribute
               0.00% 983.16us   8619  114ns   76ns  2.4840us cudaSetupArgument
               0.00% 495.35us   1989  249ns   102ns  2.0940us cudaConfigureCall
               0.00% 119.46us   1  119.46us  119.46us  119.46us cuDeviceTotalMem
               0.00% 44.191us   1  44.191us  44.191us  44.191us cuDeviceGetName
               0.00% 9.2680us   1  9.2680us  9.2680us  9.2680us cudaSetDevice
               0.00% 1.4690us   2  734ns   120ns  1.3490us cuDeviceGet
               0.00% 1.4110us   3  470ns   136ns  1.0810us cuDeviceGetCount
```

## B. Multi-GPU implementation in the single node

```
[ditto@hades01 test]$ srun -n 1 --gres=gpu:2 -ppp nvprof ./multi_gpu ./1414.in ./a.out
srun: job 132604 queued and waiting for resources
srun: job 132604 has been allocated resources
==21257== NVPROF is profiling process 21257, command: ./multi_gpu ./1414.in ./a.out
==21257== Profiling application: ./multi_gpu ./1414.in ./a.out
==21257== Profiling result:
      Type Time(%)     Time    Calls      Avg      Min      Max   Name
GPU activities:  97.04%  46.3374s  1326  34.945ms  34.524ms  36.502ms  FW3(int*, int, int, int, int, int)
               1.18%  561.31ms   665  844.08us  423.20us  140.08ms  [CUDA memcpy HtoD]
               1.17%  557.12ms   665  837.77us  420.67us  136.67ms  [CUDA memcpy DtoH]
               0.61%  291.13ms   1326  219.55us  215.65us  229.73us  FW2(int*, int, int, int)
               0.01%  6.1619ms   1326  4.6460us  4.5440us  5.5040us  FW1(int*, int, int)
API calls:    97.61%  47.4056s  1326  35.751ms  35.269ms  37.317ms  cudaDeviceSynchronize
               1.16%  563.09ms   667  844.21us  14.439us  140.10ms  cudaMemcpy
               0.82%  396.73ms    1  396.73ms  396.73ms  396.73ms  cudaHostAlloc
               0.37%  178.75ms    2  89.376ms  1.1836ms  177.57ms  cudaMalloc
               0.03%  16.930ms   3978  4.2560us  3.0140us  234.33us  cudaLaunch
               0.00%  2.0632ms   17238   119ns   74ns  2.1760us  cudaSetupArgument
               0.00%  833.59us   188  4.4340us  221ns  185.42us  cuDeviceGetAttribute
               0.00%  682.58us   3978   171ns   96ns  1.8780us  cudaConfigureCall
               0.00%  403.31us    1  403.31us  403.31us  403.31us  cudaGetDeviceProperties
               0.00%  338.70us    2  169.35us  168.43us  170.27us  cuDeviceTotalMem
               0.00%  84.139us    2  42.069us  37.886us  46.253us  cuDeviceGetName
               0.00%  18.437us    3  6.1450us  3.3210us  11.426us  cudaSetDevice
               0.00%  14.867us    2  7.4330us  2.2120us  12.655us  cudaFree
               0.00%  3.6500us    1  3.6500us  3.6500us  3.6500us  cudaGetDeviceCount
               0.00%  3.5120us    4   878ns   261ns  2.7000us  cuDeviceGet
               0.00%  2.3870us    3   795ns   282ns  1.6450us  cuDeviceGetCount
               0.00%  1.7200us    2   860ns   534ns  1.1860us  cuDeviceGetDevice
```

## C. Multi-GPU implementation with MPI

```
[ditto@hades01 test]$ srun -n 2 --gres=gpu:1 -ppp nvprof ./multi_node ./1414.in ./a.out
==22538== NVPROF is profiling process 22538, command: ./multi_node ./1414.in ./a.out
==22538== NVPROF is profiling process 22536, command: ./multi_node ./1414.in ./a.out
==22536== Profiling application: ./multi_node ./1414.in ./a.out
==22538== Profiling application: ./multi_node ./1414.in ./a.out
==22538== Profiling result:
      Type Time(%)     Time    Calls      Avg      Min      Max   Name
GPU activities:  98.23%  55.2469s   663  83.329ms  82.805ms  89.107ms  FW3(int*, unsigned long, int, unsigned long, int, int)
               0.74%  416.61ms   333  1.2511ms  417.92us  277.48ms  [CUDA memcpy HtoD]
               0.74%  414.61ms   332  1.2488ms  412.83us  270.88ms  [CUDA memcpy DtoH]
               0.29%  162.61ms   663  245.26us  242.56us  256.74us  FW2(int*, unsigned long, int, unsigned long)
               0.01%  3.8759ms   663  5.8450us  5.6640us  7.8080us  FW1(int*, unsigned long, int)
==22536== Profiling result:
      Type Time(%)     Time    Calls      Avg      Min      Max   Name
GPU activities:  98.22%  55.2147s   663  83.280ms  81.926ms  88.125ms  FW3(int*, unsigned long, int, unsigned long, int, int)
               0.74%  416.86ms   333  1.2518ms  412.03us  272.31ms  [CUDA memcpy DtoH]
               0.74%  416.59ms   332  1.2548ms  417.83us  277.94ms  [CUDA memcpy HtoD]
               0.29%  162.67ms   663  245.36us  242.11us  256.71us  FW2(int*, unsigned long, int, unsigned long)
               0.01%  3.8629ms   663  5.8260us  5.6320us  6.9440us  FW1(int*, unsigned long, int)
API calls:    98.17%  28.8396s   665  43.368ms  434.21us  356.90ms  cudaMemcpy
               1.77%  519.81ms    1  519.81ms  519.81ms  519.81ms  cudaHostAlloc
               0.04%  12.233ms   1989  6.1500us  2.9900us  31.881us  cudaLaunch
               0.01%  2.2078ms    1  2.2078ms  2.2078ms  2.2078ms  cudaMemcpy
               0.00%  1.0732ms    2  536.60us  13.329us  1.0599ms  cudaFree
               0.00%  1.0433ms   8619   121ns   75ns  3.7630us  cudaSetupArgument
               0.00%  673.41us    94  7.1630us  122ns  331.94us  cuDeviceGetAttribute
               0.00%  482.19us   1989  242ns   105ns  1.8220us  cudaConfigureCall
               0.00%  155.78us    1  155.78us  155.78us  155.78us  cuDeviceTotalMem
               0.00%  42.445us    1  42.445us  42.445us  42.445us  cuDeviceGetName
               0.00%  9.7400us    1  9.7400us  9.7400us  9.7400us  cudaSetDevice
               0.00%  2.1160us    3   705ns   193ns  1.4650us  cuDeviceGetCount
               0.00%  1.9430us    2   971ns   204ns  1.7390us  cuDeviceGet
API calls:    98.16%  29.0999s   665  43.759ms  431.04us  357.02ms  cudaMemcpy
               1.78%  527.57ms    1  527.57ms  527.57ms  527.57ms  cudaHostAlloc
               0.04%  12.804ms   1989  6.4370us  3.0420us  41.523us  cudaLaunch
               0.01%  1.6530ms    1  1.6530ms  1.6530ms  1.6530ms  cudaMemcpy
               0.01%  1.6342ms    2  817.12us  11.449us  1.6228ms  cudaFree
               0.00%  1.0040ms    94  10.680us  103ns  538.67us  cuDeviceGetAttribute
               0.00%  983.16us   8619   114ns   76ns  2.4840us  cudaSetupArgument
               0.00%  495.35us   1989  249ns   102ns  2.0940us  cudaConfigureCall
               0.00%  119.46us    1  119.46us  119.46us  119.46us  cuDeviceTotalMem
               0.00%  44.191us    1  44.191us  44.191us  44.191us  cuDeviceGetName
               0.00%  9.2680us    1  9.2680us  9.2680us  9.2680us  cudaSetDevice
               0.00%  1.4690us    2   734ns   120ns  1.3490us  cuDeviceGet
               0.00%  1.4110us    3   470ns   136ns  1.0810us  cuDeviceGetCount
```

### III. Experiment & Analysis

#### A. Methodology

##### 1. System Spec

使用課堂所提供的環境。

```
Device 0: "GeForce GTX 1080"
  CUDA Driver Version / Runtime Version      9.0 / 9.0
  CUDA Capability Major/Minor version number: 6.1
  Total amount of global memory:             8114 MBytes (8507752448 bytes)
  (20) Multiprocessors, (128) CUDA Cores/MP: 2560 CUDA Cores
  GPU Max Clock rate:                      1835 MHz (1.84 GHz)
  Memory Clock rate:                       5005 MHz
  Memory Bus Width:                        256-bit
  L2 Cache Size:                           2097152 bytes
  Maximum Texture Dimension Size (x,y,z)    1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers 1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers 2D=(32768, 32768), 2048 layers
  Total amount of constant memory:           65536 bytes
  Total amount of shared memory per block:   49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                32
  Maximum number of threads per multiprocessor: 2048
  Maximum number of threads per block:        1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size (x,y,z):  (2147483647, 65535, 65535)
  Maximum memory pitch:                      2147483647 bytes
  Texture alignment:                         512 bytes
  Concurrent copy and kernel execution:      Yes with 2 copy engine(s)
  Run time limit on kernels:                 No
  Integrated GPU sharing Host Memory:        No
  Support host page-locked memory mapping:   Yes
  Alignment requirement for Surfaces:        Yes
  Device has ECC support:                    Disabled
  Device supports Unified Addressing (UVA): Yes
  Supports Cooperative Kernel Launch:        Yes
  Supports MultiDevice Co-op Kernel Launch: Yes
  Device PCI Domain ID / Bus ID / location ID: 0 / 1 / 0
  Compute Mode:
    < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 9.0, CUDA Runtime Version = 9.0, NumDevs = 1
Result = PASS
```

##### 2. Performance Metrics

在 Weak scalability 中，是使用自己產生的測資做實驗，Single-GPU 的測資中 Vertex 數量為 15000，而 Multi-GPU 則是產生 Vertex 數量為 15000x1.414 的測資。

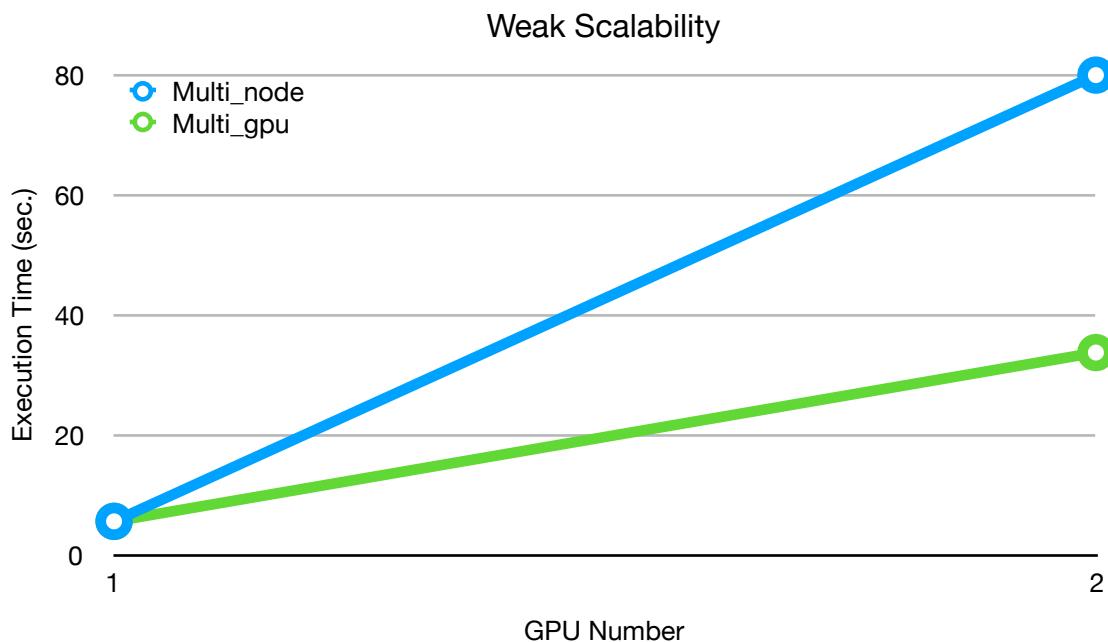
此外，在做 Time Distribution 的實驗則是使用助教提供的測資：3.in、4.in、5.in 以及自己產生的 Vertex 數量為 15000 的測資。

最後，實驗 Blocking factor 時，則是使用助教提供的 3.in 的測資，之所以用 3.in 這筆測資是因為希望可以用比較大的資料做實驗，但是當我在使用 4.in 這筆測資時，發現會因為時間上的限制導致程式無法順利執行完畢。

## B. Plots

### 1. Weak Scalability

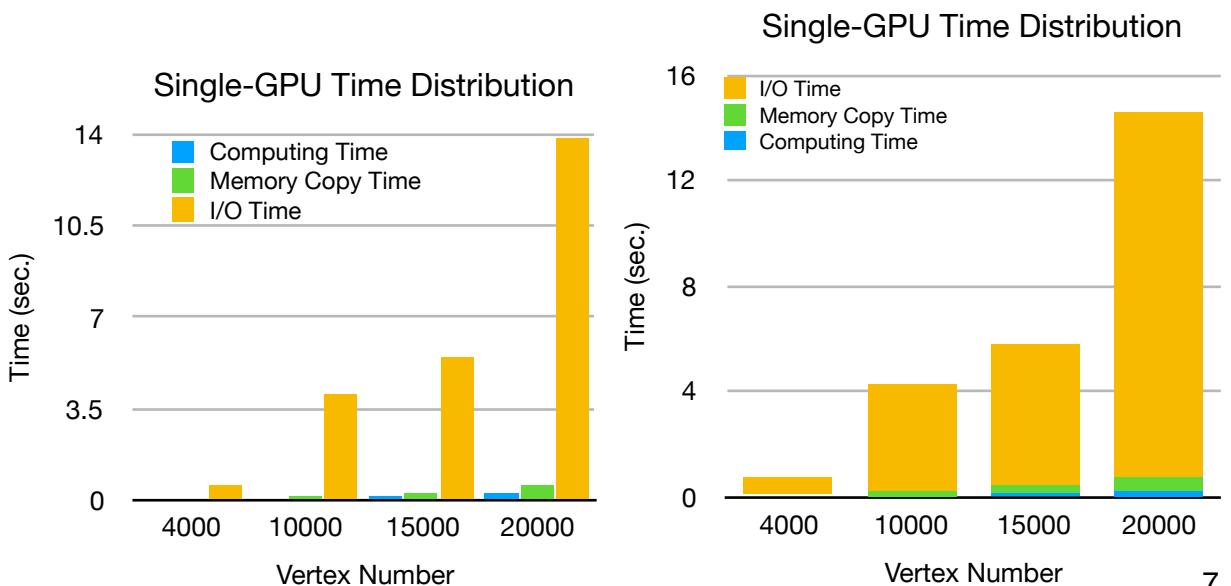
根據下圖實驗結果發現，使用 Multi\_node 的執行時間會比 Multi\_gpu 多出許多，我認為是因為在 Multi\_node 的版本中，每個 Iteration 開始的時候必須透過 MPI\_Send、MPI\_Recv 做資料的交換，最後也必須將 Rank 1 的資料回傳到 Rank 0 做檔案輸出；然而在 Multi\_gpu 的版本中，可以透過 DeviceToDevice 的方式傳遞資料，因此產生下圖的結果。



### 2. Time Distribution

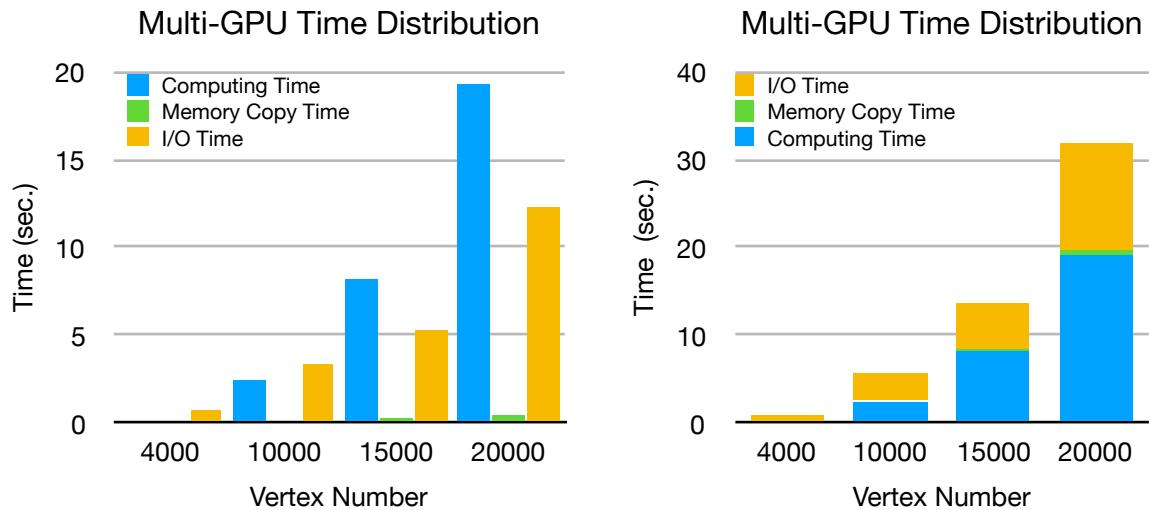
#### a) Single-GPU

由下圖所示，在 Single-GPU 的版本中，I/O 的時間佔了相當大的比例，而真正正在計算的時間以及 Memory copy 時間相對來說非常少，這個現象我認為是因為寫檔的只有一個 Process，而且檔案的需要輸出的資料量又很龐大，所以可能是正常的。



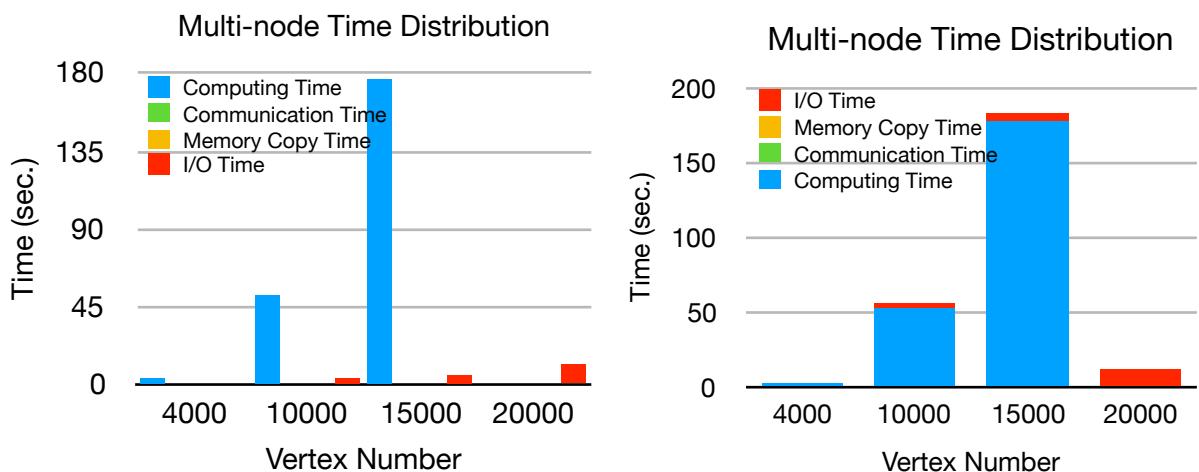
b) Multi-GPU implementation in the single node

在 Multi-GPU 的版本中，計算時間在 Vertex 數量增加會急遽的上升，因為需要計算的是任意兩個 Vertex 之間的路徑，因此時間成長會是 Vertex 數量的平方的曲線，而圖中確實也呈現預期之中的結果； I/O 的時間依舊佔了大部分的比例，也如預期中的隨著Vertex 數量的平方的曲線向上成長，Memory copy 時間是相對最少的。



c) Multi-GPU implementation with MPI

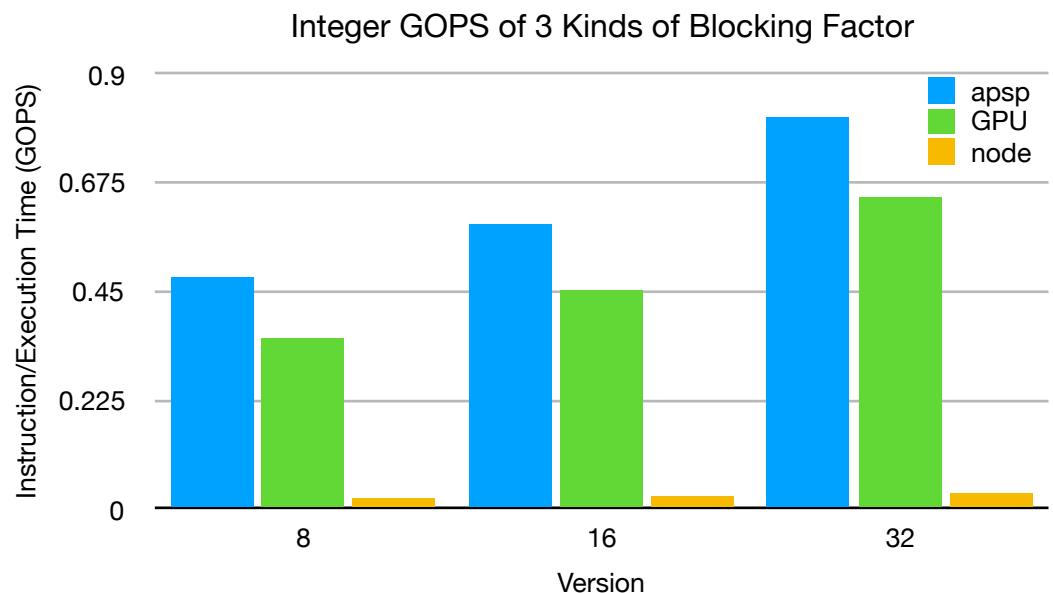
在 Multi-node 的版本中，計算時間如預期中地隨著 Vertex 數量的平方的曲線向上成長，然而實驗結果出現一個令人匪夷所思的現象，在 Vertex 的數量為 20000 時，計算時間為 0.0 秒，理論上不應該出現這樣的結果，而且在實驗的過程中，相同的 Vertex 數量所測試的 Communication 時間會有很大的起伏，因此我認為這是機器不穩定造成的，而 I/O 時間也意外的少了很多，Memory copy 時間依舊是相對最少的。



### 3. Blocking Factor

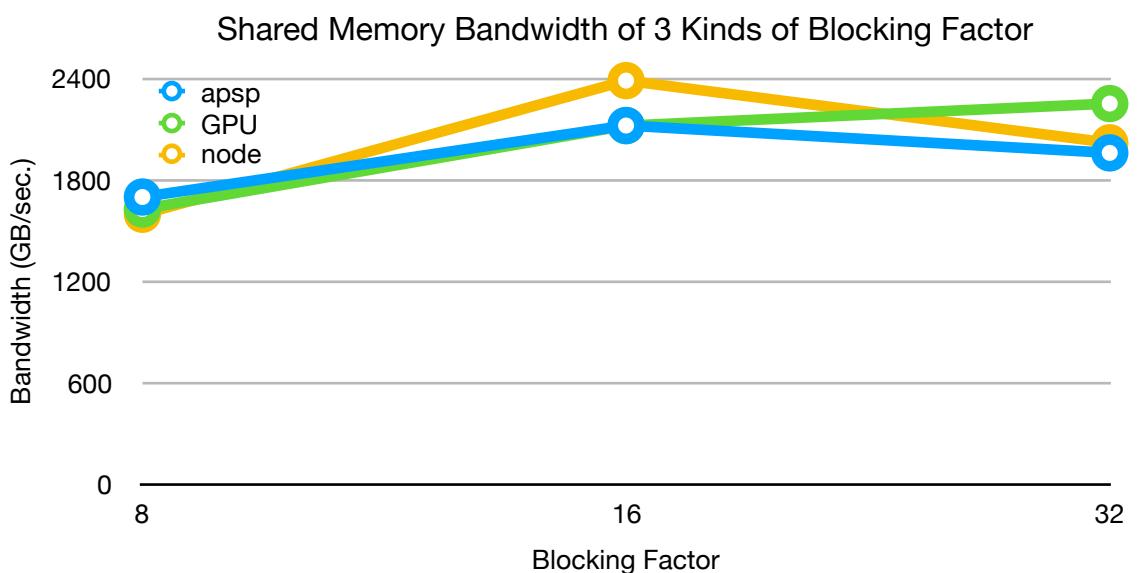
#### a) Integer GOPS

由下圖所示，Single-GPU 版本中每秒所執行的 Integer instruction 數量是最多的，因為 Multi\_gpu 版本中會將指令分配給兩張 GPU 分別執行，所以執行的指令會相對較少，但在 Multi\_node 中指令的執行數量相對少很多，我認為這是不正常的結果。此外，每個版本中不同的 Blocking Factor 會有不同的結果，在 Block 數量被分配為 1024 個 Thread 時，每秒會有最多的指令被執行，這個和預期中的結果相符。



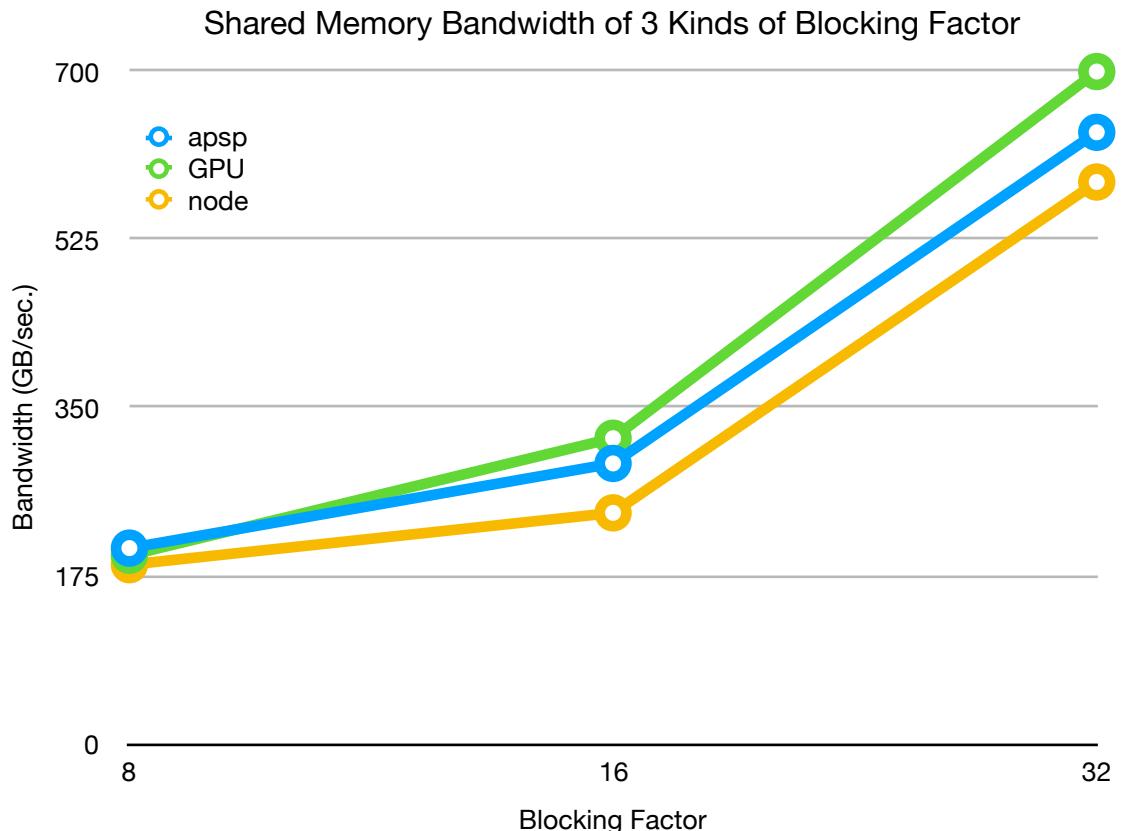
#### b) Shared Memory Bandwidth

由下圖所示，Shared memory 的 bandwidth 在 Block 中 Thread 數量為 8x8 的時候明顯較低，因為 Launch kernel 所要求的 Thread 數量較少，所以複製到 Shared memory 的資料會比較少，在 Block 中 Thread 數量為 16x16 有上升，然而在 Block 中 Thread 數量為 32x32 時 Bandwidth 會下降。



### c) Device Memory Bandwidth

由下圖所示，Device memory 的 bandwidth 在 Block 中 Thread 數量為 32x32 的時候會有比較大的 Bandwidth，我認為 Bandwidth 應該不會受到 Blocking Factor 的影響而產生變化，因為在 Kernel launch 之前就在做 Memory copy，因此有這樣的結果令人費解。



## 4. Optimization

### a) Shared memory

如 I-A-2 中所敘述，在 phase 1 的時候，我將需要被運算的  $\text{Block}(r, r)$  從 Global memory 搬到 Shared memory 中以加速運算，然而在 Kernel launch 時所要求的 Block 中的每一個 Thread 剛好被分到需要被運算的  $\text{Block}(r, r)$  中的一個元素，因此一個 Thread 根據自己的 ID 將資料搬到 Shared memory 中，也有加速的效果。在 phase 2、3 的時候，因為需要搬的資料量增加，所以必須透過 Thread ID, Block ID 進行座標轉換，讓一個 Thread 只需要一次搬 2、3 個資料到 Shared memory 上即可完成分工。

b) Occupancy optimization

如同 I-A-1 中所描述的，在開始計算之前，必須決定一個 Block 的大小，我根據 GPU 中一個 Block 最多可使用的 Thread 數量 ( $T$ ) 為依據，將一個 Block 設定為  $\lceil \sqrt{T} \rceil \times \lceil \sqrt{T} \rceil$ ，若  $V$  不為  $\lceil \sqrt{T} \rceil$  的倍數，則將  $V$  做 Padding，使得  $V + \lceil \sqrt{T} \rceil > V_{padding} \geq V$ ， $V_{padding} = \lceil \sqrt{T} \rceil \times B$  ( $B$  為一個 column 中的 Block 數量)。

c) Streaming

因為每個 Iteration 之間存在 Data dependency，所以無法透過 Streaming 進行優化。

d) Unroll

同 Streaming，因為每個 Iteration 之間存在 Data dependency，所以無法透過 Streaming 進行優化。

e) Reduce communication

在每一個 Iteration 開始前，只有將有 Data dependency 的資料傳給另一個 GPU，因此在這個部分我是有實作初步的優化。

#### IV. Experience / Conclusion

這次作業在概念上不會很困難，除了座標的換算稍微多了一些，如果沒有想清楚就寫的話，會造成在 Debug 的麻煩，在實作的同時我也學到很多，怎麼分配資源、切割資料會是很重要的一部份。除此之外，這次作業的時間只有兩個禮拜，同時又有其他的作業同時在進行，時間真的不夠用，希望還是可以給滿三周的時間。