CS542200 Parallel Programming

Homework 4: Blocked All-Pairs Shortest Path

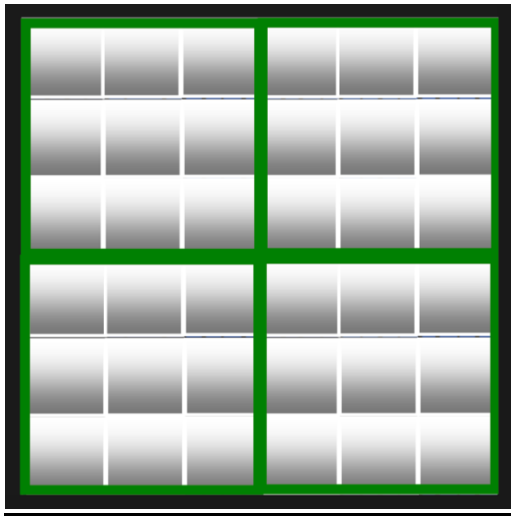
107062558 賴怡惠

Implementation

Apsp

以下圖為例，解釋切割資料。

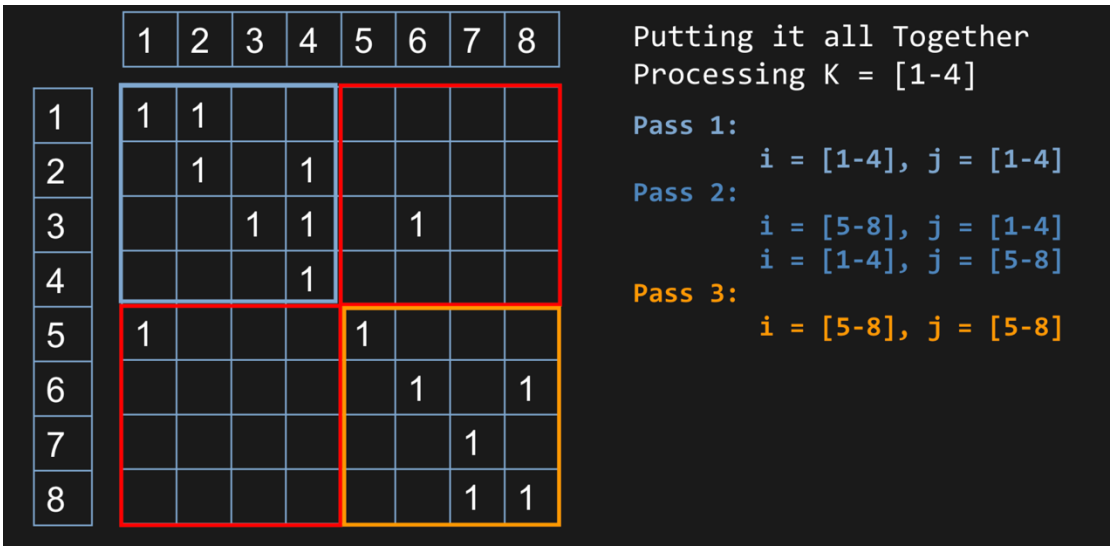
若目前有 6 個 vertices，Blocking Factor B=3，切成四個 block。



實際狀況，為了實驗方便，B 的值可以由 command line 輸入，通常是設為 #vertices，但若#vertices 大於 32，即超過一個 warp 的負荷量，仍把 B 設為 32。

```
int B = argc>3? atoi(argv[3]) : n>32? 32 : n;
```

每個 iteration 都會經過 3 個 phase，分別由藍色(pivot 所在的 block)、紅色(和 pivot 共用行或列的 block)、橘色(其他狀況)如圖所示。



Phase1: update pivot 所在的 block 最短路徑

Setp1: 把 pivot 所在 block 由 global memory 搬進 shared memory，因為該 block 在找最短路徑時，需要在同個 block 已經 update 過的值，以上圖藍色區域為例，(2,2)在 update 找最小路徑時，需要 w(2,1)+w(1,2) ，故 block 內部有 dependency，所以需要時常 access，故放在 shared memory 減少 io time。

Step2: access shared memory，執行 floyd 找最小路徑，並且 update shared memory，由 step1 可知，phase1 要 update 的 blocks 內部都有 dependency。

Step3: update 到 global memory

Phase2: update 和 pivot 所在 block 同 row、column 的 blocks 最短路徑

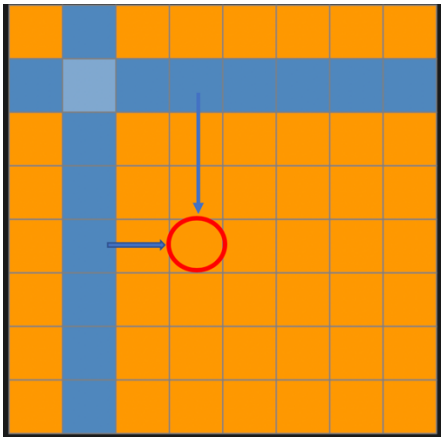
Setp1: 把 pivot 所在 block 和自己所在 block 由 global memory 搬進 shared memory，因為每個和 pivot 所在 block 同 row 或 col 的 block，update 時，都需要 pivot 的值和自己的值，需要時常 access，故放在 shared memory 減少 io time。

Step2: access shared memory，執行 floyd 找最小路徑，並且 update shared memory，由 step1 可知，phase2 要 update 的 blocks 內部都有 dependency。

Step3: update 到 global memory

Phase3: update 剩餘 blocks 的最短路徑

Setp1: 把和 pivot 同 row、col 且和自己所在 block 同 row、col 的兩個 block 由 global memory 搬進 shared memory。因為每個 phase3 要 update 的 block，都是由 phase2 update 過的 block，故把用到的兩個先放進 shared memory 有助於減少 io time。

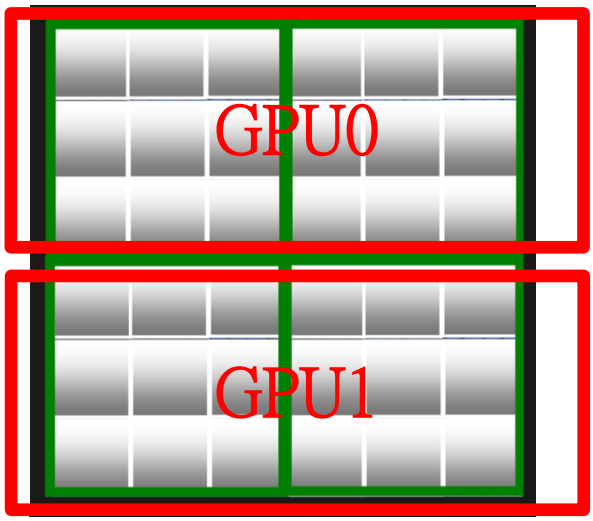


Step2: access shared memory，執行 floyd 找最小路徑，不需直接 update 到 shared memory 是因為，同個 block 內部沒有 dependency！此 phase 的 update 單純與和 pivot 同 row、col 且和自己所在 block 同 row、col 的兩個 block 還有自己原來的值有關。

Step3: update 到 global memory

Multi_gpu

以下圖為例，解釋切割資料，若目前有 6 個 vertices，Blocking Factor B=3，切成四個 block。



實際狀況，為了實驗方便，B 的值可以由 command line 輸入，通常是設為 #vertices，但若#vertices 大於 32，即超過一個 warp 的負荷量，仍把 B 設為 32。

```
int B = argc>3? atoi(argv[3]) : n>32? 32 : n;
```

因為目前有兩張在同一台機器的卡，所以我們可以把整個 Dist array 利用 row，等分兩半，上半部分給 GPU0，下半部分給 GPU1。

若 Dist 無法完美對分，則把多餘的一個(因為目前只有 2 個 gpu)row block 給 GPU0。

```
block_row = (num_omp_threads < num_blocks%num_gpus)? avg_block+1:avg_block;
```

在每個 iteration 計算時，經過三個 phase update 後(同 apsp 版本)，pivot 所在的 gpu 卡需要把 pivot 所在的整條 row block 傳給另一張卡！此外，因為 multi_gpu 只有一個 node，row block 只需要 cudaMemcpy 用 devicetodevice mode 即可！column block 不需要傳遞，是因為每張卡在 update block 時，只需要 pivot 同 column 在同張卡的部分而已！

```
cudaMemcpy(Dist + start_round*n*B, device_Dist[num_omp_threads] + start_round*n*B, size,
cudaMemcpyDeviceToHost);
```

在 iteration 結束後，兩張卡只需要把自己 update 的半部的 Dist，且因兩張卡在同個 node，所以 device memory 只需把該半部傳回 host，host 即會合併最終結果！

Multi_node

以下圖為例解釋切割資料，若目前有 6 個 vertices，Blocking Factor B=3，切成四個 block。

實際狀況，為了實驗方便，B 的值可以由 command line 輸入，通常是設為 #vertices，但若#vertices 大於 32，即超過一個 warp 的負荷量，仍把 B 設為 32。

```
int B = argc>3? atoi(argv[3]) : n>32? 32 : n;
```

因為目前有兩張在各在一台機器的卡，也就是說，一台機器有一張卡，目前有兩張卡所以我們有兩台機器。

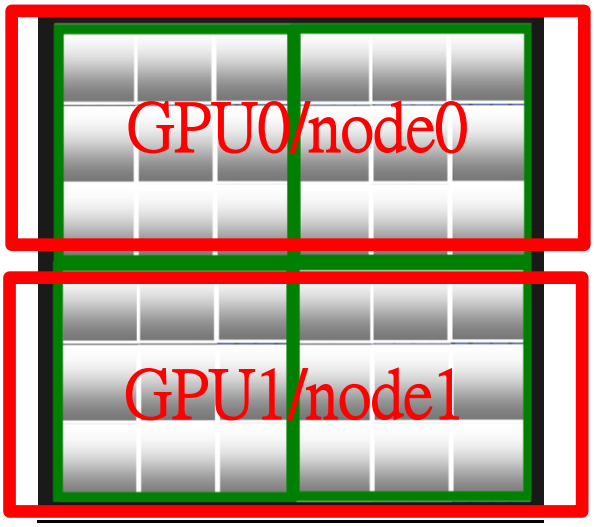
我們把整張 Dist 分成兩半，上半部分給 GPU0，下半部分給 GPU1，若 Dist 無法完美對分，則把多餘的一個 row block 全部給 GPU0。

```
int block_row = (rank < num_blocks%num_procs)? avg_block+1 : avg_block;
```

在每個 iteration 計算時，經過三個 phase update 後(同 apsp 版本)，pivot 所在的 gpu 卡需要把 pivot 所在的整條 row block 傳給另一張卡！但因為兩張卡在不同的機器上，所以溝通時需要先把 row block 回傳給 host memory 再由 host memory 利用 send,recv 傳給另一台機器。

```
        cudaMemcpy(Dist + start_round*n*B, device_Dist + start_round*n*B, size,
cudaMemcpyDeviceToHost);
        MPI_Send(buff, shared_mem_sizes[1], MPI_INT, 0, 0, MPI_COMM_WORLD);
        MPI_Recv(Dist+buff[0], buff[1], MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

在 iteration 結束後，我們在 rank=0 的 host memory 做合併！所以先把各自維護的半部傳給自己的 host memory，再透過 MPI 做跨 node 的溝通！



Profiling Results

apsp

```
[rhyhorn@hades02 hw4]$ nvprof ./apsp testcases/10.in 10.ans
==4946== NVPROF is profiling process 4946, command: ./apsp testcases/10.in 10.ans
==4946== Profiling application: ./apsp testcases/10.in 10.ans
==4946== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   52.80%  2.8463ms        32  88.946us  88.321us  89.665us  phase_three(int, int, int, int*)
                   20.70%  1.1160ms         1  1.1160ms  1.1160ms  1.1160ms  [CUDA memcpy DtoH]
                   12.48%  672.84us         1  672.84us  672.84us  672.84us  [CUDA memcpy HtoD]
                   10.47%  564.13us        32  17.629us  13.248us  18.400us  phase_two(int, int, int, int*)
                    3.55%  191.24us        32  5.9760us  5.8240us  6.4010us  phase_one(int, int, int, int*)
      API calls:   97.12%  317.62ms         1  317.62ms  317.62ms  317.62ms  cudaMalloc
                    1.62%  5.3066ms         2  2.6533ms  599.49us  4.7071ms  cudaMemcpy
                    0.72%  2.3388ms         1  2.3388ms  2.3388ms  2.3388ms  cuDeviceGetName
                    0.25%  801.77us        94  8.5290us     295ns  359.03us  cuDeviceGetAttribute
                    0.14%  467.54us        96  4.8700us  4.0890us  34.788us  cudaLaunch
                    0.09%  278.11us         1  278.11us  278.11us  278.11us  cuDeviceTotalMem
                    0.04%  126.37us         1  126.37us  126.37us  126.37us  cudaFree
                    0.02%  53.385us       384     139ns     102ns  4.3750us  cudaSetupArgument
                    0.01%  16.560us         1  16.560us  16.560us  16.560us  cudaSetDevice
                    0.00%  15.987us        96     166ns     150ns  1.0280us  cudaConfigureCall
                    0.00%  5.2810us         3  1.7600us     375ns  4.3830us  cuDeviceGetCount
                    0.00%  1.5480us         2     774ns     303ns  1.2450us  cuDeviceGet
```

Multi_gpu

```
[rhyhorn@hades02 hw4]$ nvprof ./multi_gpu testcases/10.in 10.ans
==4975== NVPROF is profiling process 4975, command: ./multi_gpu testcases/10.in 10.ans
==4975== Profiling application: ./multi_gpu testcases/10.in 10.ans
==4975== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   56.53%  2.8864ms        32  90.200us  89.601us  91.328us  phase_three(int, int, int, int*, int)
                   14.41%  735.78us         1  735.78us  735.78us  735.78us  [CUDA memcpy DtoH]
                   14.25%  727.75us         1  727.75us  727.75us  727.75us  [CUDA memcpy HtoD]
                   11.06%  564.74us        32  17.648us  13.248us  18.304us  phase_two(int, int, int, int*)
                    3.75%  191.23us        32  5.9760us  5.8250us  6.4000us  phase_one(int, int, int, int*)
      API calls:   97.94%  324.83ms         1  324.83ms  324.83ms  324.83ms  cudaMalloc
                    1.21%  4.0114ms         2  2.0057ms  820.58us  3.1908ms  cudaMemcpy
                    0.37%  1.2179ms        96  12.686us  11.320us  66.239us  cudaLaunch
                    0.24%  799.92us        94  8.5090us     300ns  363.06us  cuDeviceGetAttribute
                    0.08%  272.24us         1  272.24us  272.24us  272.24us  cuDeviceTotalMem
                    0.07%  221.39us         2  110.70us  4.6090us  216.78us  cudaFree
                    0.05%  158.97us       416     382ns     306ns  6.6900us  cudaSetupArgument
                    0.02%  66.896us         1  66.896us  66.896us  66.896us  cuDeviceGetName
                    0.01%  46.971us        96     489ns     446ns  1.8700us  cudaConfigureCall
                    0.00%  12.348us         1  12.348us  12.348us  12.348us  cudaSetDevice
                    0.00%  4.6900us         3  1.5630us     300ns  3.8920us  cuDeviceGetCount
                    0.00%  3.4780us         1  3.4780us  3.4780us  3.4780us  cudaGetDeviceCount
                    0.00%  1.4540us         2     727ns     392ns  1.0620us  cuDeviceGet
```

Multi_node

```
[rhyhorn@hades02 hw4]$ srun -n2 -p pp --gres=gpu:1 nvprof ./multi_node testcases/10.in 10.ans
==10868== NVPROF is profiling process 10868, command: ./multi_node testcases/10.in 10.ans
==10866== NVPROF is profiling process 10866, command: ./multi_node testcases/10.in 10.ans
==10868== Profiling application: ./multi_node testcases/10.in 10.ans
==10866== Profiling application: ./multi_node testcases/10.in 10.ans
==10868== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   37.35%  1.3360ms        32  41.749us  40.289us  43.169us  phase_three(int, int, int, int*, int)
                   26.34%  942.44us        17  55.437us  3.2640us  625.64us  [CUDA memcpy HtoD]
                   16.94%  606.12us        16  37.882us  19.457us  311.91us  [CUDA memcpy DtoH]
                   14.24%  509.51us        32  15.922us  12.320us  16.416us  phase_two(int, int, int, int*)
                    5.12%  183.33us        32  5.7290us  5.4080us  12.224us  phase_one(int, int, int, int*)
==10866== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   36.47%  1.2884ms        32  40.262us  38.657us  42.049us  phase_three(int, int, int, int*, int)
                   27.03%  954.80us        16  59.674us  20.576us  643.69us  [CUDA memcpy HtoD]
                   16.81%  593.70us        17  34.923us  4.3520us  296.55us  [CUDA memcpy DtoH]
                   14.56%  514.53us        32  16.079us  11.648us  16.448us  phase_two(int, int, int, int*)
      API calls:   97.68%  249.21ms         1  249.21ms  249.21ms  249.21ms  cudaMalloc
                    1.52%  3.8662ms        33  117.16us  9.6790us  734.94us  cudaMemcpy
                    5.13%  181.31us        32  5.6660us  5.0240us  9.6000us  phase_one(int, int, int, int*)
                    0.28%  705.04us        94  7.5000us     111ns  297.45us  cuDeviceGetAttribute
                    0.20%  513.51us        96  5.3490us  3.7130us  29.455us  cudaLaunch
                    0.16%  401.01us         1  401.01us  401.01us  401.01us  cuDeviceGetName
                    0.07%  183.07us         1  183.07us  183.07us  183.07us  cudaFree
                    0.06%  143.25us         1  143.25us  143.25us  143.25us  cuDeviceTotalMem
                    0.03%  67.724us       416     162ns      92ns  14.405us  cudaSetupArgument
                    0.01%  21.704us         1  21.704us  21.704us  21.704us  cudaSetDevice
                    0.01%  17.437us        96     181ns     116ns     908ns  cudaConfigureCall
                    0.00%  3.0290us         3  1.0090us     194ns  2.6370us  cuDeviceGetCount
                    0.00%     486ns         2     243ns     114ns     372ns  cuDeviceGet
      API calls:   97.28%  250.69ms         1  250.69ms  250.69ms  250.69ms  cudaMalloc
                    1.74%  4.4755ms        33  135.62us  13.290us  1.0206ms  cudaMemcpy
                    0.52%  1.3293ms        94  14.141us     108ns  480.10us  cuDeviceGetAttribute
                    0.18%  475.57us         1  475.57us  475.57us  475.57us  cuDeviceTotalMem
                    0.18%  463.90us        96  4.8320us  3.4830us  21.083us  cudaLaunch
                    0.05%  121.26us         1  121.26us  121.26us  121.26us  cudaFree
                    0.02%  61.804us         1  61.804us  61.804us  61.804us  cuDeviceGetName
                    0.02%  51.052us       416     122ns      90ns  3.2620us  cudaSetupArgument
                    0.01%  17.163us        96     178ns     112ns     647ns  cudaConfigureCall
                    0.00%  4.8540us         1  4.8540us  4.8540us  4.8540us  cudaSetDevice
                    0.00%  2.1950us         3     731ns     169ns  1.7660us  cuDeviceGetCount
                    0.00%     630ns         2     315ns     170ns     460ns  cuDeviceGet
```

Experiment&Analysis

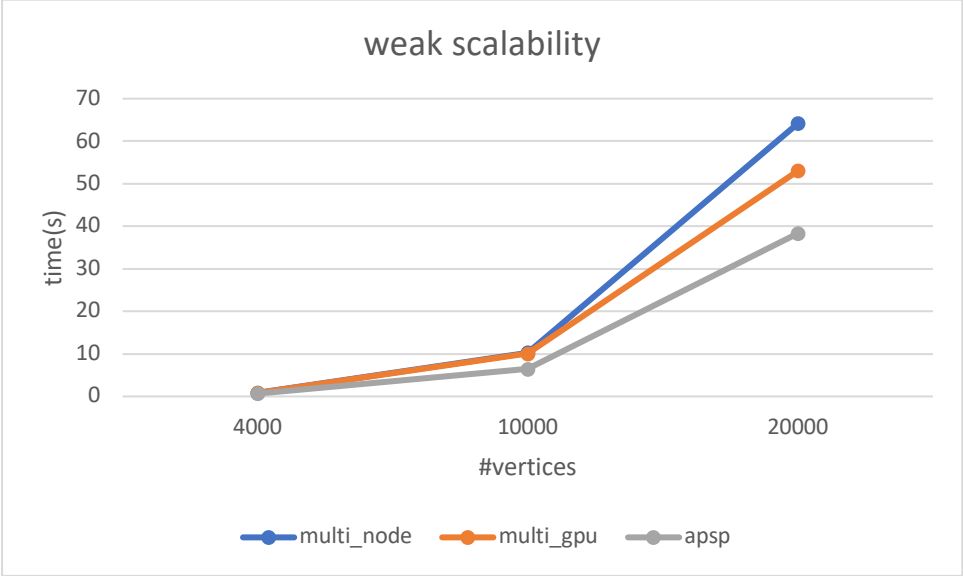
1. System Spec

```
[rhyhorn@hades02 hw4]$ lscpu
架構：                x86_64
CPU 作業模式：      32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                12
On-line CPU(s) list:   0-11
每核心執行緒數：2
每通訊端核心數：6
Socket(s):             1
NUMA 節點：          1
供應商識別號：   GenuineIntel
CPU 家族：           6
型號：               79
Model name:            Intel(R) Core(TM) i7-6850K CPU @ 3.60GHz
製程：               1
CPU MHz：              1598.484
BogoMIPS:              7200.00
虛擬：               VT-x
L1d 快取：           32K
L1i 快取：           32K
L2 快取：            256K
L3 快取：            15360K
NUMA node0 CPU(s)：    0-11
```

```
[rhyhorn@hades02 hw4]$ nvidia-smi
Wed Dec 19 16:23:12 2018
+-----------------------------------------------------------------------------+
| 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 |
| 42%   62C    P2   115W / 216W |   5724MiB /  8114MiB |    100%      Default |
+-------------------------------+----------------------+----------------------+
|   1  GeForce GTX 1080    On   | 00000000:4D:00.0 Off |                  N/A |
|  5%   48C    P8    14W / 216W |     11MiB /  8113MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0      6140      C   python3                                     2605MiB |
|    0      6496      C   python3                                     2605MiB |
|    0      6821      C   ./apsp                                       501MiB |
```

2. Weak Scalability & Time Distribution

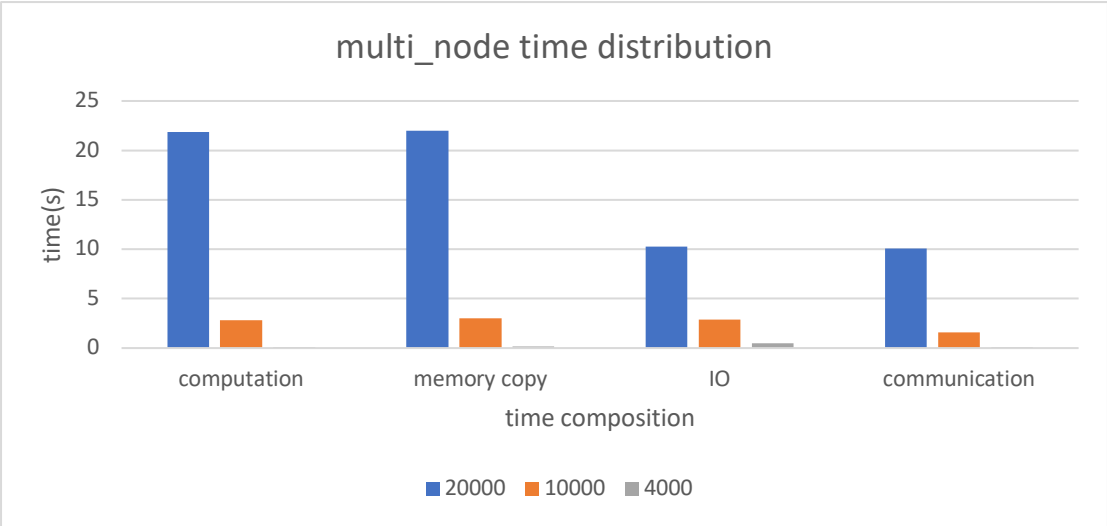


由 # vertices = 20000 的測資搭配以下的 time distibution 可猜測 communication 是主要拉慢效能的部分，而 multi_gpu 因為做太頻繁的 memory copy 導致它雖然有兩張卡跑，但是跑得仍然比 apsp 慢！

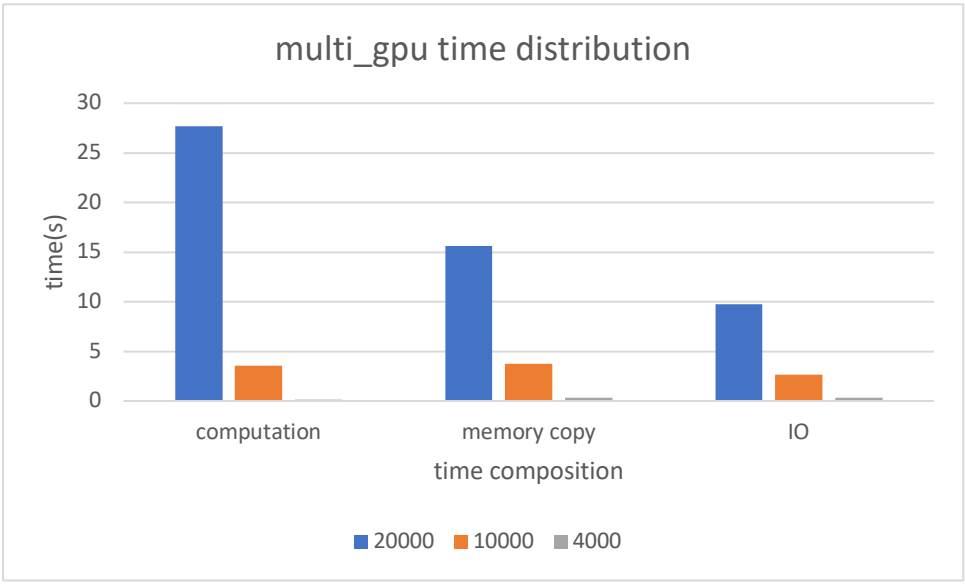
Execution time(s)\ #vertices	4000	10000	20000
multi_node	0.847864	10.27	64.163
multi_gpu	0.83	10.025	53.01
apsp	0.667	6.476	38.322

How to measure?

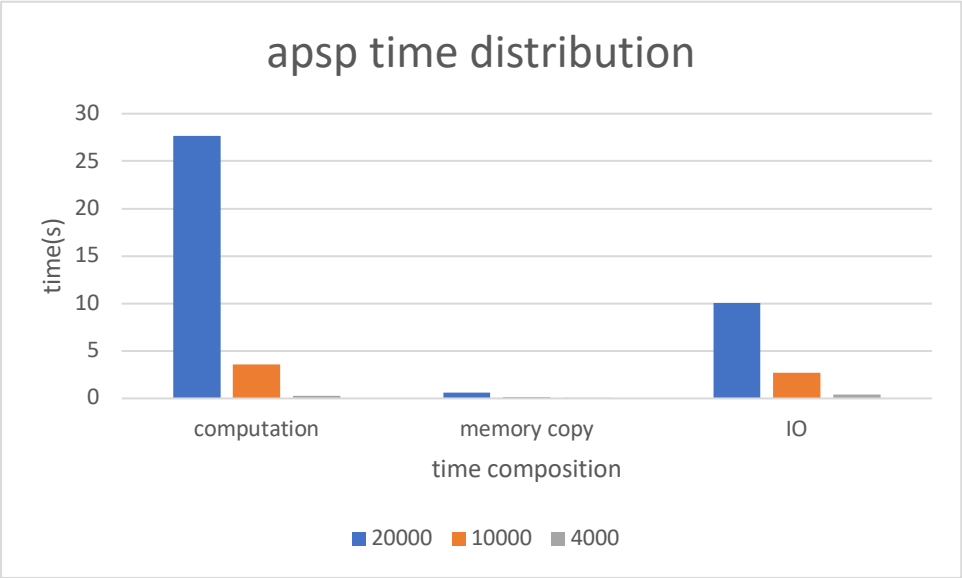
1. Computation, memory copy 可從 nvprof 得知，主要由 phase3 和 memcpy 貢獻
2. Io 可利用 clock()包住 input, output file 的部分計算
3. Communication 使用 MPI_Wtime()包住 send, recv 的部分



# vertices\time category	Computation(s)	memory copy(s)	IO(s)	Communication (s)
20000	21.86	22	10.24	10.063
10000	2.79	3	2.861727	**1.61971**
4000	0.089	0.173	0.47964	0.106224



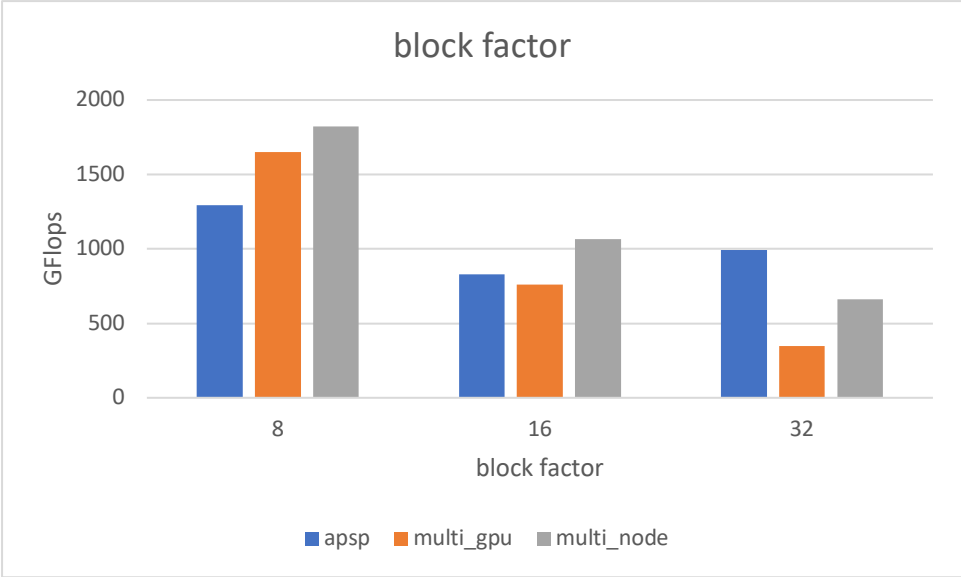
#vertices\time category	Computation(s)	memory copy(s)	IO(s)
20000	27.66	15.63	9.72
10000	3.59	3.765	2.67
4000	0.17	0.32	0.34



#vertices\time category	Computation(s)	memory copy(s)	IO(s)
20000	27.64	0.642	10.04
10000	3.586	0.16	2.73
4000	0.259	0.028	0.38

綜合三種狀況的 computation time 可發現，雖然一張卡的 computation time 並不會是兩張卡 computation time 的兩倍，我猜可能是我在兩張卡的情況，gpu 內部寫太多 if-else 拖慢 computation time。

4. Blocking Factor



GFlops\block factor	8	16	32
apsp	1294.28933	827.719237	993.791336
multi_gpu	1648.93044	761.924055	349.05305
multi_node	1820.90683	1063.65796	661.469491

一開始我以為 block factor 只要在 32 內也就是小於 warp 數量，performance 應該要變好，但我認為 performance 仍然不佳的原因是跑的時間進步不顯著，可能是因為太頻繁的 memory copy, communication 和 cuda code 放太多 if-else 導致在資料量大時，這些拉慢效能的因素被放大了！所以 block factor 越大 GFlops 反而變小。

5. Optimization

◎ Shared memory

Phase 1

把 pivot 所在 block 由 global memory 搬進 shared memory，因為該 block 在找最短路徑時，需要在同個 block 已經 update 過的值，以上圖藍色區域為例，(2,2)在 update 找最小路徑時，需要 w(2,1)+w(1,2) ，故 block 內部有 dependency，所以需要時常 access，故放在 shared memory 減少 io time。

Phase2

把 pivot 所在 block 和自己所在 block 由 global memory 搬進 shared memory，因為每個和 pivot 所在 block 同 row 或 col 的 block，update 時，都需要 pivot 的值和自己的值，需要時常 access，故放在 shared memory 減少 io time。

◎ Unroll

平行 Floyd algo 的 k

```
#pragma unroll
    for(int k=0; k<B; ++k){
        if(target > shared_col[y*B + k] + shared_row[k*B + x]){
            target = shared_col[y*B + k] + shared_row[k*B + x];
        }
    }
```

從 global memory 搬資料到 shared memory

```
#pragma unroll
            for(int k=0; k<B; ++k){
                  if(shared_Dist[y*B + x] > shared_pivot[y*B + k] + shared_Dist[k*B + x]){
                        shared_Dist[y*B + x] = shared_pivot[y*B + k] + shared_Dist[k*B + x];
                  }
                  __syncthreads();
            }
```