資工系100062236林修安

Parallel Programming HW4

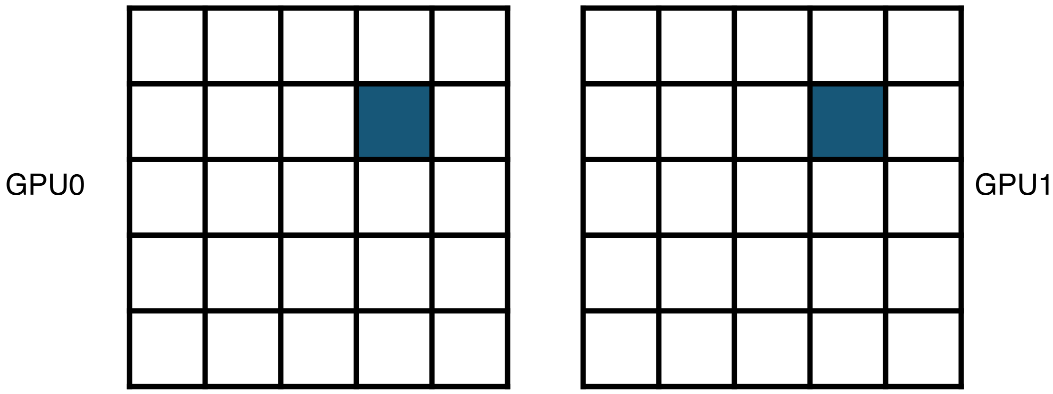
Blocked All-Pairs Shortest Path

Implementation

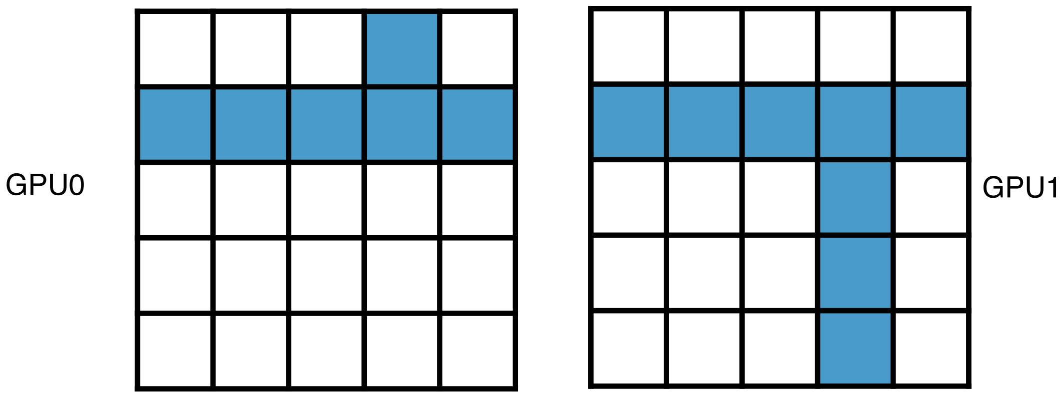
- Single GPU
    1. Change `Dist[]` to 1-D array
    2. Use `dim3` data structure to launch GPU
    3. Inside kernel function, use `blockIdx.x` and `blockIdx.y` to relabel block indices, and `threadIdx.x` and `threadIdx.y` to relabel real array indices, so such four loops can be parallelized
    4. After each `k`, threads should be synchronized so the afterward iterations have the updated values

- OpenMp
    1. The mentioned modifications from single GPU version
    2. Change `dev_dist[]` to a 2-D array that GPU0 holds `dev_dist[0]` and GPU1 holds `dev_dist[1]`
    3. Inside block_APSP(), use `omp_get_thread_num()` as gpu_id to realize the parallelism
    4. Phase 1: Both GPU calculates the pivot block
Phase 2: GPU0 gets the pivot row, and the upper part of pivot column
GPU1 gets the pivot row, and the lower part of the pivot column
Phase 3: GPU0 gets the upper part of the `dev_dist[]`
GPU1 gets the lower part of the `dev_dist[]`
    5. After phase 3, GPU0 and GPU1 have to copy the upper part and the lower part respectively to host, and take the complementary part back to GPU.

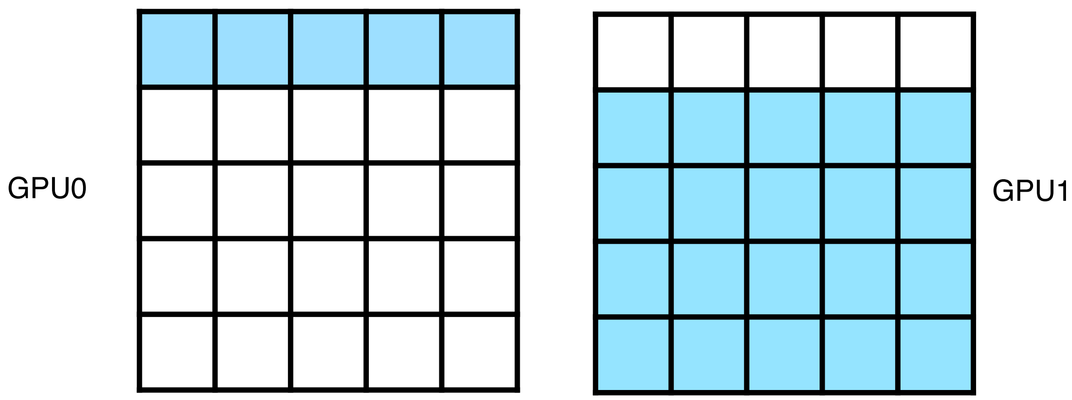
- MPI
    1. All phases are same as OpenMp version
    2. After each round, GPU0 and GPU1 have to copy the upper part and the lower part respectively to host
    3. Process0 sends the upper part to process1, and process1 sends the lower part to process0 using non-blocking send
    4. Each process copies the received part to each GPU



For OpenMp and MPI versions, GPU0 and GPU1 get the pivot block and do calculation in phase 1.



In phase 2, GPU0 gets the pivot row and the upper pivot column; GPU1 also gets the pivot row, and the lower pivot column.



In phase 3, GPU0 gets the upper part above the pivot row; GPU1 gets the lower part below the pivot row.

Profiling Results

- nvprof ./HW4_cuda.exe Testcase/in5 out5

```
[user12@gpucluster2 HW4]$ nvprof ./HW4_cuda.exe Testcase/in5 out5
==26999== NVPROF is profiling process 26999, command: ./HW4_cuda.exe Testcase/in5 out5
total  time  59101.933
==26999== Profiling application: ./HW4_cuda.exe Testcase/in5 out5
==26999== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 99.05%  56.3268s       846  66.580ms     934ns  569.32ms  cal(int, int, int, int, int, int, int, int*)
  0.50%  282.34ms         1  282.34ms  282.34ms  282.34ms  [CUDA memcpy DtoH]
  0.45%  257.03ms         1  257.03ms  257.03ms  257.03ms  [CUDA memcpy HtoD]

==26999== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 99.43%  58.6655s         2  29.3327s  257.67ms  58.4078s  cudaMemcpy
  0.52%  307.08ms         1  307.08ms  307.08ms  307.08ms  cudaMalloc
  0.03%  20.630ms         1  20.630ms  20.630ms  20.630ms  cudaFree
  0.01%  5.5659ms       846  6.5790us  6.0720us  39.422us  cudaLaunch
  0.00%  1.8420ms      6768     272ns     246ns  12.048us  cudaSetupArgument
  0.00%  510.44us       166  3.0740us     261ns  108.15us  cuDeviceGetAttribute
  0.00%  312.43us       846     369ns     338ns  3.0700us  cudaConfigureCall
  0.00%  79.953us         2  39.976us  35.628us  44.325us  cuDeviceTotalMem
  0.00%  58.336us         2  29.168us  26.152us  32.184us  cuDeviceGetName
  0.00%  4.0650us         1  4.0650us  4.0650us  4.0650us  cudaSetDevice
  0.00%  2.1490us         2  1.0740us     539ns  1.6100us  cuDeviceGetCount
  0.00%  1.8490us         4     462ns     337ns     571ns  cuDeviceGet
  0.00%  1.4800us         1  1.4800us  1.4800us  1.4800us  cudaGetDeviceCount
```

This part is the main results I use as the experiment outcomes

- nvprof ./HW4_openmp.exe Testcase/in5 out5

```
[user12@gpucluster2 HW4]$ nvprof ./HW4_openmp.exe Testcase/in5 out5
==27241== NVPROF is profiling process 27241, command: ./HW4_openmp.exe Testcase/in5 out5
total      time  66265.210
==27241== Profiling application: ./HW4_openmp.exe Testcase/in5 out5
==27241== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 81.53%  57.8522s      1128  51.287ms     793ns  573.53ms  cal(int, int, int, int, int, int, int, int*)
  9.70%  6.87938s       189  36.399ms  195.62us  703.57ms  [CUDA memcpy HtoD]
  8.77%  6.22440s       188  33.109ms  178.24us  632.93ms  [CUDA memcpy DtoH]

==27241== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 88.75%  70.1226s       188  372.99ms  1.0760us  1.25781s  cudaMemcpyPeer
  4.11%  3.25028s         3  1.08343s  650.47ms  1.89358s  cudaMemcpy
  3.72%  2.93531s         2  1.46766s  1.08206s  1.85325s  cudaMalloc
  3.42%  2.69864s      1128  2.3924ms  6.5090us  496.89ms  cudaLaunch
  0.00%  3.2953ms      9024     365ns     246ns  12.572us  cudaSetupArgument
  0.00%  669.12us      1128     593ns     386ns  13.410us  cudaConfigureCall
  0.00%  633.45us       191  3.3160us  1.2660us  19.336us  cudaSetDevice
  0.00%  578.00us       166  3.4810us     264ns  141.42us  cuDeviceGetAttribute
  0.00%  88.225us         2  44.112us  36.663us  51.562us  cuDeviceGetName
  0.00%  76.671us         2  38.335us  35.557us  41.114us  cuDeviceTotalMem
  0.00%  37.867us         1  37.867us  37.867us  37.867us  cudaFree
  0.00%  3.1590us         4     789ns     413ns     929ns  cuDeviceGet
  0.00%  2.2240us         2  1.1120us     670ns  1.5540us  cuDeviceGetCount
```

1. computing time
2. memcpy time
3. memcpy time

- nvprof —print-summary-per-gpu ./HW4_openmp.exe Testcase/in5 out5

```
[user12@gpucluster2 HW4]$ nvprof --print-summary-per-gpu ./HW4_openmp.exe Testcase/in5 out5
==27495== NVPROF is profiling process 27495, command: ./HW4_openmp.exe Testcase/in5 out5
total     time  56900.494
==27495== Profiling application: ./HW4_openmp.exe Testcase/in5 out5
==27495== Profiling result:

==27495== Device "Tesla M2090 (0)"
Time(%)      Time     Calls       Avg       Min       Max  Name
 92.21%  28.9818s       564  51.386ms     845ns  575.07ms  cal(int, int, int, int, int, int, int, int*)
  3.91%  1.22786s        94  13.062ms  239.46us  69.300ms  [CUDA memcpy DtoH]
  3.88%  1.21986s        95  12.841ms  194.08us  61.025ms  [CUDA memcpy HtoD]

==27495== Device "Tesla M2090 (1)"
Time(%)      Time     Calls       Avg       Min       Max  Name
 92.32%  28.8798s       564  51.205ms     782ns  569.17ms  cal(int, int, int, int, int, int, int, int*)
  3.98%  1.24388s        94  13.233ms  261.38us  83.971ms  [CUDA memcpy HtoD]
  3.71%  1.15996s        94  12.340ms  177.89us  24.607ms  [CUDA memcpy DtoH]
```

Information about different threads can also be separately output

- mpirun -np 2 -hostfile hostfile ./profile.sh -o result.%q{PMI_RANK} ./HW4_mpi.exe Testcase/in5 out5

  nvprof —import-profile result.0.

```
[user12@gpucluster2 HW4]$ mpirun -np 2 -hostfile hostfile ./profile.sh -o result.%q{PMI_RANK} ./HW4_mpi.exe Testcase/in5 out5
==27767== NVPROF is profiling process 27767, command: ./HW4_mpi.exe Testcase/in5 out5
==27765== NVPROF is profiling process 27765, command: ./HW4_mpi.exe Testcase/in5 out5
total[0]  time 101235.293
comp[0]   time  30521.711
comm[0]   time  48565.561
mem[0]    time  21678.266
total[1]  time 101284.605
comp[1]   time  29922.008
comm[1]   time  48075.114
mem[1]    time  22896.766
==27765== Generated result file: /home/cs542200/user12/HW4/result.0.
==27767== Generated result file: /home/cs542200/user12/HW4/result.1.
[user12@gpucluster2 HW4]$ nvprof --import-profile result.0.
======== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 58.21%  29.0014s       564  51.421ms     849ns  568.17ms  cal(int, int, int, int, int, int, int, int*)
 27.01%  13.4587s        95  141.67ms  441.99us  2.49106s  [CUDA memcpy HtoD]
 14.78%  7.36277s        93  79.170ms  251.88us  490.58ms  [CUDA memcpy DtoH]

======== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 58.36%  30.6486s        94  326.05ms  16.682ms  616.00ms  cudaEventSynchronize
 41.38%  21.7348s       189  115.00ms     964ns  2.52154s  cudaMemcpy
  0.22%  115.25ms         1  115.25ms  115.25ms  115.25ms  cudaMalloc
  0.02%  10.371ms       166  62.477us     354ns  8.8386ms  cuDeviceGetAttribute
  0.01%  6.8556ms       564  12.155us  6.4140us  57.975us  cudaLaunch
  0.00%  1.7430ms      4512     386ns     246ns  11.604us  cudaSetupArgument
  0.00%  1.1750ms       188  6.2500us  3.0830us  19.368us  cudaEventRecord
  0.00%  418.32us        94  4.4500us  2.3800us  15.869us  cudaEventElapsedTime
  0.00%  379.11us       564     672ns     360ns  9.1090us  cudaConfigureCall
  0.00%  290.50us         1  290.50us  290.50us  290.50us  cudaFree
  0.00%  257.14us         2  128.57us  100.82us  156.32us  cuDeviceTotalMem
  0.00%  121.54us         2  60.768us  58.304us  63.232us  cuDeviceGetName
  0.00%  13.948us         2  6.9740us  1.9720us  11.976us  cudaEventCreate
  0.00%  6.4300us         1  6.4300us  6.4300us  6.4300us  cudaSetDevice
  0.00%  3.0000us         4     750ns     616ns     874ns  cuDeviceGet
  0.00%  2.9580us         2  1.4790us     956ns  2.0020us  cuDeviceGetCount
```

nvprof —import-profile result.1.

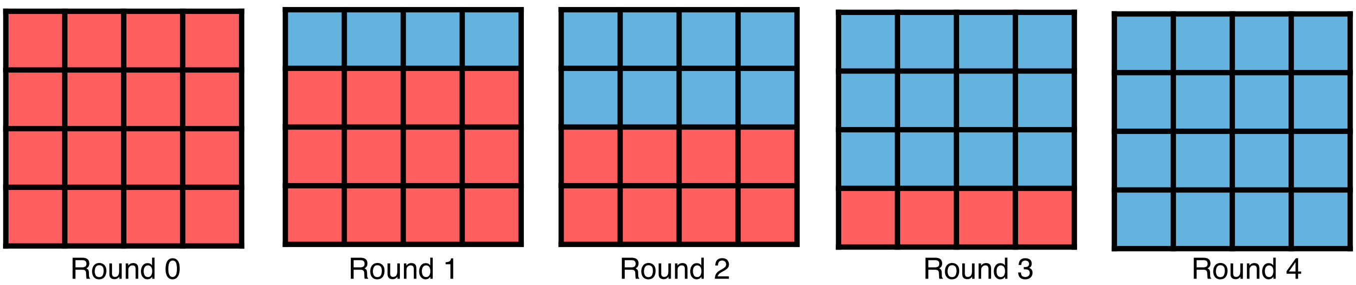
```
[user12@gpucluster2 HW4]$ nvprof --import-profile result.1.
======== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 56.45%  28.9063s       564  51.252ms     946ns  569.74ms  cal(int, int, int, int, int, int, int, int*)
 22.88%  11.7191s        94  124.67ms  203.40us  1.60258s  [CUDA memcpy DtoH]
 20.67%  10.5840s        94  112.60ms  257.92us  843.64ms  [CUDA memcpy HtoD]

======== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 56.41%  29.9467s        94  318.58ms  8.8569ms  946.22ms  cudaEventSynchronize
 43.27%  22.9696s       189  121.53ms  1.3450us  1.60381s  cudaMemcpy
  0.30%  160.40ms         1  160.40ms  160.40ms  160.40ms  cudaMalloc
  0.01%  6.5765ms       564  11.660us  6.5140us  58.254us  cudaLaunch
  0.00%  1.7231ms      4512     381ns     256ns  8.3400us  cudaSetupArgument
  0.00%  1.6952ms       166  10.212us     330ns  501.49us  cuDeviceGetAttribute
  0.00%  1.0909ms       188  5.8020us  2.9900us  13.097us  cudaEventRecord
  0.00%  373.84us        94  3.9770us  2.5830us  10.947us  cudaEventElapsedTime
  0.00%  362.13us         1  362.13us  362.13us  362.13us  cudaFree
  0.00%  348.10us       564     617ns     327ns  6.8860us  cudaConfigureCall
  0.00%  176.90us         2  88.448us  85.979us  90.917us  cuDeviceTotalMem
  0.00%  110.90us         2  55.447us  45.886us  65.009us  cuDeviceGetName
  0.00%  11.096us         2  5.5480us  1.9600us  9.1360us  cudaEventCreate
  0.00%  6.4870us         1  6.4870us  6.4870us  6.4870us  cudaSetDevice
  0.00%  3.3410us         2  1.6700us  1.0400us  2.3010us  cuDeviceGetCount
  0.00%  2.9530us         4     738ns     630ns     837ns  cuDeviceGet
```

Experiment & Analysis

- Time Evaluation:
  - Always select the maximum time as the termination of execution.

- GFLOPS Calculation:
  - I count the total calculation required in one execution as the total amount of floating operations, then divide it by 10^9 and the execution time.
  - For single GPU version, the counting is done as below:
    (1) An execution requires $\frac{n}{B}$ rounds
    (2) During each round, there are $\left(\frac{n}{B}\right)^2$ blocks
    (3) To do APSP, there are B intermediate vertices
    (4) For each intermediate vertex, B^2 calculations are required
    (5) Obtain counts of calculation is $\frac{n}{B}\times\left(\frac{n}{B}\right)^2\times B\times B^2=n^3$
  - For OpenMp and MPI versions, the counting is done as below:
    (1) Consider GPU0, and recall that GPU0 gets the upper part of distance matrix
    (2) For round i, the number of blocks is $i\times\frac{n}{B}$, so total number of blocks calculated after execution is $\frac{n}{B}\sum_{i=1}^{n/B} i=\frac{n^2B+n^3}{2B^3}$
    (3) Additionally, B intermediate vertices and B^2 calculations for each
    (4) Obtain counts of calculation is $\frac{n^2B+n^3}{2B^3}\times B\times B^2=\frac{n^2(B+n)}{2}$



For each round in OpenMp and MPI versions, GPU0 gets the blue parts and GPU1 get the red parts, and it's evident that the number of blocks increases row by row.

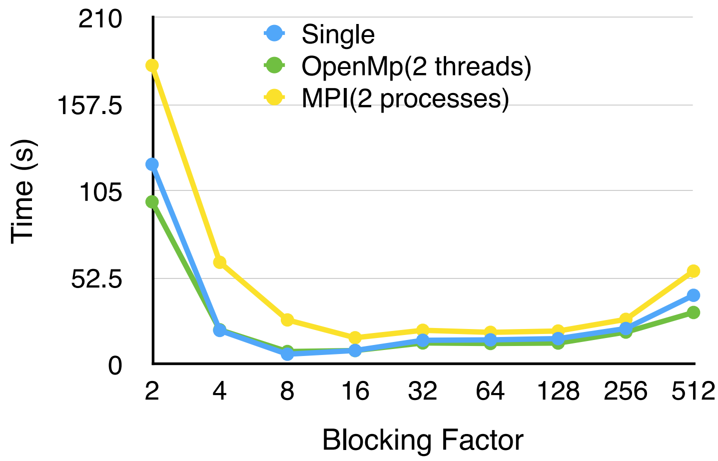
- Bandwidth Calculation:
  - Regard the number of accessing `dev_dist[]` per kernel runtime as the expected bandwidth.
  - For simplicity, an approximation is done by increment the number of accessing `dev_dist[]` by 6 for the distance update section for each calculation.
  - Obtain the bandwidth by multiplying GFLOPS by 6, and by 4 for the unit GB, so the plots of GFLOPS and bandwidth are basically the same.

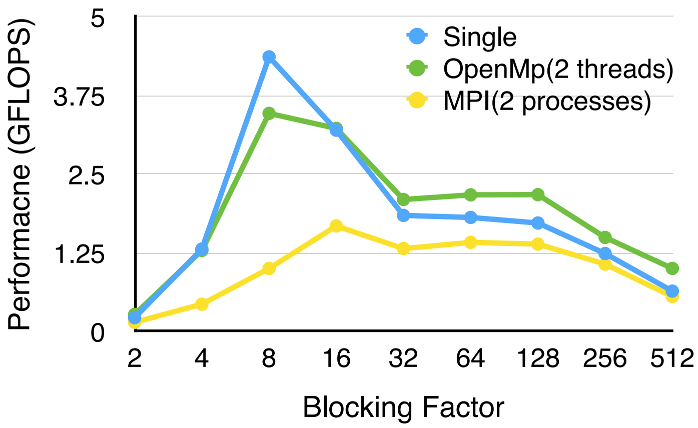
- Execution Time & GFLOPS & Bandwidth v.s. Blocking Factor I
  - Input Setting :

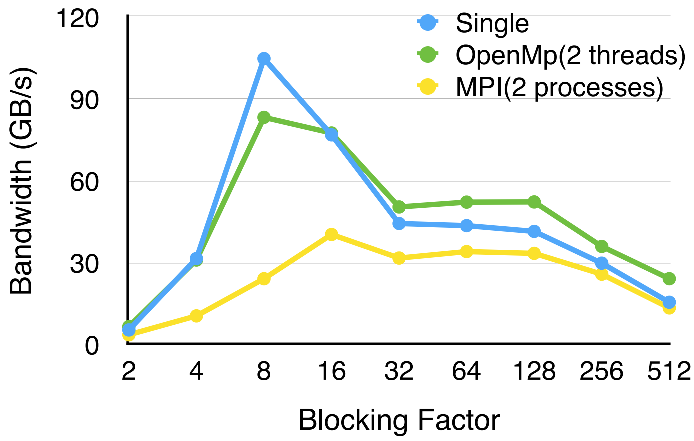
```
block(round, round)
thread(min(32,B), min(32,B))
Testcase in4
OpenMp (2 threads)
MPI (2 processes)
```

  - Results :

	Single	OpenMp (2 threads)	MPI (2 processes)
B=2	121.139329	98.429806	180.994125
B=4	20.567450	20.953182	61.823029
B=8	6.183443	7.804682	26.938044
B=16	8.432997	8.404206	16.154587
B=32	14.611323	12.990598	20.665523
B=64	14.872236	12.680870	19.430720
B=128	15.630576	12.929623	20.223118
B=256	21.735550	19.562106	27.286578
B=512	41.839919	31.440321	56.492201







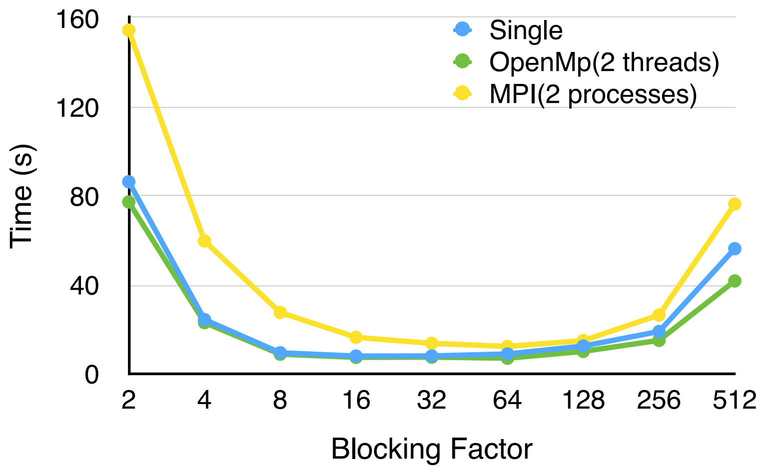
  - Observation :
    (1) For B=8~128, the execution time remains relatively low. When blocking factor is too small (B=2) or too large (B=512), the execution time tends to be longer.
    (2) For B<32, the GFLOPS is hard to reveal conclusion because the dimensions of block and thread vary from B to B. For B≧32, since the dimension of thread is fixed, the tendency is more reasonable.
    (3) To get a clearer view, I fixed the dimensions of block and thread in the next experiment.

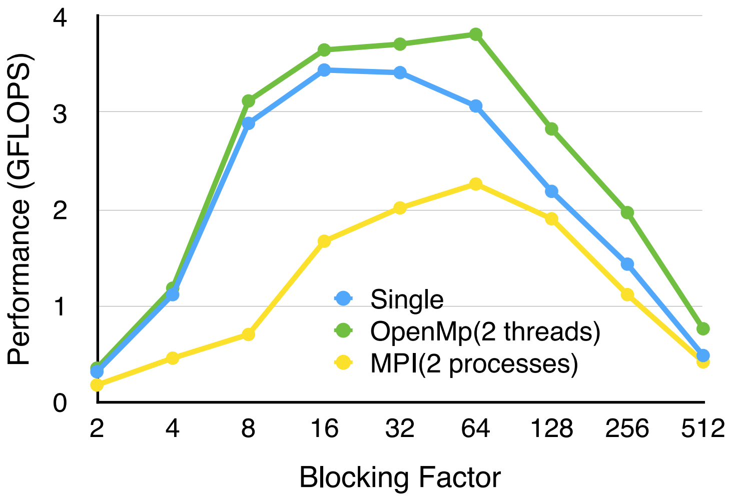
- Execution Time & GFLOPS & Bandwidth v.s. Blocking Factor II
  - Input Setting :

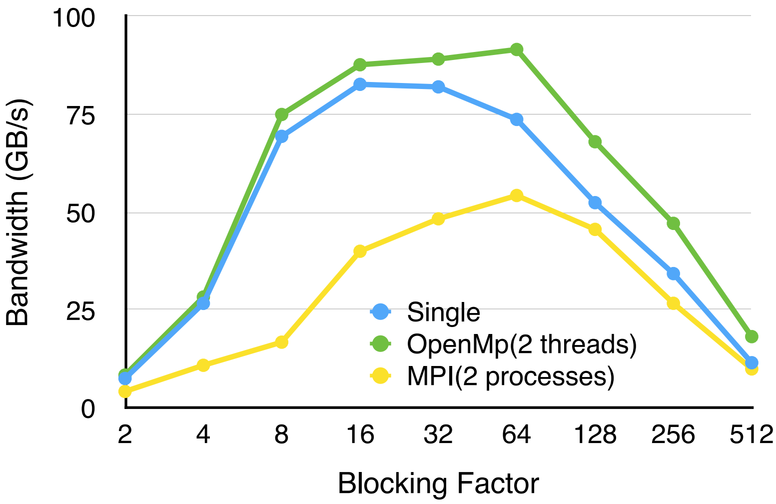
block(10, 10)
thread(10, 10)
Testcase in4
OpenMp (2 threads)
MPI (2 processes)

  - Results :

	Single	OpenMp (2 threads)	MPI (2 processes)
B=2	86.327791	77.210647	154.572869
B=4	24.26173	22.904621	59.636753
B=8	9.336398	8.663839	27.456777
B=16	7.835076	7.248722	16.272898
B=32	7.897341	7.346052	13.551727
B=64	8.787723	6.864621	12.190754
B=128	12.353731	9.932101	14.808557
B=256	18.883344	14.911443	26.315696
B=512	56.171169	41.661553	76.245637







  - Observation :

(1) MPI version requires longer execution time because it suffers from communication between processes. Thus the GFLOPS of MPI version is the lowest among the three.

(2) OpenMp version is the fastest and achieves higher performance than the others. It is only slightly faster than single version because after each round, GPUs have to exchange information by `cudaMemcpy()`.

(3) When B is too small, for example, B=2, the blocking is not effective because in a 3000x3000 matrix, a 2x2 block is almost as small as the non-blocked size. It certainly helps, but not much though. Therefore, when B=2 and 4, a large amount of time is still needed.

(4) As blocking factor grows, more elements are being parallelized, so the execution time decreases.

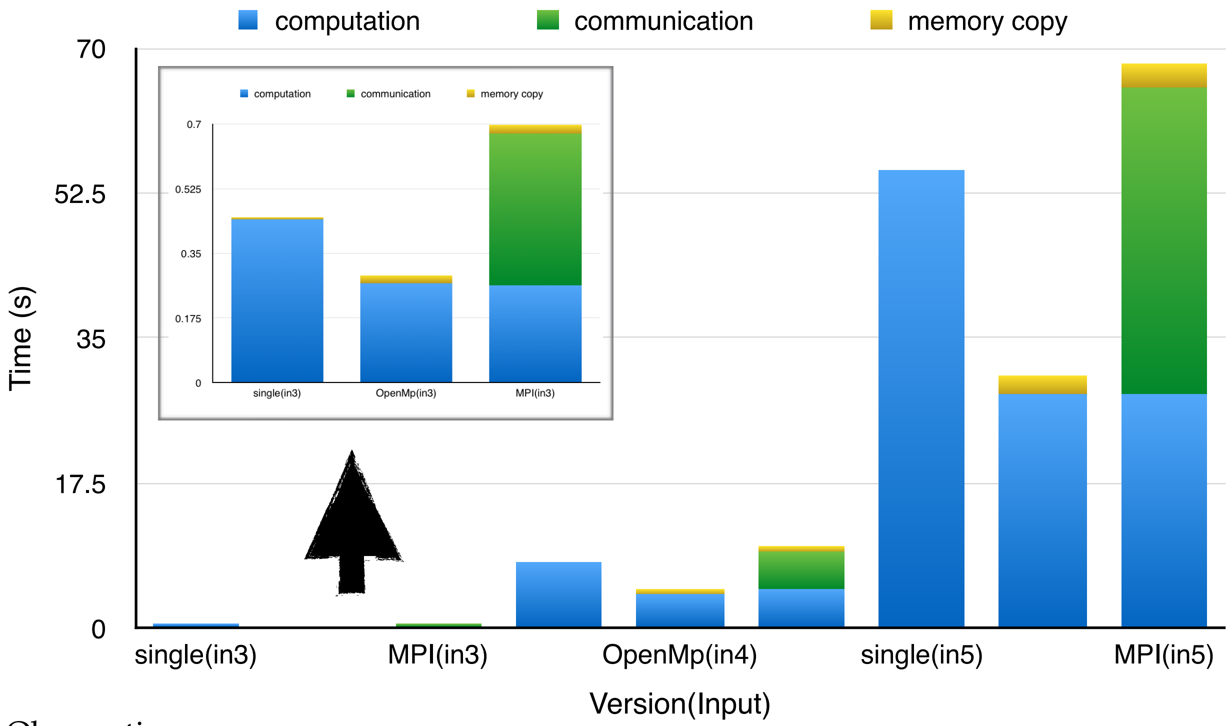
(5) When B is too large, the innate limitation of hardware may cause slowdown. During phase 3, three blocks are to be accessed, but if the block size is too large that all three of such blocks do not fit into the GPU cache, then cache misses occur, which considerably influences the performance.
(Gayathri Venkataraman, Sartaj Sahni, and Srabani Mukhopadhyaya, "A Blocked All-Pairs Shortest-Paths Algorithm")

- Time Distribution

- Input Setting :

```
block(round, round)
thread(min(32,B), min(32,B))
B = 64
OpenMp (2 threads)
MPI (2 processes)
```

- Results :



- Observation :

(1) The computation time of OpenMp and MPI versions is half of that of single version, and this is reasonable because the problem is distributed to 2 GPUs in OpenMp and MPI.

(2) Although MPI can do computation in parallel, it suffers from massive amount of communication overhead that dominates the overall performance. Such overhead leads to the consequences in the previous experiments that MPI version requires longer execution time.

(3) Computing time increases as the input size grows.

(4) Communication time increases as the input size grows because there are more data to be sent and received.

(5) Memory copy time also increases as input size grows because there are more data to be transferred between host and device. But OpenMp and MPI have the same amount of memory copy time because the times of memory copy calls are the same while single version calls `cudaMemcpy()` only twice.

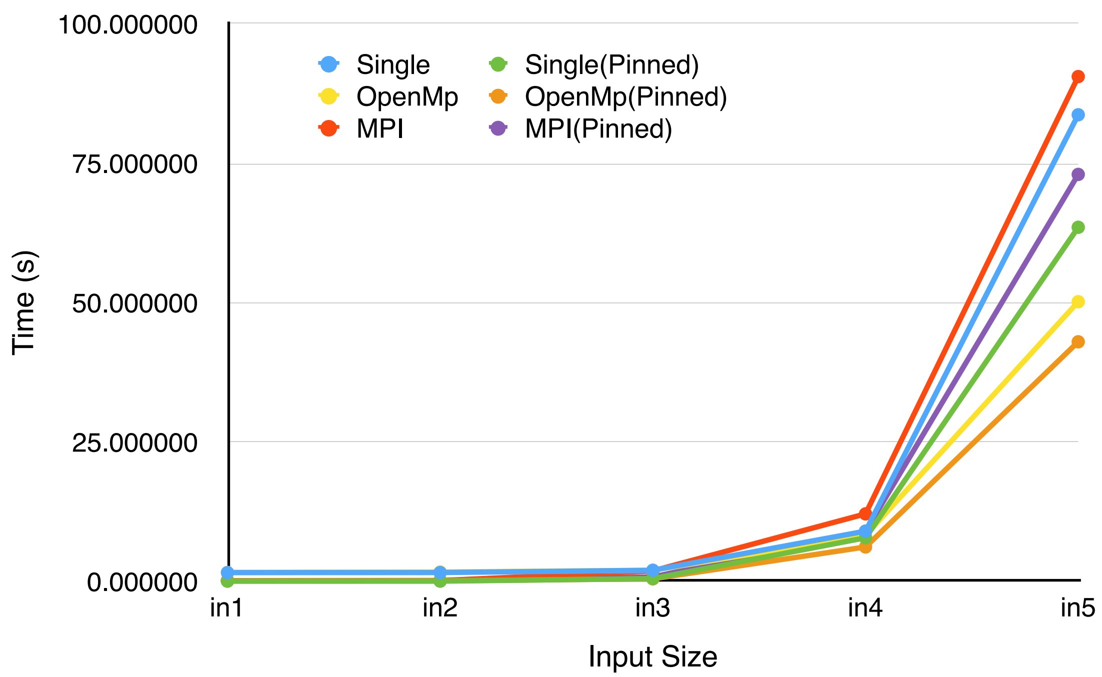
• Weak Scalability & Optimization (Pinned Memory)

- Input Setting :

block(10, 10)
thread(10, 10)
B = 64
OpenMp (2 threads)
MPI (2 processes)

- Results :

	Single	Single (Pinned)	OpenMp	OpenMp (Pinned)	MPI	MPI (Pinned)
in1	1.545760	0.000492	1.562793	0.057982	0.096190	0.094115
in2	1.554278	0.008190	1.652941	0.072133	0.110442	0.110438
in3	1.957913	0.465849	1.953298	0.425449	1.846156	0.765513
in4	8.973730	7.785813	8.200664	6.138620	12.073054	8.966215
in5	83.645468	63.467112	47.118546	42.931893	90.494182	72.952075



- Observation :

(1) Total execution time increases as the input size grows.

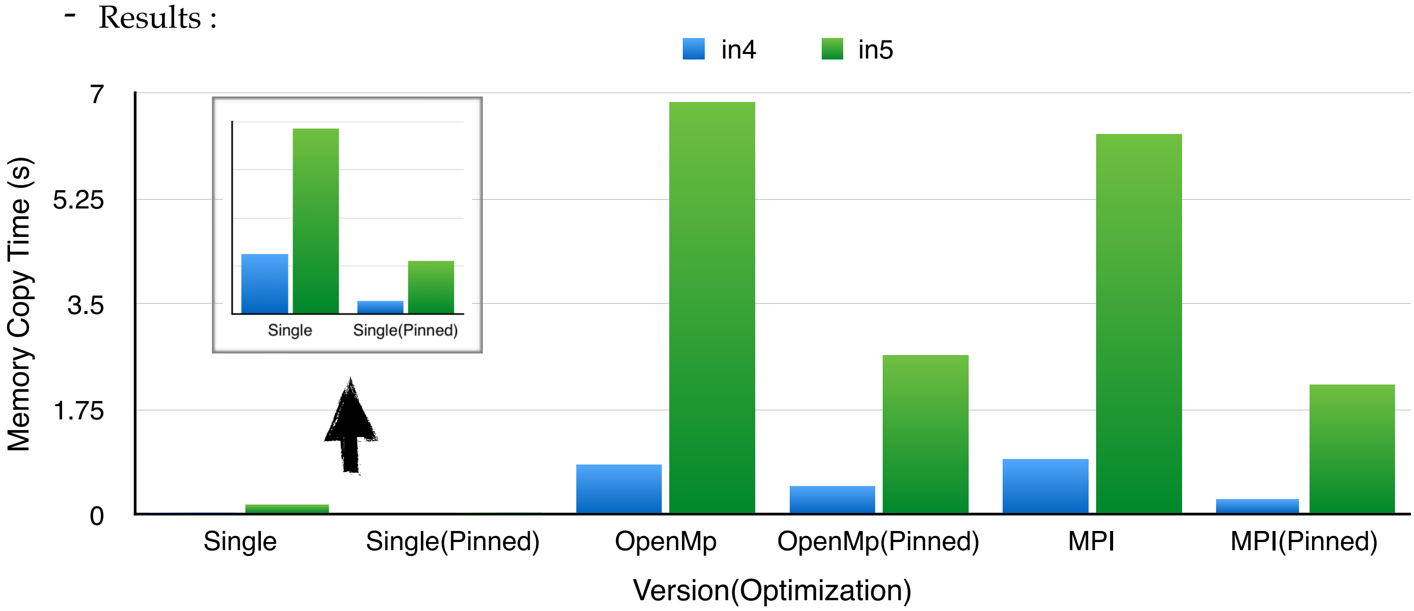
(2) Pinned memory speeds up the execution to certain extent, but the effect of pinned memory is not apparent in perspective of total execution time.

• Optimization (Pinned Memory)

- Input Setting :

block(10, 10)
thread(10, 10)
B = 64
OpenMp (2 threads)
MPI (2 processes)

- Results :



- Observation :

(1) From the perspective of memory, pinned memory improves the memory copy rate considerably. It copies the host memory variables to the pinned array so GPU can access the variable directly through the pinned memory.

(2) For single version, pinned memory approximately triples the memory copy rate.

(3) For OpenMp and MPI version, since they call the same amount of `cudaMemcpy()`, their results are similar.

Experience & Conclusion

Blocking method indeed provides a more efficient way to solve the APSP problem when size is large, but the relationship between number of block, thread, and the value of blocking factor decides the final performance. Blocking factor has its effective range as mentioned in the experiment, while the number of thread also contributes to the performance. Giving too few threads forces the threads to wait for others; giving too many threads makes some threads to be idling, which also affect the utilization of such resource.

In HW4, I learned how to do CUDA programming, and utilize the knowledge acquired from class. The concept of blocks and threads and all the indexing are actually really abstract to me. However, after the blocked APSP assignment, I became more familiar with the idea of CUDA programming. Tools like MPI and OpenMp also become more handy when doing this assignment after all these experience.

Pinned memory optimization puzzles me at first because no matter how many times I tried, the total execution time only decreases by very few seconds, and sometimes the execution even got longer. Then I realized comparison should be done in perspective of memory transfer, which shows more reasonable outcomes.

To me, blocked APSP is not only a brilliant, but also an algorithm that is hard to think of and implement. If the sequential code was not provided, I would still be doing the basic coding part of the assignment.