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

Revision 3

1st Due: December 19, 2018
2nd Due (with 5 pts off): December 22, 2018
Final Due (with 10 pts off): December 26, 2018

1 GOAL

This assignment helps you get familiar with CUDA on the 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 optimization points.

2 PROBLEM DESCRIPTION

In this assignment, you are asked to modify the sequential Floyd-Warshall algorithm to a parallelized CUDA version which takes 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 ***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 the ***blocking factor***. For instance, we divide a 6×6 matrix into 3×3 submatrices (or blocks) by $B = 2$.

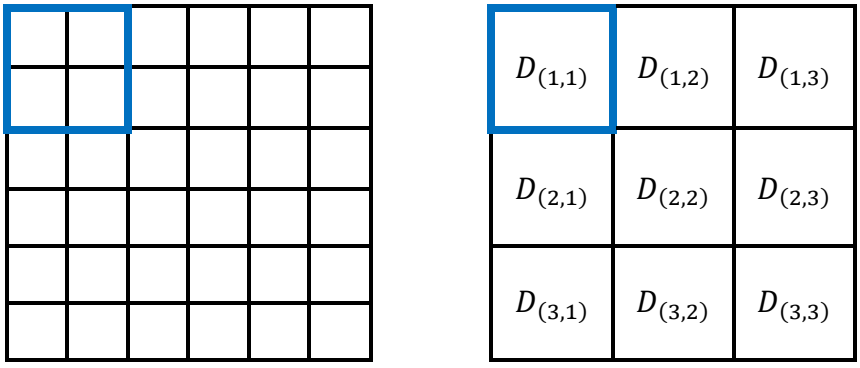


Figure 1: Divide a matrix by $B = 2$

The blocked version of the 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 assume $N = 6$ 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 $D_{(1,3)}^{(0)}$:

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

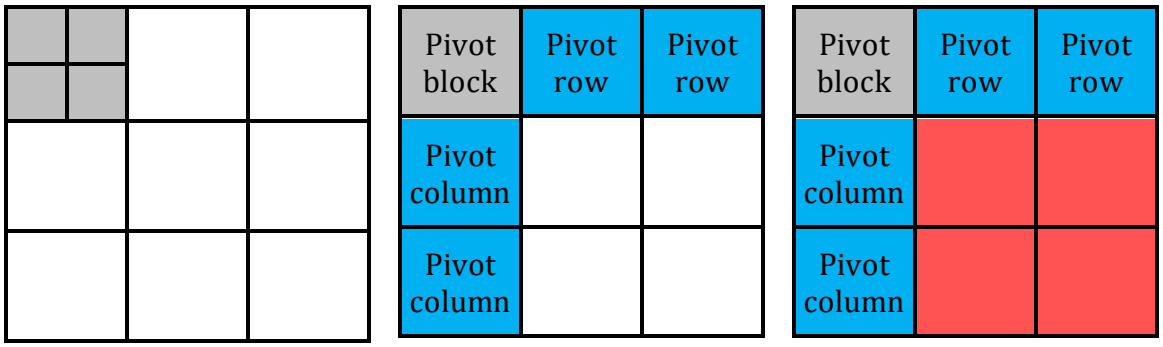
$$d^{(2)}(1,5) = \min\left(d^{(1)}(1,5), d^{(2)}(1,2) + d^{(1)}(2,5)\right)$$
$$d^{(2)}(1,6) = \min\left(d^{(1)}(1,6), d^{(2)}(1,2) + d^{(1)}(2,6)\right)$$
$$d^{(2)}(2,5) = \min\left(d^{(1)}(2,5), d^{(2)}(2,2) + d^{(1)}(2,5)\right)$$
$$d^{(2)}(2,6) = \min\left(d^{(1)}(2,6), d^{(2)}(2,2) + d^{(1)}(2,6)\right)$$

- **Phase 3:** Other blocks
  In the K-th iteration, it computes all $D^{(K\times B)}_{(h_1,h_2)}$ where $h_1, h_2 \neq K$.
  The result of these blocks depends on the result in Phase 2 and itself.

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

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

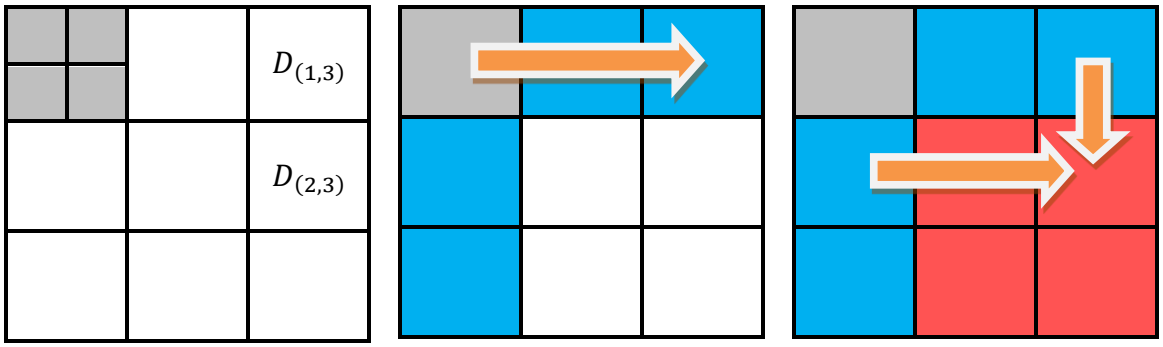
$$d^{(2)}(3,5) = \min\left(d^{(1)}(3,5), d^{(2)}(3,2) + d^{(2)}(2,5)\right)$$
$$d^{(2)}(3,6) = \min\left(d^{(1)}(3,6), d^{(2)}(3,2) + d^{(2)}(2,6)\right)$$
$$d^{(2)}(4,5) = \min\left(d^{(1)}(4,5), d^{(2)}(4,2) + d^{(2)}(2,5)\right)$$
$$d^{(2)}(4,6) = \min\left(d^{(1)}(4,6), d^{(2)}(4,2) + d^{(2)}(2,6)\right)$$



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

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

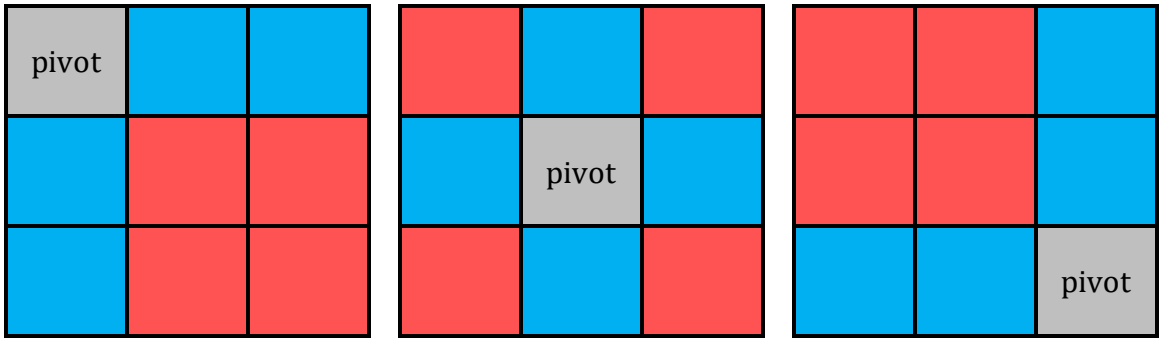
The computations of $D^{(2)}_{(1,3)}, D^{(2)}_{(2,3)}$ and its dependencies are illustrated in Figure 3.



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

Figure 3: 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

3.1 INPUT PARAMETERS

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

Your program accepts **2** input parameters,

(String) the input file name
(String) the output file name

Example

```
$ ./executable <input> <output>
```

3.2 INPUT FILE FORMAT

- The input is a binary file containing a directed graph with non-negative edge weight.
- Here is an example:

[offset]	[type]	[decimal value]	[description]
0000	32-bit integer	3	# vertices (V)
0004	32-bit integer	6	# edges (E)
0008	32-bit integer	0	Src id for edge 0
0012	32-bit integer	1	Dst id for edge 0
0016	32-bit integer	3	Weight on edge 0
0020	32-bit integer		Src id for edge 1
...	...	...	...
0076	32-bit integer		Weight on edge 5

- The first two integers mean {number of vertices} and {number of edges}
- After that, vertices and edges is a list of {source vertex id} {destination vertex id} {edge weight}
- The edge weights are non-negative integers. **The values of vertex id start from 0**.

Notice that:

- ✓ **$2 \le V \le 20000$ for single-GPU version**
- ✓ **$2 \le V \le 50000$ for multi-GPU version**
- ✓ **$V - 1 \le E \le 400000000$**
- ✓ **$0 \le src, dst < V$**
- ✓ **$0 \le W \le 100$**
- ✓ **No need to consider the condition of signed integer overflow**

3.3 Output File Format

- The output file is also binary format, and the details are listed below.
- you should output N^2 integers which are the shortest path distances between each pair of vertices.

[offset]	[type]	[decimal value]	[description]
0000	32-bit integer	0	*min* Dist(0, 0)
0004	32-bit integer	**Do it yourself**	*min* Dist(0, 1)
0008	32-bit integer	**Do it yourself**	*min* Dist(0, 2)
...	...	...	...
xxxx-4	32-bit integer	**Do it yourself**	*min* Dist(N-1, N-2)
xxxx	32-bit integer	0	*min* Dist(N-1, N-1)

- The output records must be **sorted by vertex id**
- **Distance (*i*, *j*) = 0, where *i* = *j***
- **If there is no path between i, j, please set the distance to 1000000000**

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 in the single node**

   - The restrictions of single-GPU version still hold.
   - Able to utilize multiple GPUs available on the single node.
   - Achieve better performance than the single GPU version.
   - We will allocate 2 cores in this cases when judging, so you can implement it with OpenMP / CUDA stream, etc.
   - Only need to handle 2 devices.

3. **Multi-GPU implementation with MPI**

   - The restrictions of single-GPU version still hold.
   - Able to utilize multiple GPUs available on multi-node.
   - Achieve better performance than the single GPU version
   - Only need to handle total 2 devices. (one card each node)
   - Example cmd: srun –n 2 –ppp --gres=gpu:1 ./exe
   - **Output your results by rank 0**
     (We judge the code on ramdisk to reduce the I/O overhead)

4. **Makefile**
   Please refer to the example in **/home/pp18/shared/hw4** on **hades01**

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? And why?
     (e.g. blocking factor, #blocks, #threads)

     Briefly describe your implementation in diagrams, figures and 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)

  You can use the time measured by profiling tools. (Please refer to Lab 3, 4 slides)

  1. **System Spec**

     Please attach CPU, GPU, RAM, disk, and network (Ethernet / InfiniBand) information of the system.

  2. **Weak Scalability & Time Distribution**

     Observe weak scalability of the Multi-GPU implementations. Moreover, analyze the time spent in

     1) computing

     2) communication

     3) memory copy

     4) I/O of your program w.r.t. input size.

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

     ※ We encourage you to generate your own test cases!

  3. **Blocking Factor**

     Observe what happened with different blocking factors, and plot the trend in terms of Integer GOPS and device/shared memory bandwidth. (You can get the information from profiling tools or manual)

     Please refer to Figure 6 and 7 as examples.

  4. **Optimization**

Any optimizations after you port the algorithm on gpu, descript them with sentences and charts. Here are some techniques you can implement:

- ✓ Deal with memory bank conflict
- ✓ Shared memory
- ✓ CUDA 2D alignment
- ✓ Occupancy optimization
- ✓ Streaming
- ✓ Unroll
- ✓ Reduce communication

And you should summarize all optimizations in a chart

5. **Others**

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

- **Experience / Conclusion**

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

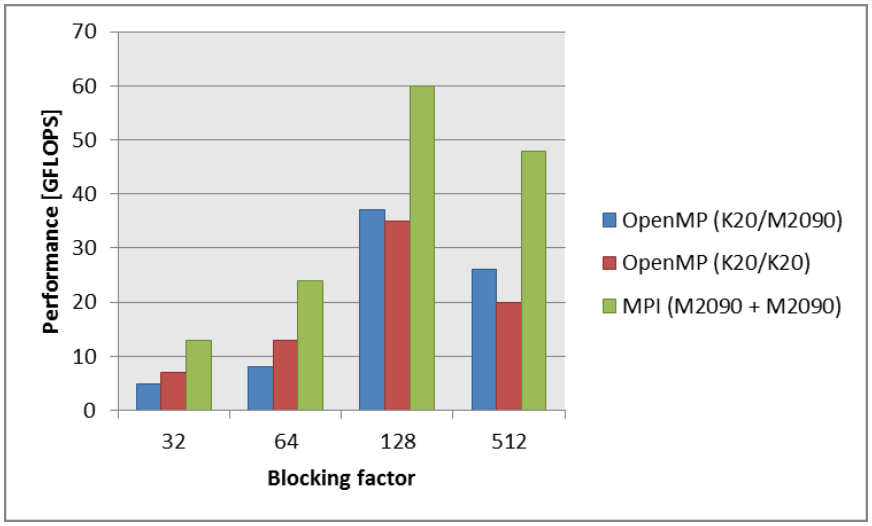


Figure 6: Example chart of performance trend w.r.t. blocking factor

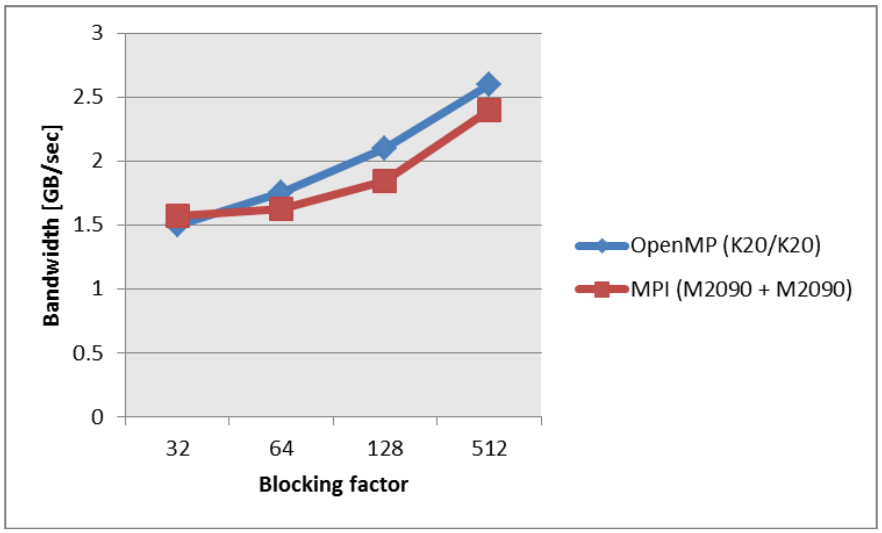


Figure 7: Example chart of memory bandwidth trend w.r.t. blocking factor

5 GRADING

1. **Correctness** (35%)
   - ✓ [15%] Single-GPU
   - ✓ [10%] Multi-GPU implementation in the single node
   - ✓ [10%] Multi-GPU implementation with MPI
2. **Performance** (30%)
   See Appendix A for the detail grading policy
   - ✓ [10%] Single-GPU
   - ✓ [10%] Multi-GPU implementation in the single node
   - ✓ [10%] Multi-GPU implementation with MPI
3. **Report** (20%)

   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 ➔ $\lceil 4\log_2(N+1) \rceil$ extra **total** points

   Do NOT put your charts without explanations. In such cases, we will not count these chart while grading your report.
4. **Demo** (15%)
5. **Total Points** $= \mathbf{min}\big((\mathbf{1}) + (\mathbf{2}) + (\mathbf{3}) + (\mathbf{4}) + (\mathbf{5}),\ \mathbf{100}\big)$
6. We grade your scores by **hw4-judge** and **hidden testcases,** make sure your programs can get correct results on all possible testcases.
7. If you are not satisfied with the performance scores, do more experiments for extra points.

6 REMINDER

1. Please submit your code and report to ~/homework/hw4 on hades01 and ilms (**DO NOT package them**):
   - ✓ **apsp.cu**
   - ✓ **mutli_gpu.cu**
   - ✓ **multi_node.cu**
   - ✓ **{student-ID}_report.pdf**
   - ✓ **Makefile**

Note:

- ✓ Your Makefile must be able to build the corresponding targets of the implementations: apsp, multi_gpu, multi_node. If you're unsure how to write a Makefile, you can use the provided example Makefile as-is.
- ✓ Your submission time for your source code will be based on the time on hades01 and your submission time for your report will be based on iLMS. **DO NOT** touch the source code after the deadline.

2. Since we have limited resources for you to use, please start your work ASAP. Do not leave it until the last day!
3. Asking questions through iLMS is welcomed!

APPENDIX A

We will use several different test cases (denoted C) to run your code.

Your performance score will be given by:

$$\sum S(C) \times \frac{T_{best}(C)}{T_{yours}(C)}$$

- C is a test case with different settings
- $S(C)$ is the score allocated to the test case.
- $T_{best}(C)$ is the shortest execution time of all students for the test case, excluding incorrect implementations.
- $T_{yours}(C)$ is your execution time for the test case, excluding incorrect implementations.

Refer ilms