Student ID: 101062319
Name: 巫承威

Homework 4 - Report

• Implementation

(a) How do you divide your data?

i. Single

In my single GPU version, I adopt the template of the sequential Blocked Floyd-Warshall version, so the framework of my program is similar with the "block_FW.cpp". Before the "block-_FW()" function, the implementation is almost the same. According to my implementation, I divide the whole n*n matrix into a block matrix which is a $\lceil \frac{N}{B} \rceil * \lceil \frac{N}{B} \rceil$ matrix and each block has B*B elements. In phase 1, I compute the primary block by a GPU block which has B*B threads, so each element will be assigned to a thread to be computed. In phase 2, I compute the blocks which has the same row index and same column index of the primary block, so there will be ($\lceil \frac{N}{B} \rceil$ * 2 - 2) blocks to be computed by the GPU. Thus, I assign the task to total $\lceil \frac{N}{B} \rceil$ * 2 grids and each grid has B*B threads. It's the same with the phase 1, each element will be assigned to the specific thread. In phase 3, I create $\lceil \frac{N}{B} \rceil * \lceil \frac{N}{B} \rceil$ grids and each grid also has B*B threads. And the following processing steps are as the as the phase 1 and phase 2. What's more, I use "shared memory" in each block to accelerate the computing speed. After $\lceil \frac{N}{B} \rceil$ rounds, all the elements will be computed and the computing part is over.

ii. OpenMP

According to my OpenMP version, I create total two CPU threads to control two GPUs which means that each CPU Host thread control one GPU. The program template is almost the same with the single-GPU version. Since I think to partition a block into two parts to be assigned to different GPU is lack of effectiveness, so I only partition phase 2 and phase 3 to different GPUs. First, I let one of the GPU to compute all the primary block(phase 1) in all rounds. Second, in phase 2, I divide the $\lceil \frac{N}{B} \rceil$ * 2 blocks to two GPUs, so each GPU will compute $\lceil \frac{N}{B} \rceil$ * 1 blocks which represent that the blocks which have same row index of the primary block will be assigned to GPU_0, and the other blocks which have same column index of the primary block will be assigned to GPU_1. Via this assignment, it may save some time and reach the goal of parallelism. Finally, in phase 3, I partition the whole blocks into two parts: "Top blocks" and "Bottom blocks" which symbolizes the blocks whose row index is smaller than primary blocks and the blocks whose row index is bigger than primary block respectively. And the assignment is the same with the phase 2, each

part of blocks will be send to different GPU to satisfy multi-GPU computation. What's more, I use "pinned memory" to be the buffer sent and received between two different GPUs.

iii. MPI

In MPI version, the way to partition the data is the same with the OpenMP version such like divide the phase 2 into row block and column blocks and divide the phase 3 into top blocks and bottom blocks. I think the only difference between MPI and OpenMP is the method to communication with two GPUs.

(b) How do you implement the communication? (in multi-GPU version)

i. OpenMP

First, I allocate the "pinned memory" in the CPU Host, and take advantage of the features of OpenMP (shared memory) and the architecture of Unified Virtual Address (UVA). So I only need to create the pinned memory to be the array only once. Then after finishing computing the results in each round and each phase, both two GPUs will update the data in pinned memory directly. However, there exists the devil, after updating the results in pinned memory, it need to call the function "cudaDeviceSynchronize()" to synchronize the difference of computing speed between two GPUs; otherwise, the answer will be wrong.

ii. MPI

It's more complex than the OpenMP version. I also create two processes to control two GPUs and let the master process do more than slave process. From master process's point of view, it need to do the following assignments step by step:

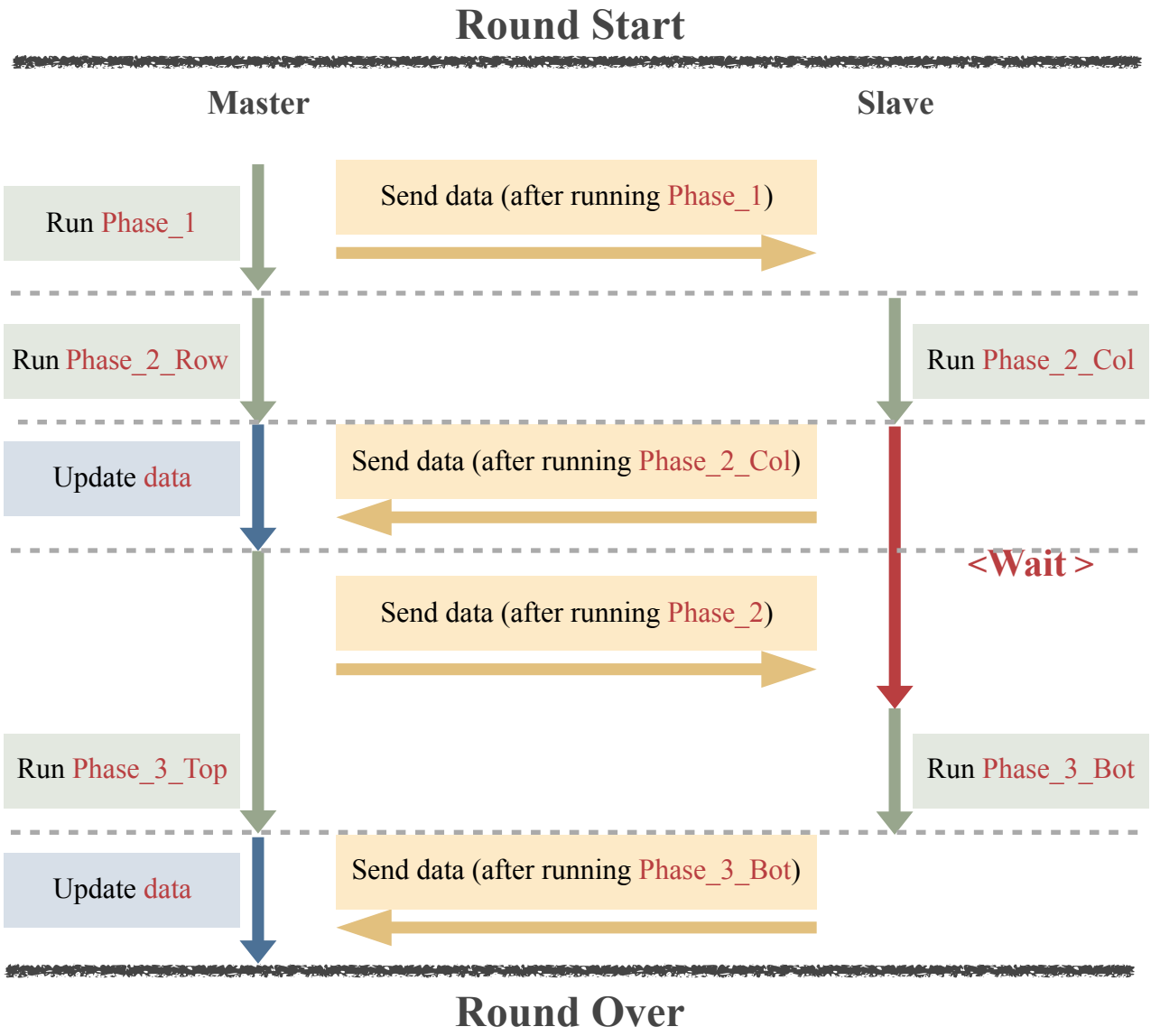
- compute all phase 1 tasks
- send the results to slave
- compute the row blocks in phase 2
- receive the results from slave (column blocks computed in phase 2)
- update the whole data
- send the results to slave (the newest data)
- compute top blocks
- receive the results from slave (bottom blocks computed in phase 3)
- update the whole data

And the slave process will do less tasks as following:

- receive the results from master (primary blocks computed in phase 1)
- compute the column blocks in phase 2
- send the results to master (column blocks are computed)

- receive the results from master (phase 2 are computed)
- compute the bottom blocks in phase 3
- send the results to master (bottom blocks are computed)

As the above tasks showing, the master process do more task and it is responsible to merge the computed results. The following executing flow diagram will show the above tasks more clearly:



(c) What 's your configuration? (e.g., blocking factor, #blocks, #threads)

According to my implementation, I need to set B to be the factor of the "N" which means that the blocking factor must be the factor of the # of the nodes(vertices). Also, due to the properties of the GPU(K20M and M2090), the blocking factor must smaller than 32, so it can satisfy the restriction of the fact that each blocks can only has 1024 threads (1024 = 32 * 32).

And the # of blocks is based on the # of the nodes(vertices) and the size of blocking factor. Similar with the # of blocks, the # of the threads depends on the blocking factor and it equals to blocking factor's square.

- Profiling Results

The configuration of the profiling results:

Blocking Factor (B): 20

Test Case: Sample Test Case 3 (1000 nodes, 5000 edges)

Sample Test Case 5 (6000 nodes, 40000 edges)

(a) Single-GPU

- Sample Test Case 3 (1000 nodes, 5000 edges)

```
[user10@gpucluster0 hw4]$ nvprof ./HW4_101062319_cuda.exe in3 cudaout3 20
==5238== NVPROF is profiling process 5238, command: ./HW4_101062319_cuda.exe in3 cudaout3 20
GPU Compute Time: 92.114304
==5238== Profiling application: ./HW4_101062319_cuda.exe in3 cudaout3 20
==5238== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 90.94%  69.886ms        50  1.3977ms  1.3865ms  1.8491ms  phase_3(int*, int, int, int, int, int)
  4.62%  3.5486ms        50  70.972us  69.475us  71.747us  phase_2(int*, int, int, int, int, int)
  1.95%  1.4970ms         1  1.4970ms  1.4970ms  1.4970ms  [CUDA memcpy DtoH]
  1.54%  1.1870ms         1  1.1870ms  1.1870ms  1.1870ms  [CUDA memcpy HtoD]
  0.95%  732.13us        50  14.642us  14.016us  15.169us  phase_1(int*, int, int, int, int, int)

==5238== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 74.47%  284.99ms         2  142.49ms  2.1110us  284.98ms  cudaEventCreate
 23.51%  89.968ms         1  89.968ms  89.968ms  89.968ms  cudaEventSynchronize
  0.97%  3.7209ms         2  1.8605ms  1.4415ms  2.2794ms  cudaMemcpy
  0.40%  1.5345ms       150  10.229us  9.1420us  41.026us  cudaLaunch
  0.30%  1.1487ms       166  6.9200us     376ns  306.44us  cuDeviceGetAttribute
  0.11%  421.33us         1  421.33us  421.33us  421.33us  cudaMalloc
  0.08%  305.40us       900     339ns     286ns  5.4550us  cudaSetupArgument
  0.05%  203.79us         1  203.79us  203.79us  203.79us  cudaFree
  0.04%  145.55us         2  72.775us  54.398us  91.152us  cuDeviceTotalMem
  0.03%  100.89us         2  50.446us  39.280us  61.612us  cuDeviceGetName
  0.02%  70.199us       150     467ns     414ns  2.2920us  cudaConfigureCall
  0.01%  30.627us         1  30.627us  30.627us  30.627us  cudaSetDevice
  0.00%  14.904us         2  7.4520us  3.7010us  11.203us  cudaEventRecord
  0.00%  9.6150us         1  9.6150us  9.6150us  9.6150us  cudaEventElapsedTime
  0.00%  5.8030us         2  2.9010us  1.2180us  4.5850us  cudaEventDestroy
  0.00%  3.7580us         4     939ns     430ns  2.1580us  cuDeviceGet
  0.00%  2.3780us         2  1.1890us     648ns  1.7300us  cuDeviceGetCount
```

- Sample Test Case 5 (6000 nodes, 40000 edges)

```
[user10@gpucluster0 hw4]$ nvprof ./HW4_101062319_cuda.exe in5 cudaout5 20
==25746== NVPROF is profiling process 25746, command: ./HW4_101062319_cuda.exe in5 cudaout5 20
GPU Compute Time: 15605.726562
==25746== Profiling application: ./HW4_101062319_cuda.exe in5 cudaout5 20
==25746== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 97.72%  15.4848s       300  51.616ms  51.541ms  54.347ms  phase_3(int*, int, int, int, int, int)
  1.16%  183.79ms         1  183.79ms  183.79ms  183.79ms  [CUDA memcpy DtoH]
  0.72%  113.97ms       300  379.88us  374.06us  621.34us  phase_2(int*, int, int, int, int, int)
  0.38%  59.648ms         1  59.648ms  59.648ms  59.648ms  [CUDA memcpy HtoD]
  0.03%  4.2790ms       300  14.263us  13.856us  14.913us  phase_1(int*, int, int, int, int, int)

==25746== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 96.28%  15.5554s         1  15.5554s  15.5554s  15.5554s  cudaEventSynchronize
  1.89%  305.16ms         2  152.58ms  2.4160us  305.16ms  cudaEventCreate
  1.52%  245.84ms         2  122.92ms  60.082ms  185.76ms  cudaMemcpy
  0.27%  44.402ms       900  49.335us  44.690us  104.19us  cudaLaunch
  0.02%  2.4564ms      5400     454ns     322ns  31.681us  cudaSetupArgument
  0.01%  817.92us       166  4.9270us     340ns  176.58us  cuDeviceGetAttribute
  0.00%  569.84us       900     633ns     520ns  7.6820us  cudaConfigureCall
  0.00%  545.46us         1  545.46us  545.46us  545.46us  cudaFree
  0.00%  306.93us         1  306.93us  306.93us  306.93us  cudaMalloc
  0.00%  100.06us         2  50.027us  48.091us  51.964us  cuDeviceTotalMem
  0.00%  97.516us         2  48.758us  37.822us  59.694us  cudaEventRecord
  0.00%  80.584us         2  40.292us  39.036us  41.548us  cuDeviceGetName
  0.00%  38.194us         1  38.194us  38.194us  38.194us  cudaEventElapsedTime
  0.00%  17.612us         1  17.612us  17.612us  17.612us  cudaSetDevice
  0.00%  13.514us         2  6.7570us  1.5050us  12.009us  cudaEventDestroy
  0.00%  3.4060us         4     851ns     370ns  2.0540us  cuDeviceGet
  0.00%  2.5420us         2  1.2710us     574ns  1.9680us  cuDeviceGetCount
```

(b) OpenMP (Multiple-GPU)

- Sample Test Case 3 (1000 nodes, 5000 edges)

```
[user10@gpucluster0 hw4]$ nvprof ./HW4_101062319_openmp.exe in3 opout3 20
==6063== NVPROF is profiling process 6063, command: ./HW4_101062319_openmp.exe in3 opout3 20
GPU 1 Compute Time: 843.029907

GPU 0 Compute Time: 1136.638672

==6063== Profiling application: ./HW4_101062319_openmp.exe in3 opout3 20
==6063== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 49.29%  564.12ms        50  11.282ms  37.985us  23.897ms  phase_3_bot(int*, int, int, int, int, int)
 45.83%  524.60ms        50  10.492ms  33.697us  22.097ms  phase_3_top(int*, int, int, int, int, int)
  2.40%  27.509ms        50  550.18us  491.22us  634.93us  phase_2_col(int*, int, int, int, int, int)
  2.38%  27.245ms        50  544.90us  475.00us  646.11us  phase_2_row(int*, int, int, int, int, int)
  0.10%  1.1000ms        50  21.999us  19.104us  25.825us  phase_1(int*, int, int, int, int, int)

==6063== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 66.77%  1.14369s       200  5.7184ms  29.326us  23.902ms  cudaDeviceSynchronize
 16.63%  284.90ms         1  284.90ms  284.90ms  284.90ms  cudaHostAlloc
 15.86%  271.61ms         4  67.901ms  2.1360us  271.57ms  cudaEventCreate
  0.52%  8.8482ms       250  35.392us  10.299us  97.404us  cudaLaunch
  0.08%  1.3471ms         1  1.3471ms  1.3471ms  1.3471ms  cudaFreeHost
  0.05%  776.83us       166  4.6790us     338ns  176.10us  cuDeviceGetAttribute
  0.04%  703.45us      1500     468ns     303ns  12.244us  cudaSetupArgument
  0.03%  515.56us       252  2.0450us     890ns  9.9370us  cudaSetDevice
  0.01%  147.17us       250     588ns     408ns  2.7160us  cudaConfigureCall
  0.01%  112.53us         4  28.131us  6.3780us  58.700us  cudaEventRecord
  0.01%  99.717us         2  49.858us  49.847us  49.870us  cuDeviceTotalMem
  0.00%  83.813us         2  41.906us  37.128us  46.685us  cuDeviceGetName
  0.00%  11.586us         4  2.8960us  1.3500us  4.3900us  cudaEventDestroy
  0.00%  11.128us         2  5.5640us  3.5800us  7.5480us  cudaEventElapsedTime
  0.00%  7.7480us         2  3.8740us  3.1020us  4.6460us  cudaEventSynchronize
  0.00%  3.5860us         4     896ns     348ns  2.2240us  cuDeviceGet
  0.00%  3.1740us         1  3.1740us  3.1740us  3.1740us  cudaGetDeviceCount
  0.00%  2.2060us         2  1.1030us     540ns  1.6660us  cuDeviceGetCount
```

- Sample Test Case 5 (6000 nodes, 40000 edges)

```
[user10@gpucluster0 hw4]$ nvprof ./HW4_101062319_openmp.exe in5 opout5 20
==6075== NVPROF is profiling process 6075, command: ./HW4_101062319_openmp.exe in5 opout5 20
GPU 1 Compute Time: 238094.187500

GPU 0 Compute Time: 239813.750000

==6075== Profiling application: ./HW4_101062319_openmp.exe in5 opout5 20
==6075== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 50.47%  157.459s       300  524.86ms  1.1503ms  2.08097s  phase_3_top(int*, int, int, int, int, int)
 48.89%  152.538s       300  508.46ms  1.3183ms  1.05516s  phase_3_bot(int*, int, int, int, int, int)
  0.33%  1.01794s       300  3.3931ms  2.7858ms  5.9937ms  phase_2_col(int*, int, int, int, int, int)
  0.31%  958.95ms       300  3.1965ms  2.5185ms  8.5425ms  phase_2_row(int*, int, int, int, int, int)
  0.00%  7.7158ms       300  25.719us  21.441us  45.122us  phase_1(int*, int, int, int, int, int)

==6075== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 99.77%  312.093s      1200  260.08ms  1.1478ms  2.08114s  cudaDeviceSynchronize
  0.11%  331.92ms         1  331.92ms  331.92ms  331.92ms  cudaHostAlloc
  0.09%  278.40ms         4  69.601ms  1.7910us  278.37ms  cudaEventCreate
  0.02%  59.522ms      1500  39.681us  10.997us  314.52us  cudaLaunch
  0.01%  28.960ms         1  28.960ms  28.960ms  28.960ms  cudaFreeHost
  0.00%  4.6115ms      9000     512ns     257ns  48.617us  cudaSetupArgument
  0.00%  4.5234ms      1502  3.0110us     983ns  26.408us  cudaSetDevice
  0.00%  1.1679ms      1500     778ns     380ns  14.473us  cudaConfigureCall
  0.00%  726.77us       166  4.3780us     284ns  161.28us  cuDeviceGetAttribute
  0.00%  114.94us         2  57.468us  4.5280us  110.41us  cudaEventSynchronize
  0.00%  108.26us         4  27.064us  5.8500us  80.876us  cudaEventRecord
  0.00%  88.076us         2  44.038us  42.254us  45.822us  cuDeviceTotalMem
  0.00%  71.794us         2  35.897us  33.672us  38.122us  cuDeviceGetName
  0.00%  20.120us         2  10.060us  9.1130us  11.007us  cudaEventElapsedTime
  0.00%  17.957us         4  4.4890us  1.3600us  9.4950us  cudaEventDestroy
  0.00%  4.0000us         1  4.0000us  4.0000us  4.0000us  cudaGetDeviceCount
  0.00%  2.9900us         4     747ns     305ns  1.8700us  cuDeviceGet
  0.00%  1.9560us         2     978ns     444ns  1.5120us  cuDeviceGetCount
```

(b) MPI (Multiple-GPU)

- Sample Test Case 3 (1000 nodes, 5000 edges)
  - Proc_0

```
======== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 58.43%  569.02ms        50  11.380ms  75.793us  22.692ms  phase_3_top(int*, int, int, int, int, int)
 37.83%  368.39ms        50  7.3678ms  88.215us  15.289ms  update_phase3(int*, int*, int, int, int, int, int)
  2.01%  19.623ms        50  392.46us  359.49us  425.39us  phase_2_row(int*, int, int, int, int, int)
  1.66%  16.210ms        50  324.20us  292.71us  364.29us  update_phase2(int*, int*, int, int, int, int, int)
  0.07%  651.36us        50  13.027us  11.608us  14.925us  phase_1(int*, int, int, int, int, int)

======== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 88.92%  977.32ms       250  3.9093ms  17.460us  22.709ms  cudaDeviceSynchronize
 10.44%  114.73ms         2  57.367ms  2.3101ms  112.42ms  cudaHostAlloc
  0.30%  3.3223ms       250  13.289us  7.6570us  42.882us  cudaLaunch
  0.18%  2.0183ms         2  1.0091ms  905.83us  1.1124ms  cudaFreeHost
  0.06%  665.42us       166  4.0080us     390ns  145.19us  cuDeviceGetAttribute
  0.05%  577.36us      1600     360ns     259ns  3.9040us  cudaSetupArgument
  0.02%  245.12us       250     980ns     397ns  6.1460us  cudaConfigureCall
  0.01%  77.886us         2  38.943us  38.402us  39.484us  cuDeviceTotalMem
  0.01%  67.530us         2  33.765us  30.766us  36.764us  cuDeviceGetName
  0.00%  23.925us         2  11.962us  7.2530us  16.672us  cudaEventRecord
  0.00%  17.521us         2  8.7600us  1.3650us  16.156us  cudaEventDestroy
  0.00%  11.904us         2  5.9520us  1.7620us  10.142us  cudaEventCreate
  0.00%  6.6650us         1  6.6650us  6.6650us  6.6650us  cudaEventElapsedTime
  0.00%  6.6160us         1  6.6160us  6.6160us  6.6160us  cudaSetDevice
  0.00%  3.6500us         4     912ns     412ns  2.0240us  cuDeviceGet
  0.00%  2.7450us         1  2.7450us  2.7450us  2.7450us  cudaEventSynchronize
  0.00%  2.4380us         2  1.2190us     682ns  1.7560us  cuDeviceGetCount
```

- Proc_1

```
======== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 96.81%  672.18ms        50  13.444ms  86.890us  30.255ms  phase_3_bot(int*, int, int, int, int, int)
  3.19%  22.115ms        50  442.29us  355.10us  588.57us  phase_2_col(int*, int, int, int, int, int)

======== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 74.85%  696.88ms       100  6.9688ms  89.969us  30.293ms  cudaDeviceSynchronize
 12.80%  119.18ms         2  59.590ms  2.1240us  119.18ms  cudaEventCreate
 11.66%  108.51ms         2  54.257ms  2.2583ms  106.26ms  cudaHostAlloc
  0.36%  3.3314ms         2  1.6657ms  1.2334ms  2.0981ms  cudaFreeHost
  0.20%  1.8683ms       100  18.682us  13.307us  155.45us  cudaLaunch
  0.07%  626.15us       166  3.7720us     352ns  129.58us  cuDeviceGetAttribute
  0.03%  260.24us       600     433ns     273ns  7.4600us  cudaSetupArgument
  0.02%  153.26us       100  1.5320us     949ns  23.777us  cudaConfigureCall
  0.01%  73.772us         2  36.886us  35.962us  37.810us  cuDeviceTotalMem
  0.01%  63.804us         2  31.902us  29.890us  33.914us  cuDeviceGetName
  0.00%  17.489us         2  8.7440us  8.4890us  9.0000us  cudaEventRecord
  0.00%  7.6400us         1  7.6400us  7.6400us  7.6400us  cudaSetDevice
  0.00%  5.8820us         1  5.8820us  5.8820us  5.8820us  cudaEventElapsedTime
  0.00%  5.5430us         2  2.7710us  1.1380us  4.4050us  cudaEventDestroy
  0.00%  3.4560us         4     864ns     410ns  1.9000us  cuDeviceGet
  0.00%  2.7410us         1  2.7410us  2.7410us  2.7410us  cudaEventSynchronize
  0.00%  2.3480us         2  1.1740us     634ns  1.7140us  cuDeviceGetCount
```

- Sample Test Case 5 (6000 nodes, 40000 edges)
  - Proc_0

```
======== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 65.96%  144.938s       300  483.13ms  1.1517ms  979.76ms  phase_3_top(int*, int, int, int, int, int)
 33.39%  73.3769s       300  244.59ms  1.2396ms  945.02ms  update_phase3(int*, int*, int, int, int, int, int)
  0.42%  923.13ms       300  3.0771ms  2.4313ms  6.0480ms  phase_2_row(int*, int, int, int, int, int)
  0.22%  482.53ms       300  1.6084ms  1.2829ms  2.8597ms  update_phase2(int*, int*, int, int, int, int, int)
  0.00%  7.7220ms       300  25.740us  20.257us  51.907us  phase_1(int*, int, int, int, int, int)

======== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 99.63%  219.907s      1500  146.61ms  21.838us  979.82ms  cudaDeviceSynchronize
  0.31%  692.14ms         2  346.07ms  79.444ms  612.69ms  cudaHostAlloc
  0.03%  66.290ms      1500  44.193us  11.658us  145.00us  cudaLaunch
  0.02%  52.503ms         2  26.251ms  25.605ms  26.897ms  cudaFreeHost
  0.00%  5.5271ms      9600     575ns     298ns  13.423us  cudaSetupArgument
  0.00%  3.4098ms      1500  2.2730us     448ns  16.535us  cudaConfigureCall
  0.00%  1.9084ms       166  11.496us     398ns  460.98us  cuDeviceGetAttribute
  0.00%  206.24us         2  103.12us  102.72us  103.52us  cuDeviceTotalMem
  0.00%  144.13us         2  72.063us  61.586us  82.540us  cuDeviceGetName
  0.00%  118.89us         2  59.446us  8.7480us  110.14us  cudaEventRecord
  0.00%  58.987us         1  58.987us  58.987us  58.987us  cudaEventSynchronize
  0.00%  15.952us         2  7.9760us  1.9540us  13.998us  cudaEventCreate
  0.00%  14.408us         1  14.408us  14.408us  14.408us  cudaSetDevice
  0.00%  9.1220us         1  9.1220us  9.1220us  9.1220us  cudaEventElapsedTime
  0.00%  6.4410us         2  3.2200us  1.2610us  5.1800us  cudaEventDestroy
  0.00%  4.3360us         4  1.0840us     427ns  2.6950us  cuDeviceGet
  0.00%  3.0070us         2  1.5030us     690ns  2.3170us  cuDeviceGetCount
```

- Proc_1

```
======== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 99.41%  111.590s       300  371.97ms  3.1106ms  828.40ms  phase_3_bot(int*, int, int, int, int, int)
  0.59%  658.99ms       300  2.1966ms  1.9918ms  2.9506ms  phase_2_col(int*, int, int, int, int, int)

======== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 98.40%  114.072s       600  190.12ms  2.0152ms  830.87ms  cudaDeviceSynchronize
  1.26%  1.45909s         2  729.55ms  78.892ms  1.38020s  cudaHostAlloc
  0.25%  294.11ms         2  147.06ms  1.8650us  294.11ms  cudaEventCreate
  0.06%  64.231ms         2  32.115ms  30.198ms  34.033ms  cudaFreeHost
  0.03%  31.329ms       600  52.214us  42.361us  166.58us  cudaLaunch
  0.00%  2.2319ms       600  3.7190us  2.8460us  14.533us  cudaConfigureCall
  0.00%  2.1039ms      3600     584ns     256ns  54.593us  cudaSetupArgument
  0.00%  702.66us       166  4.2320us     372ns  155.40us  cuDeviceGetAttribute
  0.00%  84.974us         2  42.487us  39.252us  45.722us  cuDeviceTotalMem
  0.00%  72.342us         2  36.171us  30.514us  41.828us  cuDeviceGetName
  0.00%  23.968us         2  11.984us  7.7710us  16.197us  cudaEventRecord
  0.00%  22.710us         1  22.710us  22.710us  22.710us  cudaEventElapsedTime
  0.00%  10.705us         2  5.3520us  1.5090us  9.1960us  cudaEventDestroy
  0.00%  8.7030us         1  8.7030us  8.7030us  8.7030us  cudaSetDevice
  0.00%  5.2550us         1  5.2550us  5.2550us  5.2550us  cudaEventSynchronize
  0.00%  3.6900us         4     922ns     408ns  2.1020us  cuDeviceGet
  0.00%  2.6520us         2  1.3260us     678ns  1.9740us  cuDeviceGetCount
```

- Experiment & Analysis

(a) Weak Scalability (scalability to problem size)

i. Different input size (based on sample test cases)

Sample Test Case Information:

	Nodes (vertices)	Edges
Test Case 1:	**10**	**50**
Test Case 2:	**100**	**1000**
Test Case 3:	**1000**	**5000**
Test Case 4:	**3000**	**15000**
Test Case 5:	**6000**	**40000**

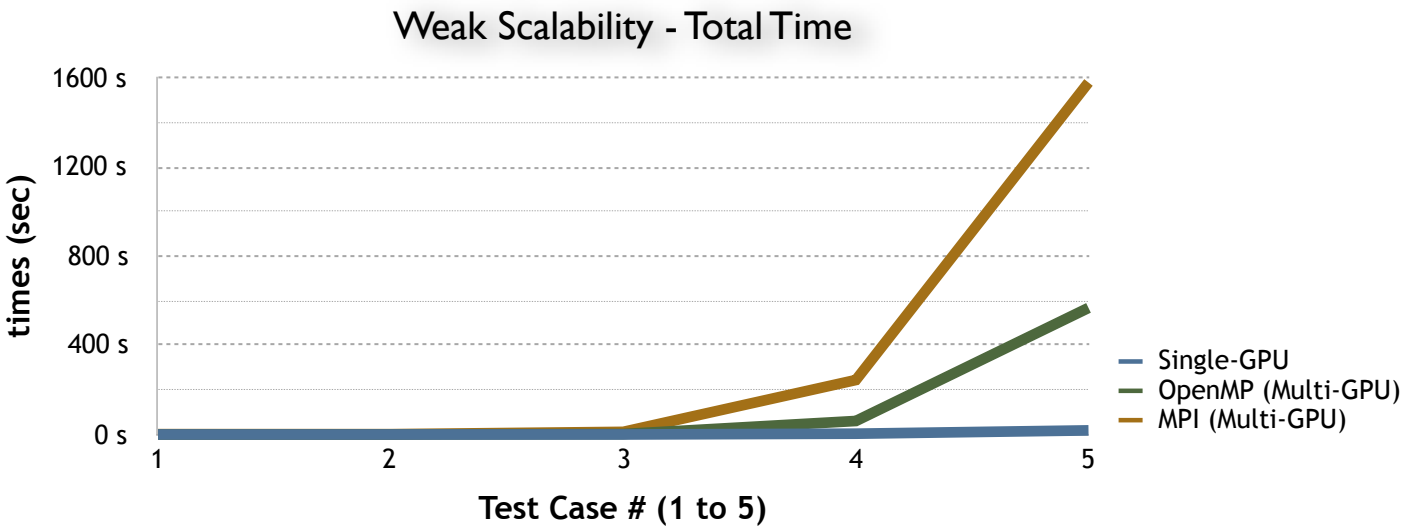
Executing Configuration & Environment:

./HW4_101062319_{version} in{#test case} out{#test case} 10

@ gpucluster0 :

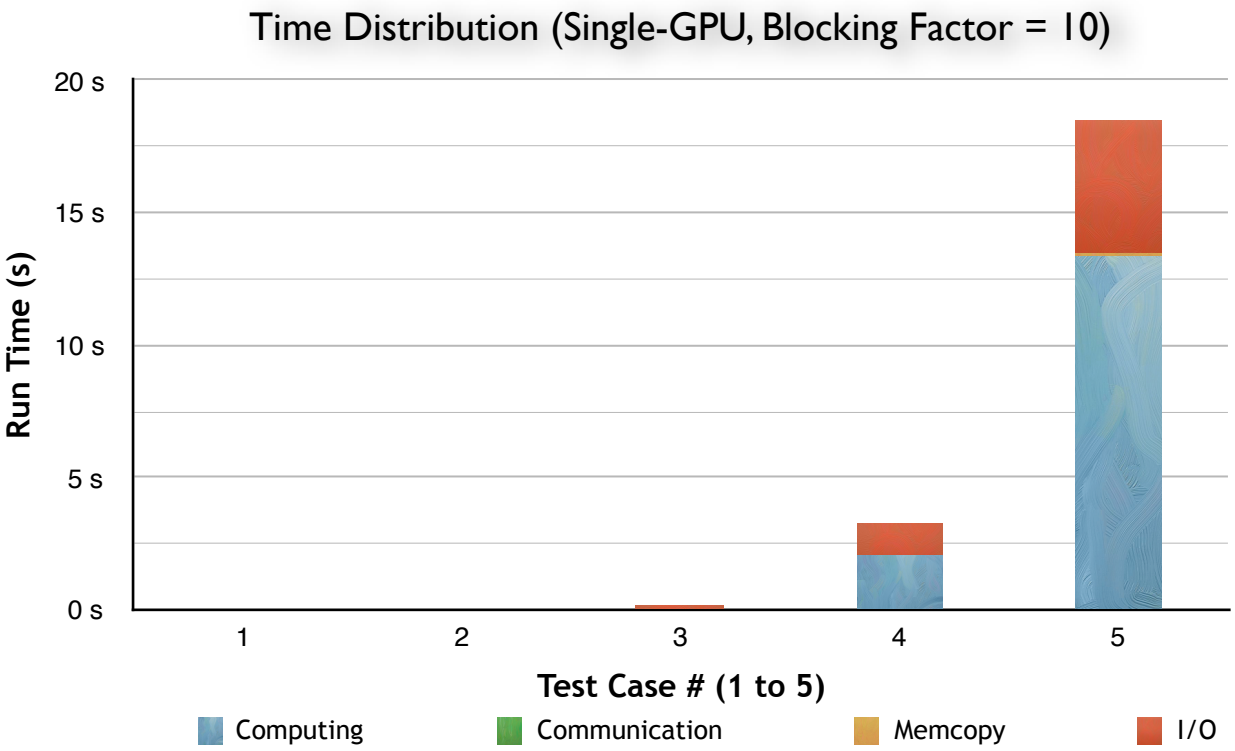
Nvidia-SMI: 346.59 Driver version: 346.59

Tesla K20m



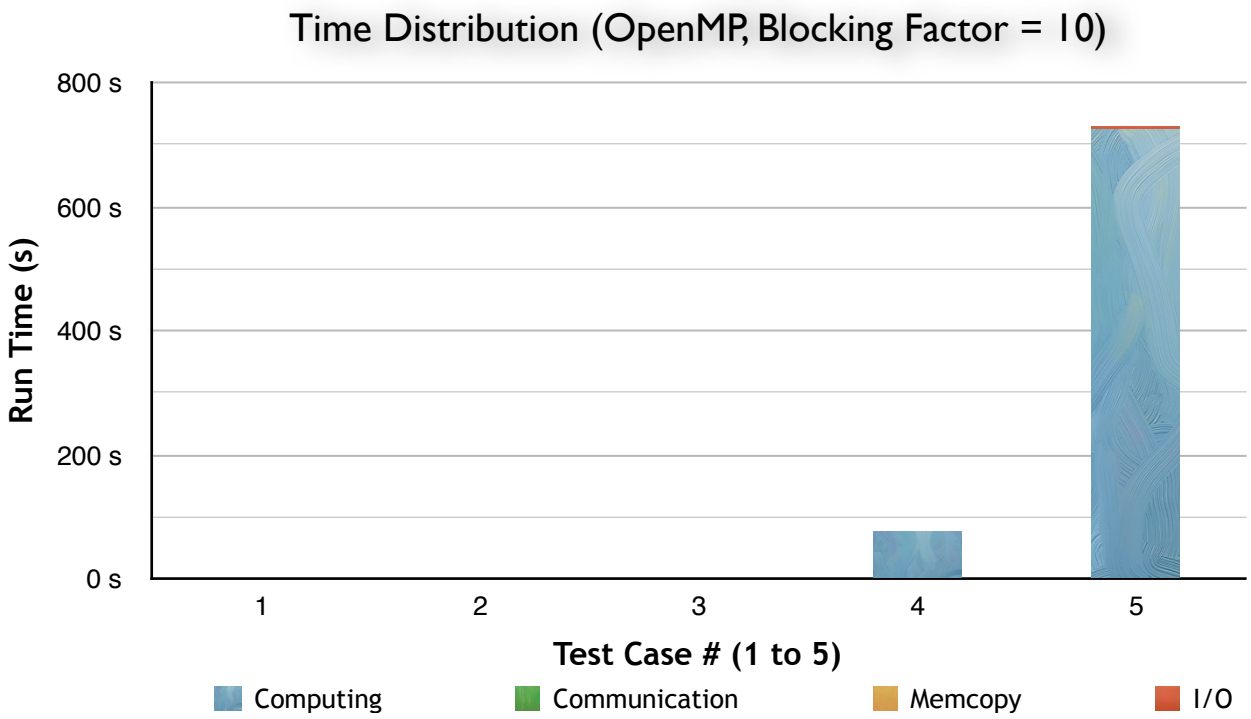
According to the above graph, it's apparently to see that the performance of the single-GPU is the best. Although the restriction is that the multi-GPU version like OpenMP and MPI must faster than single-GPU version, I think the difference of the speed is based on the location where the array exists. In my single-GPU version, I set the array live in GPU's global memory and the shared memory within each block. However, in my OpenMP and MPI version, all the stored arrays are live in "pinned memory" in the CPU Host side. So that's the main point why my single-version is much faster than the multiple-GPU version which is represented by the above graph.

(b) Time Distribution

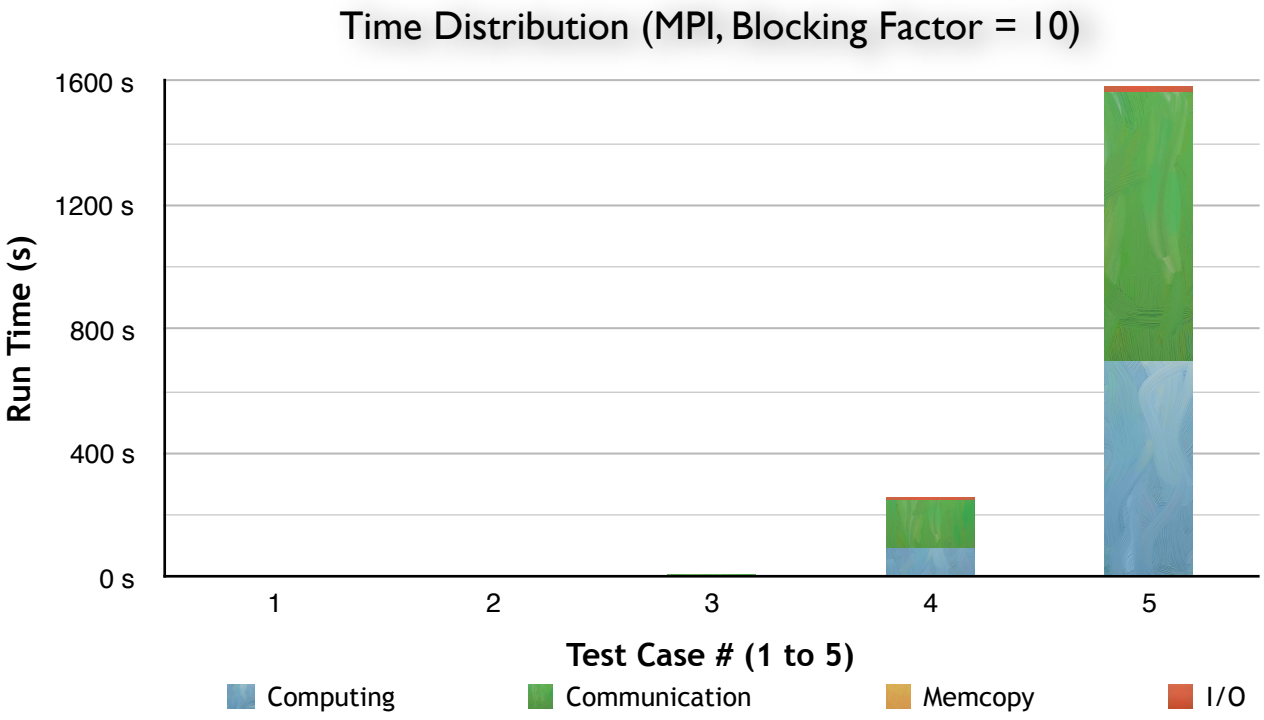


I record the computing part by the "nvprof". Since in my implementation that I only compute the data in my own kernel function "phase_1", "phase_2" and "phase_3", I only record the time that the "nvprof" shows. Due to there is only single-GPU, so the communication time is always zero. And the memory copy part is record only when executing the function "cudaMemcpy",

so I also record the time “nvprof” shows. Finally, I record the I/O part during executing the function “input” and the time spent on run the function “output”.



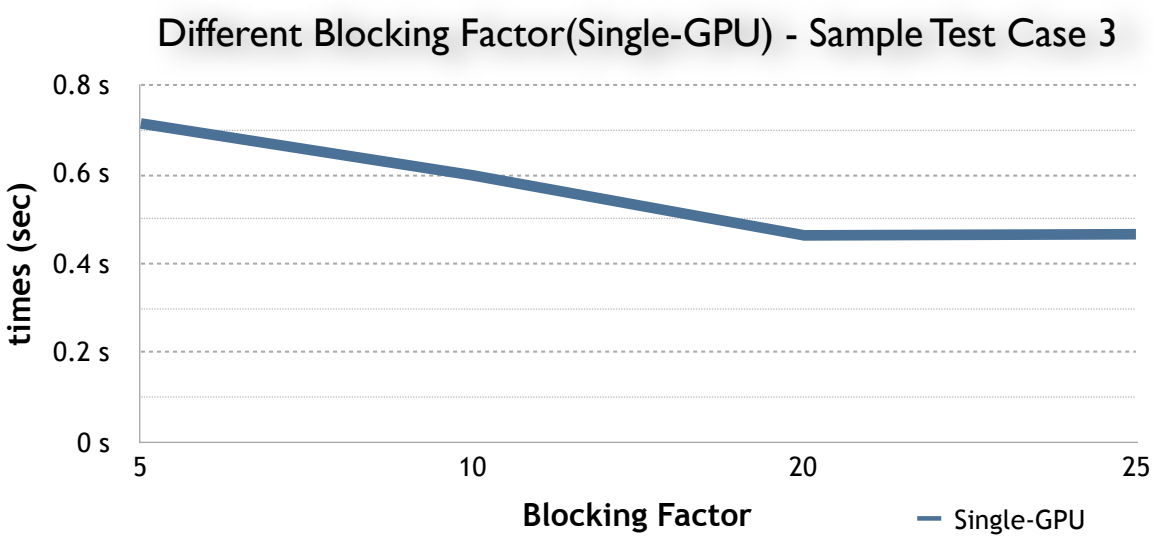
As the above bar graph shows, I record the computing part by the “nvprof”. Since in my implementation that I only compute the data in my own kernel function “phase_1”, “phase_2_row”, “phase_2_col”, “phase_3_top” and “phase_3_bot”, I only record the time that the “nvprof” shows. Due to the usage of “pinned memory”, so I think there is no need to compute the time of communication. And the memory copy part is also zero since the same reason why communication time is zero. Finally, I record the I/O part during executing the function “input” and the time spent on run the function “output”.

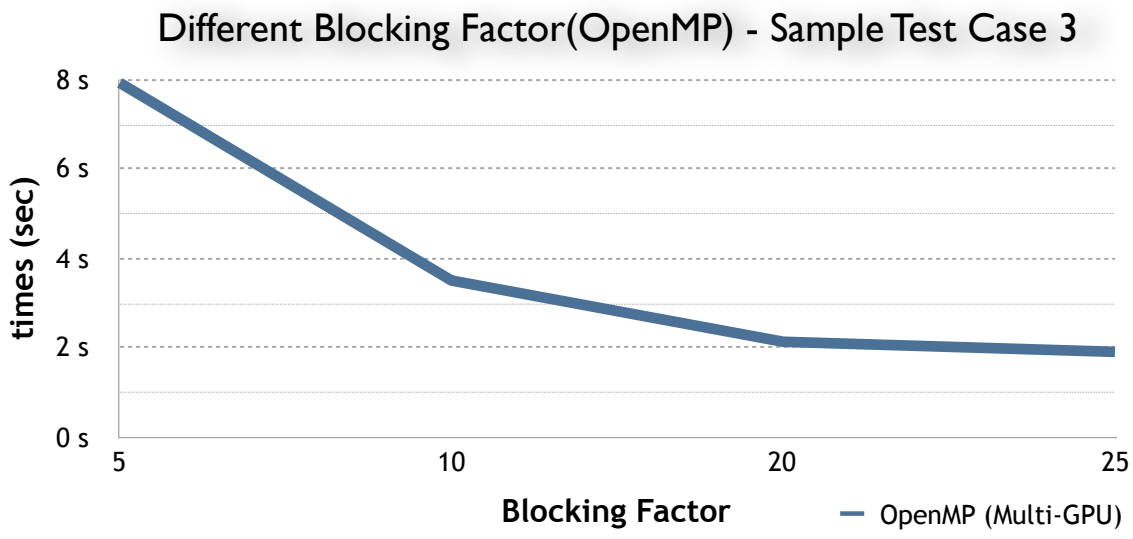


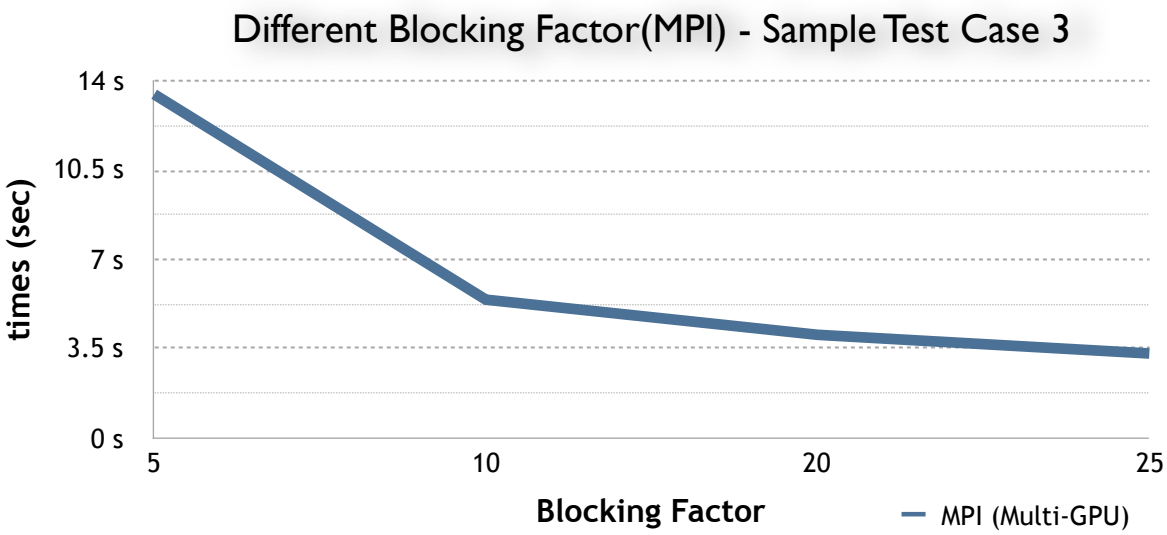
Different from the previous graph, it's easily to see the part of communication. Because of the data transmission between the master process and slave process. I record the computing part by the "nvprof". Since in my implementation that I only compute the data in my own kernel function "phase_1", "phase_2_row", "phase_2_col", "phase_3_top" and "phase_3_bot", I only record the time that the "nvprof" shows. The next part is communication, I record the communication time between every "MPI_Isend", "MPI_Send" and "MPI_Recv" function. And the memory copy part is also zero since the direct usage of pinned memory. Finally, I record the I/O part during executing the function "input" and the time spent on run the function "output".

(c) Blocking Factor

(d) Others







- Experience / Conclusion

(a) What have you learned from this assignment?

I learned so much from this assignment… First, I learned the background knowledge of Nvidia GPU architecture such like the terminology "warp", "grid" and "block". Second, I learned the syntax of CUDA, CUDA+OpenMP and CUAD+MPI. Also, there exists so many confusing bugs or complex problems such like the timing to synchronize the data and even which way to synchronize, for example, should I need to use "#pragma omp barrier" or "cudaEventSynchronize()"? Such annoying problems make me even crazy…… Fortunately, after discussing with classmates and via so many try-and-error tests, I fixed the bugs step by step so that I can submit the result…

Finally, I was really surprised by the accelerating of GPU. Since Sequential version program need to take about 10 minutes or even longer, I thought that the GPU version may be take about 5 minutes; however, to my surprise, it only need to take less than a minute… It's my first time to see the power of accelerating by the GPU via my own implementation. And to see is to believe, I totally realize why the heterogeneous system is so important and the cuda parallel technology will gradually become the accelerating assistant tool of the scientific computing.

(b) What difficulty did you encounter when implementing this assignment?

I think the most difficult part is how to implement the first code and how to accelerate it. After finishing the first cuda single-GPU code, I adopt the global memory to transfer the data. However, it's speed is a little bit slow, so I try to modify it to be the shared memory version. And I stuck at this stage so long… Due to the computation and the concept of how to indexing the new index to shared memory and how to assign the number of grids and threads, I think it's the most difficult part. And when it comes to the OpenMP and MPI version, they are also my nightmare too… I was really confused by the synchronize problem with both OpenMP and MPI version… I found out that the logic of my program seemed to work, but, it didn't… What's worse, after surveying the information on the internet, I still found that the logic is still correct, but I just couldn't find the prob-

lem… So I use try-and-error method to test every possible solution to both OpenMP and MPI version. Although I can run the program now, there still exists some problems such like why "B" must be the factor of "n" in my program… The above descriptions are all my internal murmur when I was coding and finding the solution duration the past several weeks… Fortunately, I can find a way to solve some of them to make my program runnable under specific restrictions. I think this assignment is the most difficult among the all assignments, and, it's very useful and helpful for me to understand the knowledge of parallel language and CUDA. Thanks for the teacher and all TAs' efforts this semester. I am very grateful for taking this course:)