CS542200 Parallel Programming
Homework 4-1: Blocked All-Pairs Shortest Path

Due: December 16, 2019

1 Goal

This assignment helps you get familiar with CUDA by implementing a blocked all-pairs shortest path algorithm. Besides, 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.

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 $\left\lceil\frac{N}{B}\right\rceil \times \left\lceil\frac{N}{B}\right\rceil$ 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 $B = 2$.

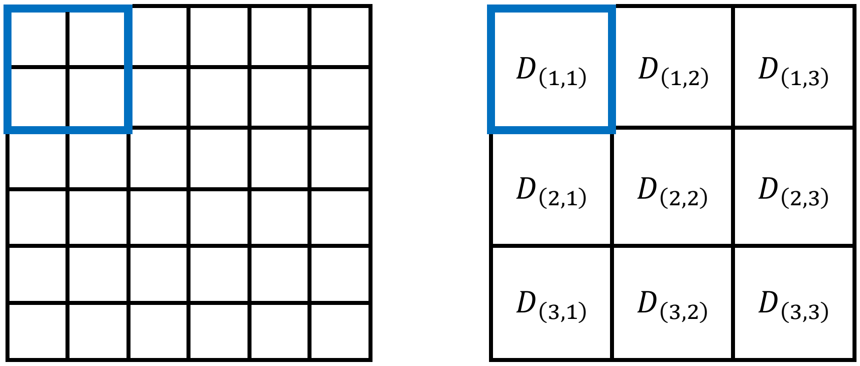


Figure 1: Divide a matrix by $B = 2$

The blocked version of the Floyd-Warshall algorithm will perform $\left\lceil \frac{N}{B} \right\rceil$ 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 $1 \leq I, J \leq \left\lceil \frac{N}{B} \right\rceil$. The block with index (I, J) is denoted by $D_{(I,J)}^{(k)}$.

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 $D_{(K,K)}^{(K \times B)}$.

  For instance, in the 1st iteration, $D_{1,1}^{(2)}$ 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/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 $D_{(1,3)}^{(0)}$:

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

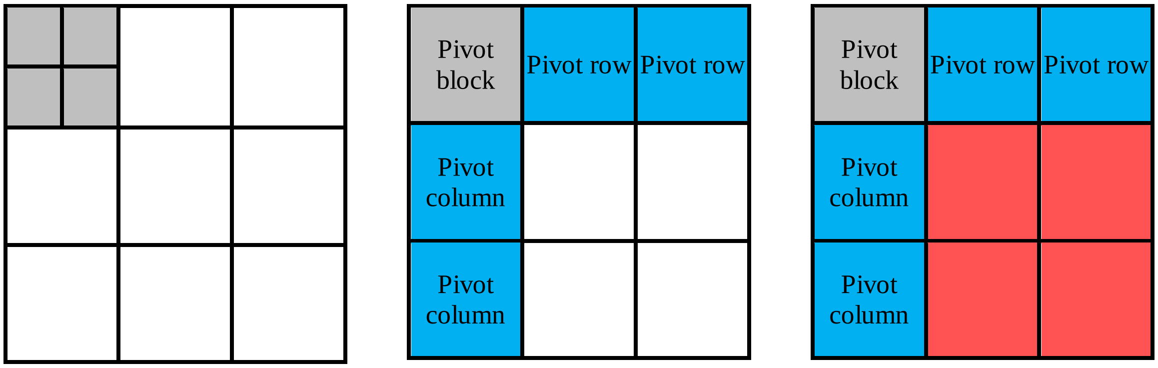
$$d^{(2)}(1,5) = \min\left(d^{(1)}(1,5), d^{(2)}(1,2) + d^{(1)}(2,5)\right)$$
$$d^{(2)}(1,6) = \min\left(d^{(1)}(1,6), d^{(2)}(1,2) + d^{(1)}(2,6)\right)$$
$$d^{(2)}(2,5) = \min\left(d^{(1)}(2,5), d^{(2)}(2,2) + d^{(1)}(2,5)\right)$$
$$d^{(2)}(2,6) = \min\left(d^{(1)}(2,6), d^{(2)}(2,2) + d^{(1)}(2,6)\right)$$

- 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 depends 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 $D_{(1,3)}^{(2)}$:

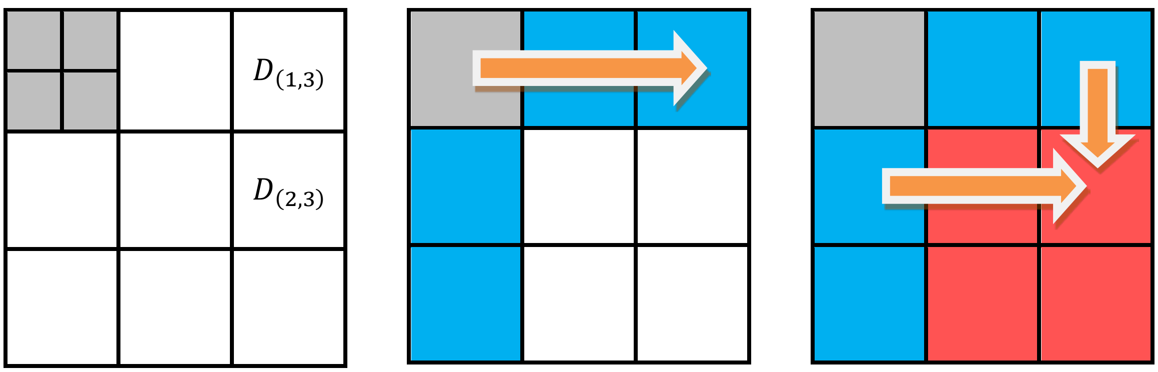
$$d^{(1)}(3,5) = \min\left(d^{(0)}(3,5), d^{(2)}(3,1) + d^{(2)}(1,5)\right)$$
$$d^{(1)}(3,6) = \min\left(d^{(0)}(3,6), d^{(2)}(3,1) + d^{(2)}(1,6)\right)$$
$$d^{(1)}(4,5) = \min\left(d^{(0)}(4,5), d^{(2)}(4,1) + d^{(2)}(1,5)\right)$$
$$d^{(1)}(4,6) = \min\left(d^{(0)}(4,6), d^{(2)}(4,1) + d^{(2)}(1,6)\right)$$

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



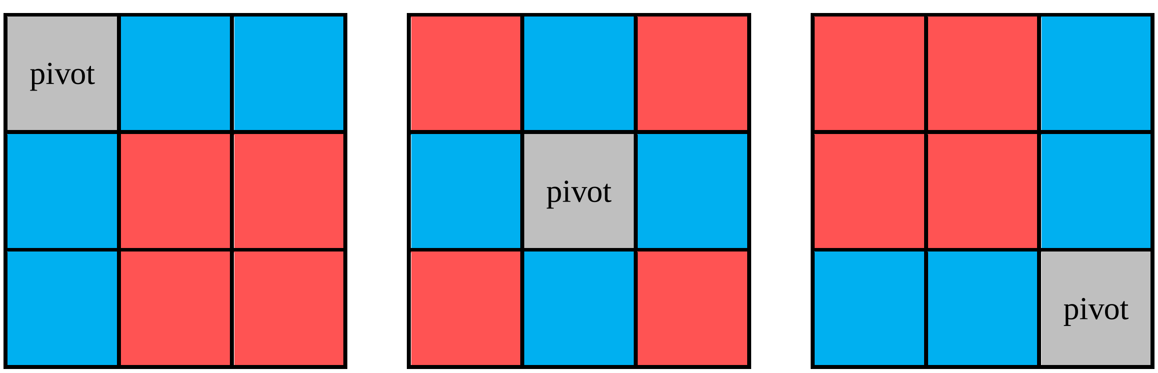
(a) Phase 1 (b) Phase 2 (c) Phase 3

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



(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



(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	source id for edge 0
0012	32-bit integer	1	destination id for edge 0
0016	32-bit integer	3	weight on edge 0
0020	32-bit integer		source 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, every consecutive 3 integers denote an edge, which is composed with <*source vertex*>, <*destination vertex*>, and <*edge weight*>
- The edge weights are non-negative integers. **The values of vertex id start from 0**.

Notice that:

- ✓ $2 \le V \le 40000$ for single GPU version
- ✓ $0 \le E \le V \times (V - 1)$
- ✓ $0 \le$ *source vertex id, destination vertex id* $< V$
- ✓ $0 \le W \le 100$
- ✓ 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 (1 << 30) - 1

4 Working Items

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

1. CUDA version blocked APSP algorithm
   - ➢ 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. Makefile
   Please refer to the example in **/home/pp19/share/hw4** on **hades01**

3. Report
   - ➢ Implementation

- ✓ How do you divide your data?
- ✓ 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 of following metrics on the biggest kernel of your program using NVIDIA profiling tools. NVIDIA Profier Guide

- ✓ occupancy
- ✓ sm efficiency
- ✓ shared memory load/store throughput
- ✓ global load/store throughput

It’s better to use print-screen or tables as results on your report.

Analysis those metrics and describe what’s the bottleneck of your implementation.

- ➢ Experiment & Analysis

We encourage you to show your results by figures, charts, and description.

- ✓ System Spec

  Please attach CPU, GPU, RAM and disk information of the system.

- ✓ Time Distribution

  Analyze the time spent in

  1) computing

  2) communication

  3) memory copy (H2D, D2H)

  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!

- ✓ Blocking Factor

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

Please refer to figure 6 and 7 as examples.

- ✓ Optimization

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

  - ✓ Coalesced memory access
  - ✓ Shared memory
  - ✓ Handle bank conflict
  - ✓ CUDA 2D alignment
  - ✓ Occupancy optimization
  - ✓ Large blocking factor
  - ✓ Reduce communication
  - ✓ Streaming

  And you should summarize all optimizations in a chart

  Please refer to figure 8 as example

- ✓ 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!

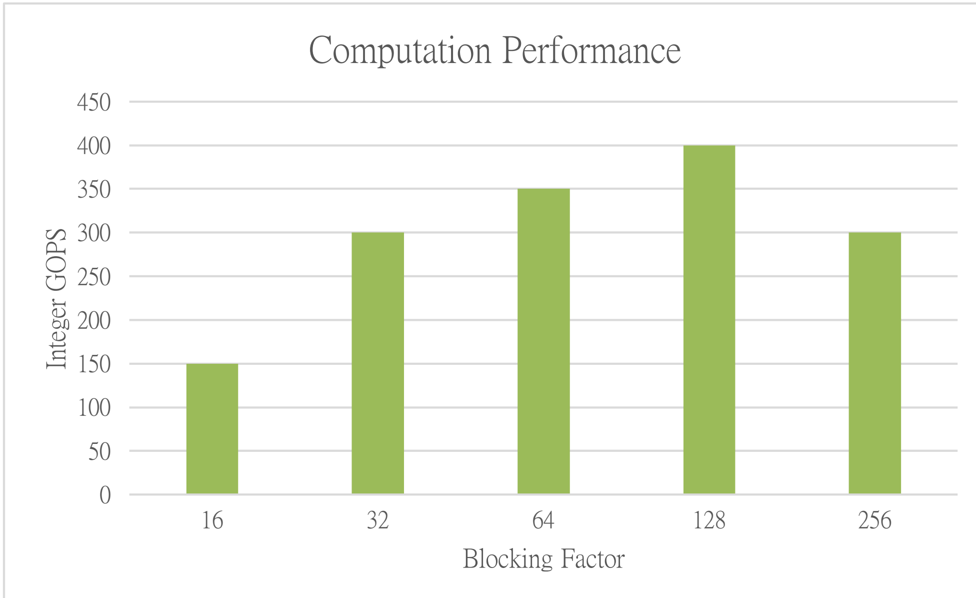


Figure 6: Example chart of performance trend w.r.t. blocking factor

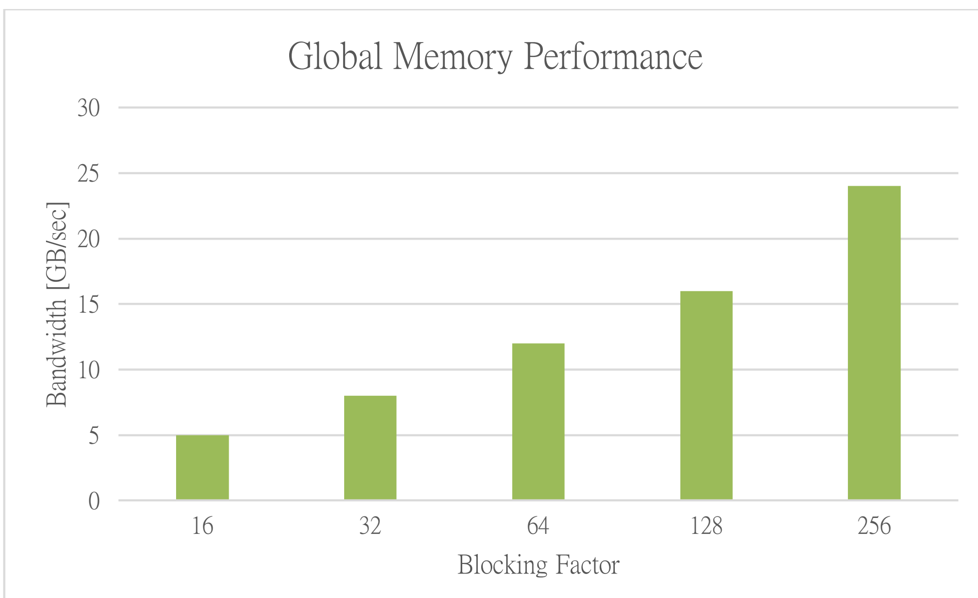


Figure 7: Example chart of global memory bandwidth trend w.r.t. blocking factor

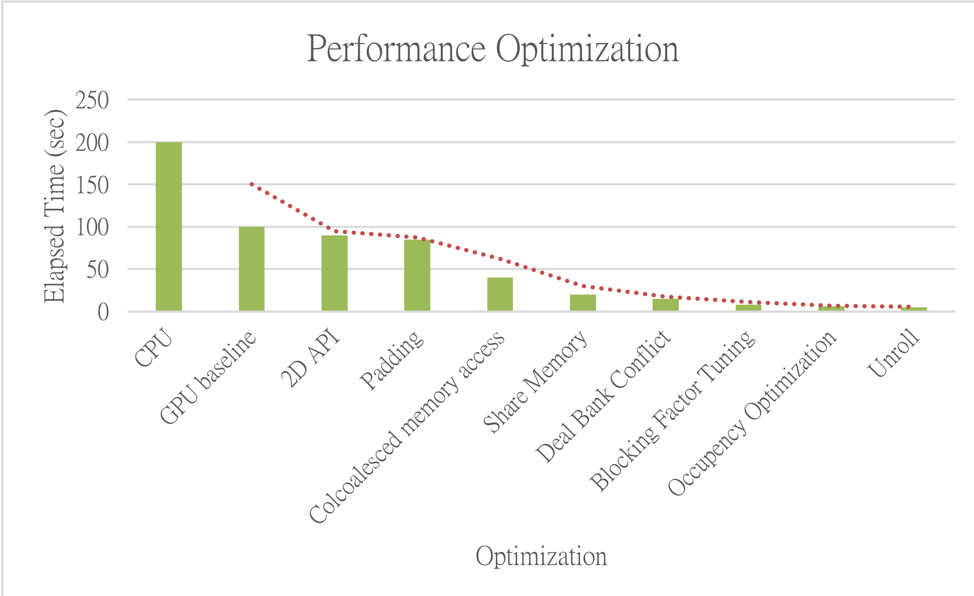


Figure 8: Example chart of performance optimization

5 GRADING

1. Correctness (35%)
2. Performance (30%)
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 ➔ $\lceil 3 \log_2(N+1) \rceil$ extra total points

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

4. Demo (15%)
5. Total Points $= \min\big((1) + (2) + (3) + (4) + (5),\ 100\big)$
6. We grade your scores by hw4a-judge (or hw4-1-judge) and hidden testcases**,** make sure your programs can get correct results on all possible testcases. We will grade the performance score by the largest testcase you can pass within 30 seconds. (Larger testcases, higher score)
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/hw4-1 on **both hades01** and **ilms** (DO NOT pack them):
   - ✓ hw4-1.cu
   - ✓ {student-ID}_report.pdf
   - ✓ Makefile

Note:

- ✓ Your Makefile must be able to build the corresponding targets of the implementations: hw4-1. 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!