CS542200 Parallel Programming
Homework 4: Blocked All-Pairs Shortest Path

1st Due: January 10, 2016
2nd Due (with 5 pts off): January 13, 2016
Final Due (with 10 pts off): January 17, 2016

1 GOAL

This assignment helps you get familiar with CUDA on multi-GPU environment by implementing a blocked all-pairs shortest path algorithm. Besides, in order to measure the performance and scalability of your program, experiments are required. Finally, we encourage you to optimize your program by exploring different optimizing strategies for bonus points.

2 PROBLEM DESCRIPTION

In this assignment, you are asked to modify sequential Floyd-Warshall algorithm to a parallelized CUDA version which take advantages of multiple GPUs.

Given an $N \times N$ matrix $W = [w(i,j)]$ where $w(i,j) \geq 0$ represents the distance (weight of the edge) from a vertex i to a vertex j in a ***simple directed graph*** with N vertices. We define an $N \times N$ matrix $D = [d(i,j)]$ where $d(i,j)$ denotes the shortest-path distance from a vertex i to a vertex j. Let $D^{(k)} = \left[d^{(k)}(i,j)\right]$ be the result which all the intermediate vertices are in the set $\{1,2,\cdots,k\}$.

We define $d^{(k)}(i,j)$ as follows:

$$d^{(k)}(i,j) = \begin{cases} w(i,j) & if\ k = 0; \\ \min\left(d^{(k-1)}(i,j), d^{(k-1)}(i,k) + d^{(k-1)}(k,j)\right) & if\ k \geq 1. \end{cases}$$

The matrix $D^{(N)} = d^{(N)}(i,j)$ gives the answer to the APSP problem.

In the blocked APSP algorithm, we partition D into $\lceil N/B \rceil \times \lceil N/B \rceil$ blocks of $B \times B$ submatrices. The number B is called ***blocking factor***. For instance, we divide a 6×6 matrix into 3×3 submatrices (or blocks) by $B = 2$.

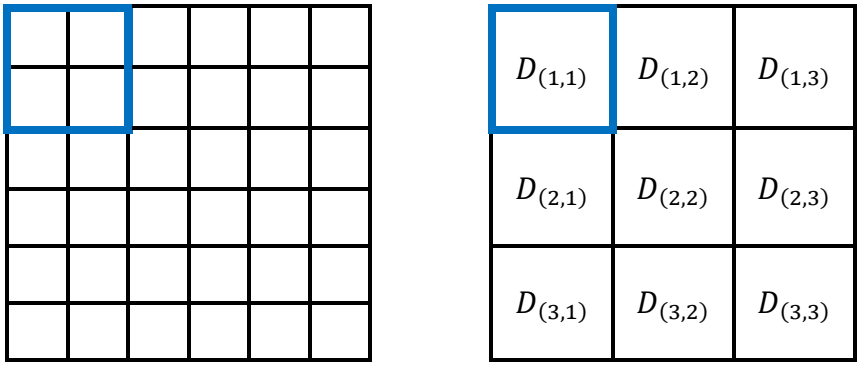


Figure 1: Divide a matrix by $B = 2$

Blocked version of Floyd-Warshall algorithm will perform $\lceil N/B \rceil$ rounds, and each round is divided into 3 phases. It performs B iterations in each phase.

Assumes a block is identified by its index (I, J), where $1 \leq I, J \leq \lceil N/B \rceil$. The block with index (I, J) is denoted by $D^{(k)}_{(I,J)}$.

In the following explanation, we assumes $N = 6$ and $B = 2$. The execution flow is described step by step as follows:

- **Phase 1:** Self-dependent blocks
  In the K-th iteration, the 1st phase is to compute $B \times B$ pivot block $D^{(K \times B)}_{(K,K)}$.
  For instance, in the 1st iteration, $D^{(2)}_{1,1}$ is computed as follows:

$$d^{(1)}(1,1) = \min\left(d^{(0)}(1,1), d^{(0)}(1,1) + d^{(0)}(1,1)\right)$$
$$d^{(1)}(1,2) = \min\left(d^{(0)}(1,2), d^{(0)}(1,1) + d^{(0)}(1,2)\right)$$
$$d^{(1)}(2,1) = \min\left(d^{(0)}(2,1), d^{(0)}(2,1) + d^{(0)}(1,1)\right)$$
$$d^{(1)}(2,2) = \min\left(d^{(0)}(2,2), d^{(0)}(2,1) + d^{(0)}(1,2)\right)$$

$$d^{(2)}(1,1) = \min\left(d^{(1)}(1,1), d^{(1)}(1,2) + d^{(1)}(2,1)\right)$$
$$d^{(2)}(1,2) = \min\left(d^{(1)}(1,2), d^{(1)}(1,2) + d^{(1)}(2,2)\right)$$
$$d^{(2)}(2,1) = \min\left(d^{(1)}(2,1), d^{(1)}(2,2) + d^{(1)}(2,1)\right)$$
$$d^{(2)}(2,2) = \min\left(d^{(1)}(2,2), d^{(1)}(2,2) + d^{(1)}(2,2)\right)$$

  Note that result of $d^{(2)}$ depends on the result of $d^{(1)}$ and therefore cannot be computed in parallel with the computation of $d^{(1)}$.

- **Phase 2:** Pivot-row and pivot-column blocks

  In the K-th iteration, it computes all $D_{(h,K)}^{(K\times B)}$ and $D_{(K,h)}^{(K\times B)}$ where $h \neq K$.

  The result of pivot-row/pivot-column blocks depend on the result in Phase 1 and itself

  For instance, in the 1st iteration, the result of $D_{(1,3)}^{(2)}$ depends on $D_{(1,1)}^{(2)}$ and $D_{(1,3)}^{(0)}$:

$$d^{(1)}(1,5) = \min\left(d^{(0)}(1,5), d^{(2)}(1,1) + d^{(0)}(1,5)\right)$$
$$d^{(1)}(1,6) = \min\left(d^{(0)}(1,6), d^{(2)}(1,1) + d^{(0)}(1,6)\right)$$
$$d^{(1)}(2,5) = \min\left(d^{(0)}(2,5), d^{(2)}(2,1) + d^{(0)}(1,5)\right)$$
$$d^{(1)}(2,6) = \min\left(d^{(0)}(2,6), d^{(2)}(2,1) + d^{(0)}(1,6)\right)$$

$$d^{(2)}(1,5) = \min\left(d^{(1)}(1,5), d^{(2)}(1,2) + d^{(1)}(2,5)\right)$$
$$d^{(2)}(1,6) = \min\left(d^{(1)}(1,6), d^{(2)}(1,2) + d^{(1)}(2,6)\right)$$
$$d^{(2)}(2,5) = \min\left(d^{(1)}(2,5), d^{(2)}(2,2) + d^{(1)}(2,5)\right)$$
$$d^{(2)}(2,6) = \min\left(d^{(1)}(2,6), d^{(2)}(2,2) + d^{(1)}(2,6)\right)$$

- **Phase 3:** Other blocks

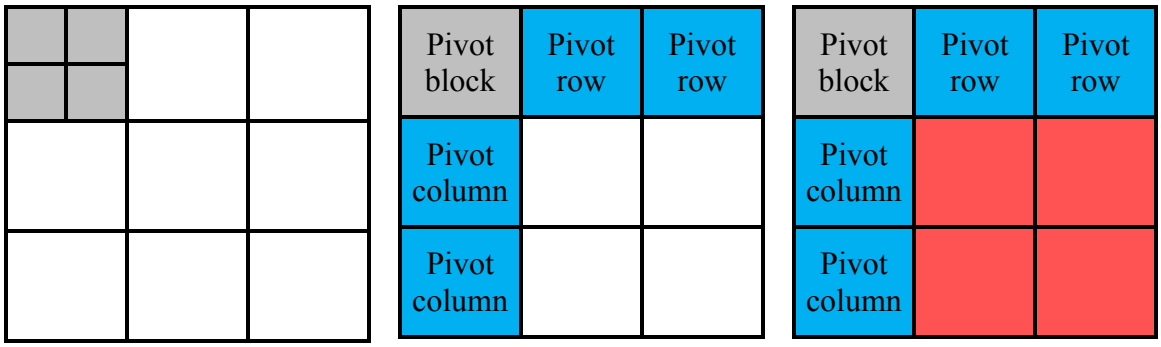
  In the K-th iteration, it computes all $D_{(h_1,h_2)}^{(K\times B)}$ where $h_1, h_2 \neq K$.

  The result of these blocks depend on the result in Phase 2 and itself.

  For instance, in the 1st iteration, the result of $D_{(2,3)}^{(2)}$ depends on $D_{(2,1)}^{(2)}$ and $D_{(1,3)}^{(2)}$:

$$d^{(1)}(3,5) = \min\left(d^{(0)}(3,5), d^{(2)}(3,1) + d^{(2)}(1,5)\right)$$
$$d^{(1)}(3,6) = \min\left(d^{(0)}(3,6), d^{(2)}(3,1) + d^{(2)}(1,6)\right)$$
$$d^{(1)}(4,5) = \min\left(d^{(0)}(4,5), d^{(2)}(4,1) + d^{(2)}(1,5)\right)$$
$$d^{(1)}(4,6) = \min\left(d^{(0)}(4,6), d^{(2)}(4,1) + d^{(2)}(1,6)\right)$$

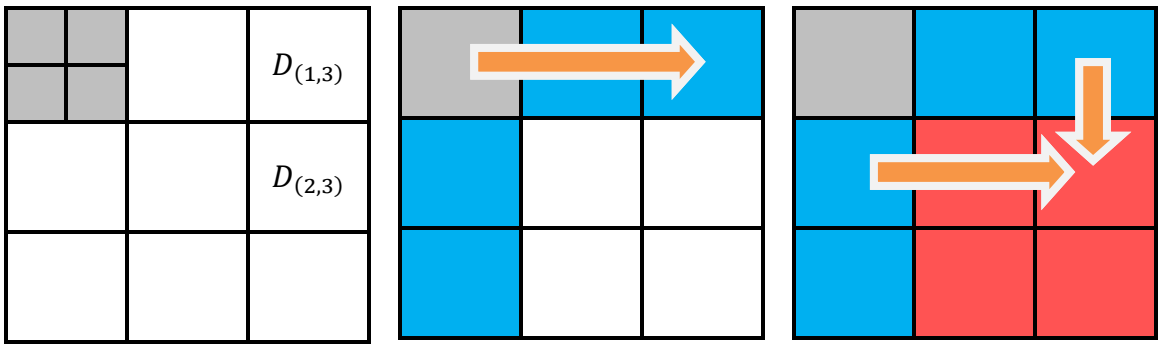
$$d^{(2)}(3,5) = \min\left(d^{(1)}(3,5), d^{(2)}(3,2) + d^{(2)}(2,5)\right)$$
$$d^{(2)}(3,6) = \min\left(d^{(1)}(3,6), d^{(2)}(3,2) + d^{(2)}(2,6)\right)$$
$$d^{(2)}(4,5) = \min\left(d^{(1)}(4,5), d^{(2)}(4,2) + d^{(2)}(2,5)\right)$$
$$d^{(2)}(4,6) = \min\left(d^{(1)}(4,6), d^{(2)}(4,2) + d^{(2)}(2,6)\right)$$



(a) Phase 1 (b) Phase 2 (c) Phase 3

Figure 2: The 3 phases of blocked FW algorithm in the 1st iteration

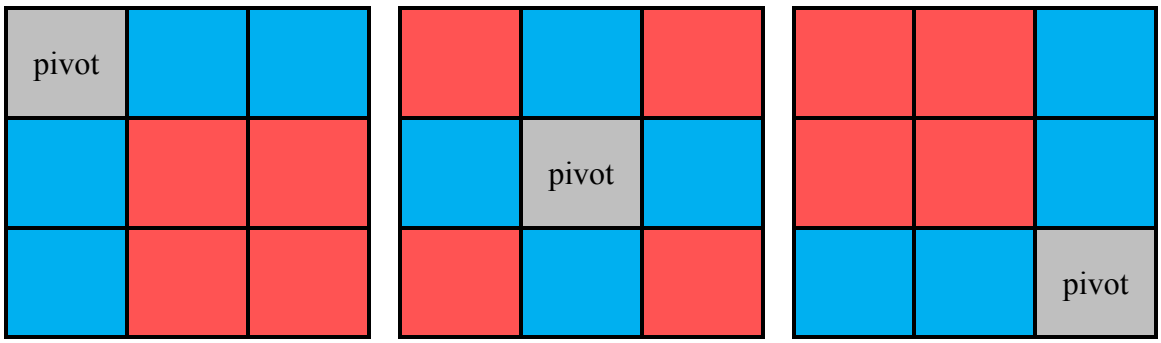
The computations of $D^{(2)}_{(1,3)}, D^{(2)}_{(2,3)}$ and its dependencies are illustrated in Figure 3.



(d) Phase 1 (e) Phase 2 (f) Phase 3

Figure 2: Dependencies of $D^{(2)}_{(1,3)}, D^{(2)}_{(2,3)}$ in the 1st iteration

In this particular example where $N = 6$ and $B = 2$, we will require $\lceil N/B \rceil = 3$ rounds.



(a) Round 1 (b) Round 2 (c) Round 3

Figure 4: Blocked FW algorithm in each iteration

3 INPUT / OUTPUT FORMAT

1. Your program are required to read an input file, and generate output in another file.

2. Your program accepts 2 input parameters. They are:

   i、 (String) the input file name
   ii、 (String) the output file name
   iii、 (Integer) the blocking factor

   Make sure users can assign test cases through command line. For instance:

```
[s104012345@pp01 ~]# ./executable in_file out_file 32
```

   TAs will judge your program as follows:

```
[s104012345@pp01 ~]# diff -b out_file answer
```

3. The 1st line of a input test case consists of 2 integers $\boldsymbol{N}$ ($1 \leq N \leq 10000$) and $\boldsymbol{M}$ ($0 \leq M \leq 10^9$) separated by a single space, which represents number of vertices and number of edge weight assignments respectively.

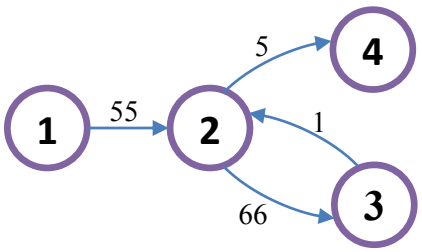
   Each of the following M lines consists of 3 integers $\boldsymbol{i}, \boldsymbol{j}$ and $\boldsymbol{W}$ ($i \neq j$), separated by a single space between any two numbers.

   - $\boldsymbol{i}$ represents the index of the source vertex ($1 \leq i \leq N$)
   - $\boldsymbol{j}$ represents the index of the destination vertex ($1 \leq j \leq N$)
   - $\boldsymbol{W}$ represents the distance (weight of edge) from vertex i to vertex j ($0 \leq W \leq 100$)

   Edges which are not listed in the input file do not exist in the graph. That is, for all $i \neq j$, if edge(i, j) does not show up in the input at all, vertex i does not have an edge to vertex j. But since we are dealing with a ***directed*** graph, this does NOT imply that edge(j, i) is also non-existent.

   Besides, if there are re-assignments of an edge, please follow the latest one.

```
[s104012345@pp01 ~]# cat testcase
4 4
1 2 55
2 3 66
3 2 1
2 4 5
```



(a) Content of a sample input file (b) The corresponding graph

Figure 5: Sample Input

4. For output file, list the shortest-path distance of all vertex pairs.

   Assume N represents total number of vertices, the output file should consists of N lines, each line consists of N numbers and separate them by a single space.

   The number at the i^{th} line and the j^{th} column is the shortest-path distance from the i^{th} vertex to the j^{th} vertex if there is a path; otherwise, the corresponding output should be **INF**.

```
[s104012345@pp01 ~]# cat output
0 55 121 60
INF 0 66 5
INF 1 0 6
INF INF INF 0
```

Figure 6: Sample output

The sample test cases are provided in **/home/cs542200/user0/shared/** on gpucluster.

4 WORKING ITEMS

You are required to implement 3 versions of blocked Floyd-Warshall algorithm under the given restrictions.

1. **Single-GPU**
   - Implement blocked APSP algorithm as described in Section 2.
   - The main algorithm should be implemented in CUDA C/C++ kernel functions.
   - Achieve better performance than sequential Floyd-Warshall implementation.
2. **Multi-GPU implementation with OpenMP**
   - The restrictions of single-GPU version still hold.
   - Able to utilize multiple GPUs available on a single node.
   - Achieve better performance than single GPU version.
3. **Multi-GPU implementation with MPI**
   - The restrictions of single-GPU version still hold.
   - Able to utilize multiple GPUs available on multiple nodes.
   - Achieve better performance than single GPU version.
4. **Makefile**
   Please refer to some examples in **/home/cs542200/user0/shared/** on gpucluster.

5. Report

- **Implementation**

  (a) How do you divide your data?

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

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

  Briefly describe your implementation in diagrams, figures or sentences.

- **Profiling Results**
  Provide the profiling results using profiling tools. (e.g. nvprof, nvvp)
  It's better to use print-screen & paste as results on your report.

- **Experiment & Analysis**
  How and why you do these experiments? The result of your experiments, and try to explain them. (put some figures and charts)

  Note that we have different type of GPUs in our cluster, please indicate which GPUs are used to run these experiments in each chart.

  You can use the time measured by profiling tools. (Please refer to Lab 2 slides)

  i、 System Spec
  If you run your experiments on your own machine, please attach CPU, GPU, RAM, disk and network (Ethernet / InfiniBand) information of the system.

  ii、 Weak Scalability & Time Distribution
  Observe weak scalability of the three implementations. Moreover, analyze the time spent in **1) *computing***, **2) *communication***, **3) *memory copy*** and **4) *I/O*** of your program w.r.t. input size.

  You should explain how you measure these time in your program, and compare the time distribution under different configurations.

  ※ We encourage you to generate your own test cases!

  iii、 Blocking Factor
  Observe what happened with different blocking factor, and plot the trend in terms of GFLOPS and device/shared memory bandwidth.

  Please refer to Figure 6 and 7 as examples.

  iv、 Others
  Additional charts with explanation and studies. The more, the better.

- **Experience / Conclusion**

Note 1: **Please make sure you have read the working items carefully.**
Do NOT just scroll down to this page and start writing your report!

Note 2: **The numbers in the following charts may not come from real data.**
Do NOT compare them with your own results!

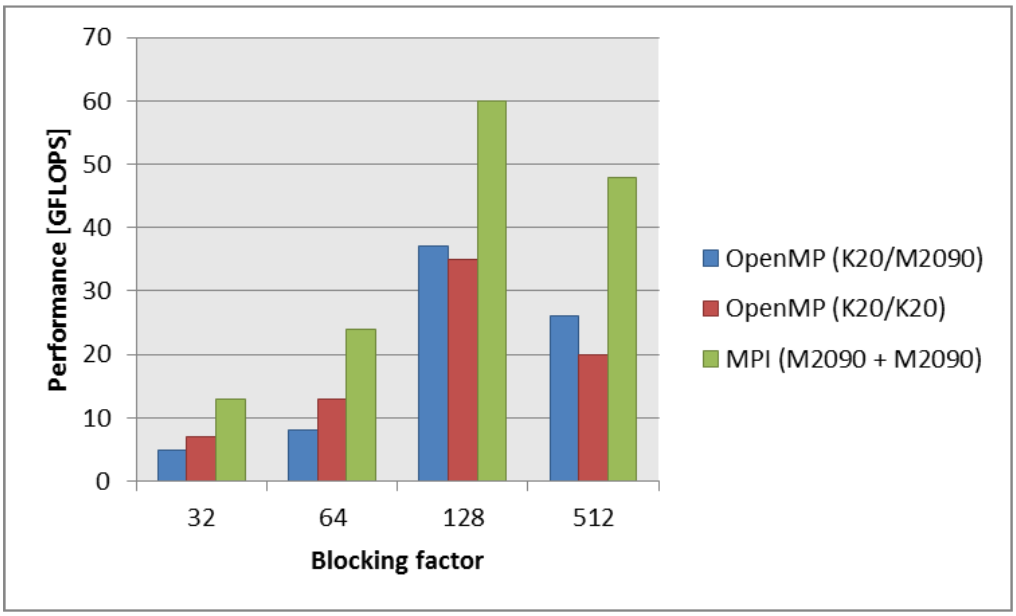


Figure 6: Example chart of performance trend w.r.t. blocking factor

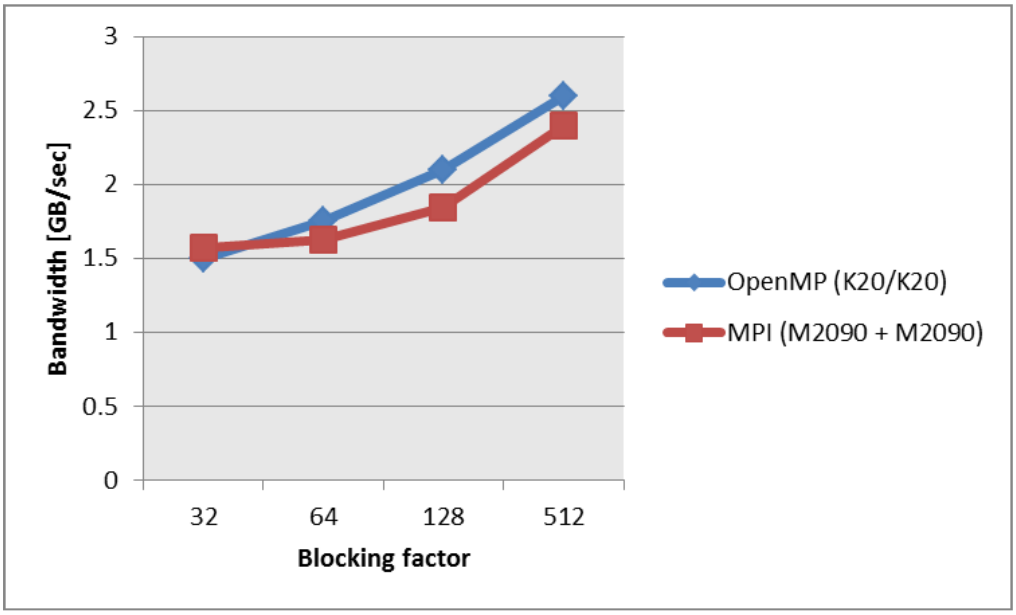


Figure 7: Example chart of memory bandwidth trend w.r.t. blocking factor

5 GRADING

1. **Correctness** (48%)

   i、 [16%] Single-GPU

   ii、 [16%] Multi-GPU implementation with OpenMP

   iii、[16%] Multi-GPU implementation with MPI

2. **Report** (32%)

   Grading is based on your evaluation results, discussion and writing.
   If you want to get more points, design as more experiments as you can.

   N different extra experiments ➔ $\lceil 3\log_2(N+1) \rceil$ extra points

   Do NOT put your charts without explanations. In such cases, we will not count these chart while grading your report.

3. **Demo** (20%)

4. **Bonus** (15%)

   1st optimization technique gets 6 points.
   2nd optimization technique gets 5 points.
   3rd optimization technique gets 4 points.

   Trivial techniques such as changing compiler flags, tuning configurations (blocking factor / block size of kernel), use a better GPU, simply adding #pragma unroll … are all **unacceptable**.

   Some example techniques are listed as follows:

   (a) Shared memory (well-suited for this problem)

   (b) Streaming

   (c) Dynamic load-balancing (can only be implemented in multi-GPU versions)

   (d) Resolve bank conflicts

   (e) Others (TA will judge in demo)

   These optimization techniques should be implemented in all 3 versions. Each technique should achieve better performance compared to the one without it under the best possible configuration. **(Prove it with some experiments)**

   For instance, the 1st + 2nd +3rd techniques optimized code should achieve better performance than the one with only 1st + 2nd optimization techniques.

5. **Total Points** $= \mathbf{min}\big((1)+(2)+(3)+(4), 100\big)$ **− (Late Submission Penalty)**

6 REMINDER

1. Please package your codes and report in a file named **HW4_{student-ID}.zip** which contains:

   i、 HW4_{student-ID}_cuda.c (or .cpp)

   ii、 HW4_{student-ID}_openmp.c (or .cpp)

   iii、HW4_{student-ID}_mpi.c (or .cpp)

   iv、HW4_{student-ID}_report.pdf

   v、 Makefile

   And upload to iLMS before **1/10/2016 (Sun) 23:59**

2. Since we have limited resources for you guys to use, please start your work ASAP. Do not leave it until the last day!

3. Late submission penalty policy please refer to the course syllabus. **But note that this time we will not accept any further submission after 1/17/2016 (Sun) 23:59**

4. Asking questions through iLMS or email are welcomed!