CS542200 Parallel Programming

Homework 3: All-Pairs Shortest Path

Due: Dec 25 23:59, 2024

1 Goal

This assignment helps you manage to solve the all-pairs shortest path problem with CPU threads and then further accelerate the program with CUDA accompanied by Blocked Floyd-Warshall algorithm. In this assignment, you will realize how powerful GPUs can be. Finally, we encourage you to optimize your program by exploring different optimizing strategies for performance points.

2 Requirements

- In this assignment, you are asked to implement 3 versions of programs that solve the all-pairs shortest path problem.
  - *CPU version (hw3-1) on Apollo*
    - You are required to use **threading** to parallelize the computation in your program.
    - You can choose any threading library or framework you like (pthread, std::thread, OpenMP, Intel TBB, etc).
    - You can choose any algorithm to solve the problem.
    - You must implement the shortest path algorithm yourself. (Do not use libraries to solve the problem. Ask TA if unsure).
  - *Single-GPU version (hw3-2) on Apollo*
    - Should be optimized to get the performance points (20%).
  - *Multi-GPU version (hw3-3) on Apollo*
    - **Must use 2 GPUs**. Single GPU version is not accepted and will get 0 for correctness score in hw3-3 (even if you get AC on scoreboard).

3 Blocked Floyd-Warshall algorithm

Given an $V \times V$ 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 V vertices. We define an $V \times V$ matrix $D = [d(i,j)]$ where $d(i,j)$ denotes the shortest-path distance from a vertex i

to a vertex j. Let $D^{(k)} = [d^{(k)}(i,j)]$ be the result which all the intermediate vertices are in the set $\{0, 1, 2, ..., k-1\}$.

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

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

The matrix $D^{(V)} = d^{(V)}(i,j)$ gives the answer to the all-pairs shortest path problem.

In the blocked all-pairs shortest path algorithm, we partition D into $\lceil V/B \rceil \times \lceil V/B \rceil$ blocks of $B \times B$ submatrices. The number B is called the *blocking factor*. For instance, in figure 1, we divide a 6×6 matrix into 3×3 submatrices (or blocks) by $B = 2$.

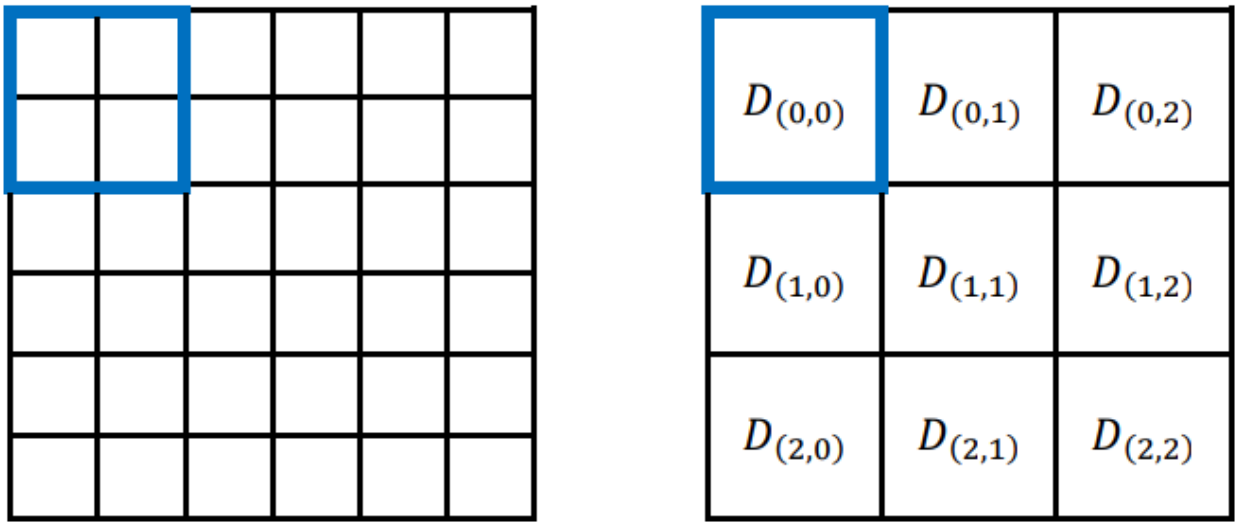

Figure 1: Divide a matrix by B = 2

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

Assuming a block is identified by its index (I,J), where $0 \leq I,J < \lceil V/B \rceil$. 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 round, the first phase is to compute the $B \times B$ pivot block $D^{(k \cdot B)}_{(k-1,k-1)}$.

  For instance, in the 1st round, $D^{(2)}_{(0,0)}$ is computed as follows:

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

Note that the 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 round, it computes all $D^{(k \cdot B)}_{(h,k-1)}$ and $D^{(k \cdot B)}_{(k-1,h)}$ where $h \neq k - 1$.

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

  For instance, in the 1st round, the result of $D^{(2)}_{(0,2)}$ depends on $D^{(2)}_{(0,0)}$ and $D^{(0)}_{(0,2)}$:

$$d^{(1)}(0,4) = min(d^{(0)}(0,4), d^{(2)}(0,0) + d^{(0)}(0,4))$$
$$d^{(1)}(0,5) = min(d^{(0)}(0,5), d^{(2)}(0,0) + d^{(0)}(0,5))$$
$$d^{(1)}(1,4) = min(d^{(0)}(1,4), d^{(2)}(1,0) + d^{(0)}(0,4))$$
$$d^{(1)}(1,5) = min(d^{(0)}(1,5), d^{(2)}(1,0) + d^{(0)}(0,5))$$
$$d^{(2)}(0,4) = min(d^{(1)}(0,4), d^{(2)}(0,1) + d^{(1)}(1,4))$$
$$d^{(2)}(0,5) = min(d^{(1)}(0,5), d^{(2)}(0,1) + d^{(1)}(1,5))$$
$$d^{(2)}(1,4) = min(d^{(1)}(1,4), d^{(2)}(1,1) + d^{(1)}(1,4))$$
$$d^{(2)}(1,5) = min(d^{(1)}(1,5), d^{(2)}(1,1) + d^{(1)}(1,5))$$

Phase 3: other blocks.

In the k-th round, it computes all $D^{(k \cdot B)}_{(h_1,h_2)}$ where $h_1, h_2 \neq k - 1$.

The result of these blocks depends on the result from phase 2 and itself.

For instance, in the 1st round, the result of $D^{(2)}_{(1,2)}$ depends on $D^{(2)}_{(1,0)}$ and $D^{(2)}_{(0,2)}$:

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

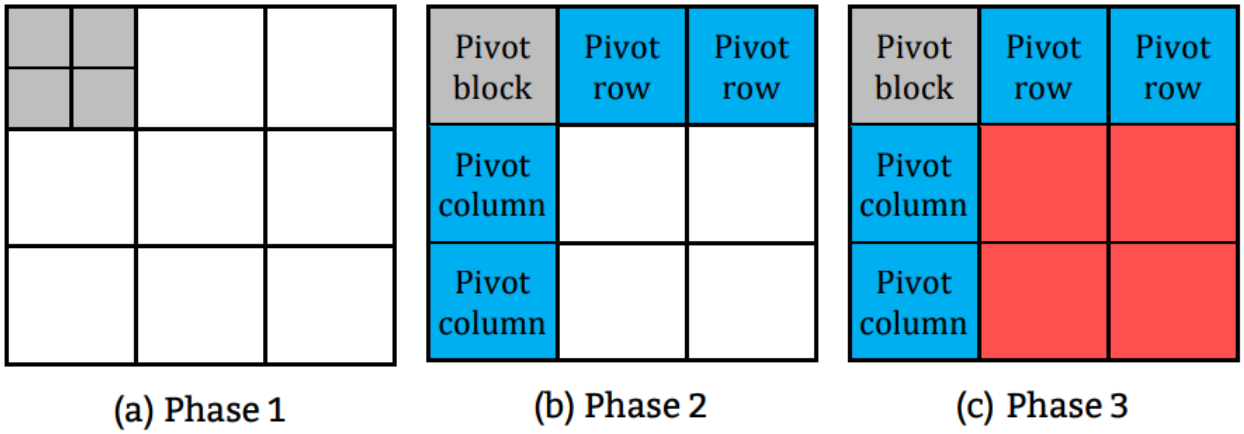

Figure 2: The 3 phases of the blocked FW algorithm in the first round.

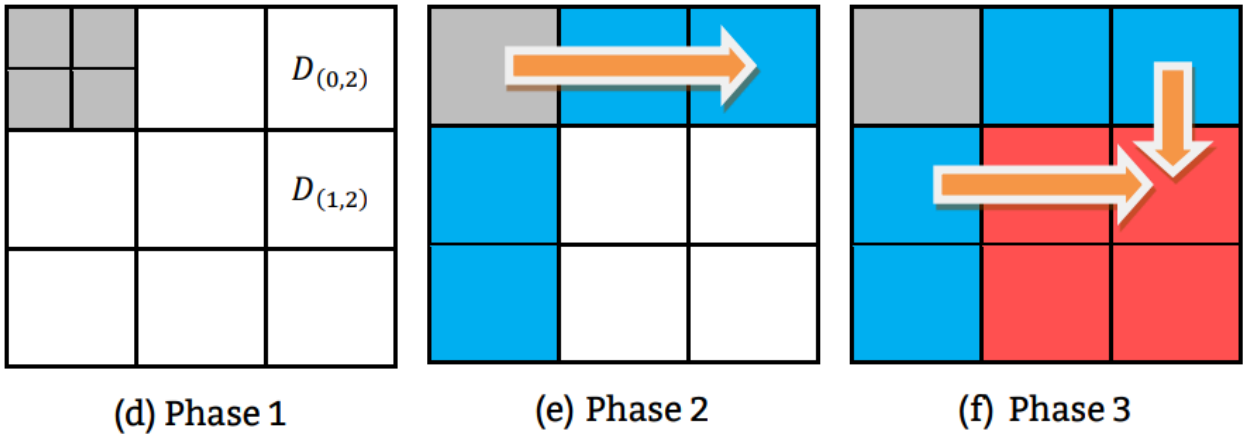

Figure 3: The computations of $D^{(2)}_{(0,2)}$, $D^{(2)}_{(1,2)}$ and their dependencies in the first round.

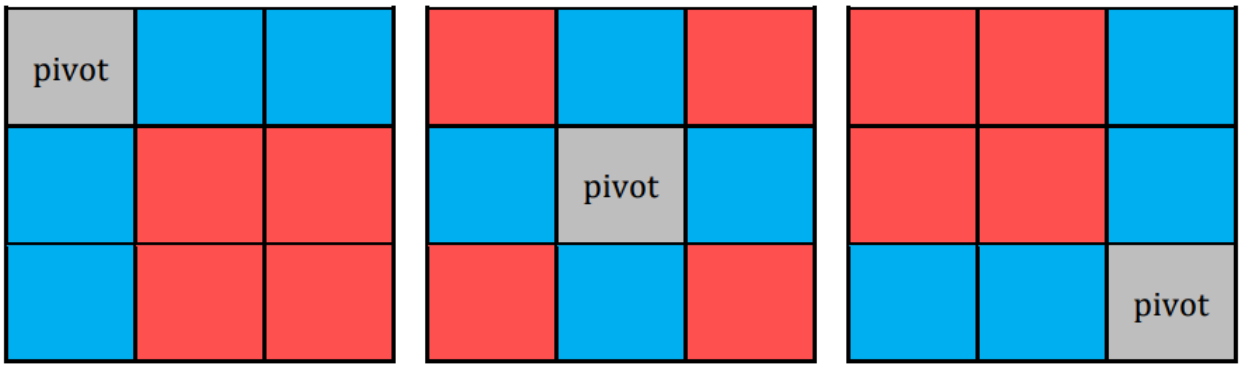

Figure 4: In this particular example where $V = 6$ and $B = 2$, we will require $\lceil V/B \rceil = 3$ rounds.

4 Run your programs

- **Command line specification**

```
# CPU
srun -N1 -n1 -cCPUS ./hw3-1 INPUTFILE OUTPUTFILE
# Single-GPU
srun -N1 -n1 --gres=gpu:1 ./hw3-2 INPUTFILE OUTPUTFILE
# Multi-GPU
srun -N1 -n1 -c2 --gres=gpu:2 ./hw3-3 INPUTFILE OUTPUTFILE
```

  - `CPUS`: Number of CPUs, specified by TA.
  - `INPUTFILE`: The pathname of the input file. Your program should read the input graph from this file.
  - `OUTPUTFILE`: The pathname of the output file. Your program should output the shortest path distances to this file.
- **Input specification**
  - The input is a directed graph with non-negative edge distances.
  - The input file is a binary file containing 32-bit integers. You can use the `int` type in C/C++.
  - The first two integers are *the number of vertices (V)* and *the number of edges (E)*.
  - Then, there are E edges. Each edge consists of 3 integers:

1. *source vertex id* (src_i)
2. *destination vertex id* (dst_i)
3. *edge weight* (w_i)

- The values of vertex indexes & edge indexes start at 0.
- The ranges for the input are:
  - $2 \leq V \leq 6000$ **(CPU)**
  - $2 \leq V \leq 40000$ **(Single-GPU)**
  - $2 \leq V \leq 60000$ **(Multi-GPU)**
  - $0 \leq E \leq V \times (V - 1)$
  - $0 \leq src_i, dst_i < V$
  - $src_i \neq dst_i$
  - if $src_i = src_j$ then $dst_i \neq dst_j$ (there will not be repeated edges)
  - $0 \leq w_i \leq 1000$

Here's 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	edge 0's distance
0020	32-bit integer		src id for edge 1
…	…	…	…
0076	32-bit integer		edge 5's distance

- **Output specification**
  - The output file is also in binary format.
  - For an input file with V vertices, you should output an output file containing V^2 integers.

- The first V integers should be the shortest path distances for starting from edge 0: $dist(0, 0), dist(0, 1), dist(0, 2), \ldots, dist(0, V-1)$; then the following Vintegers would be the shortest path distances starting from edge 1: $dist(1, 0), dist(1, 1), dist(1, 2), \ldots, dist(1, V-1)$; and so on, totaling V^2 integers.
- $dist(i, j) = 0$ where $i = j$.
- If there is no valid path between$i \rightarrow j$, please output with: $dist(i, j) = 2^{30} - 1 = 1073741823$.

Example output file:

offset	type	decimal value	description
0000	32-bit integer	0	$dist(0,\ 0)$
0004	32-bit integer	?	$dist(0,\ 1)$
0008	32-bit integer	?	$dist(0,\ 2)$
…	…	…	…
$4V^2 - 8$	32-bit integer	?	$dist(V-1,\ V-2)$
$4V^2 - 4$	32-bit integer	0	$dist(V-1,\ V-1)$

5 Report

Answer the questions below. You are recommended to use the same section numbering as they are listed.

1. Implementation

a. Which algorithm do you choose in hw3-1?

b. How do you divide your data in hw3-2, hw3-3?

c. What's your configuration in hw3-2, hw3-3? And why? (e.g. blocking factor, #blocks, #threads)

d. How do you implement the communication in hw3-3?

e. Briefly describe your implementations in diagrams, figures or sentences.

2. Profiling Results (hw3-2)

Provide the profiling results of following metrics on the biggest kernel of your program using NVIDIA profiling tools. NVIDIA Profiler Guide.

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

3. Experiment & Analysis

a. System Spec

If you didn't use our Apollo server for the experiments, please show the CPU, RAM, disk of the system.

b. Blocking Factor **(hw3-2)**

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) (You might want to check nvprof and Metrics Reference)

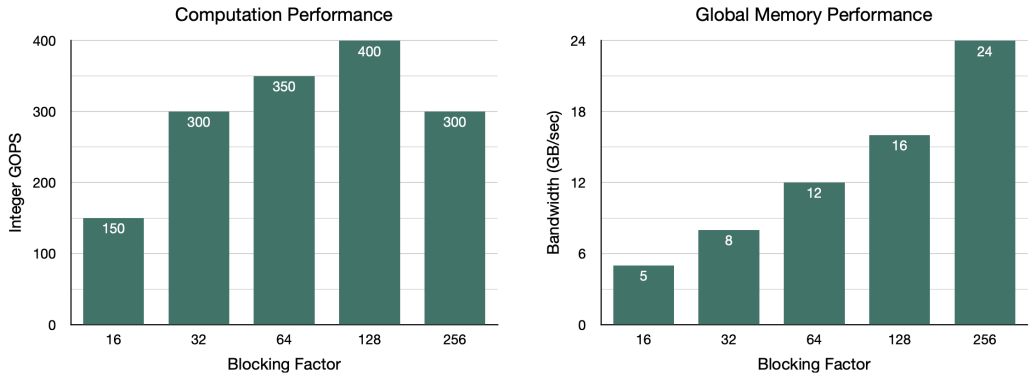

Figure 5: Example chart of performance and global memory bandwidth trend w.r.t. blocking factor

Note:

To run nvprof on Apollo with flags like --metrics, please run on the slurm partition nvidia. e.g. srun -p nvidia -N1 -n1 --gres=gpu:1 nvprof --metrics gld_throughput ./hw3-2 /home/pp23/share/hw3-2/cases/c01.1 c01.1.out

Please note that nvprof is only available for the GTX 1080. For more details,

please refer to the Nvprof Transition Guide.

c. Optimization **(hw3-2)**

Any optimizations after you port the algorithm on GPU, describe 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

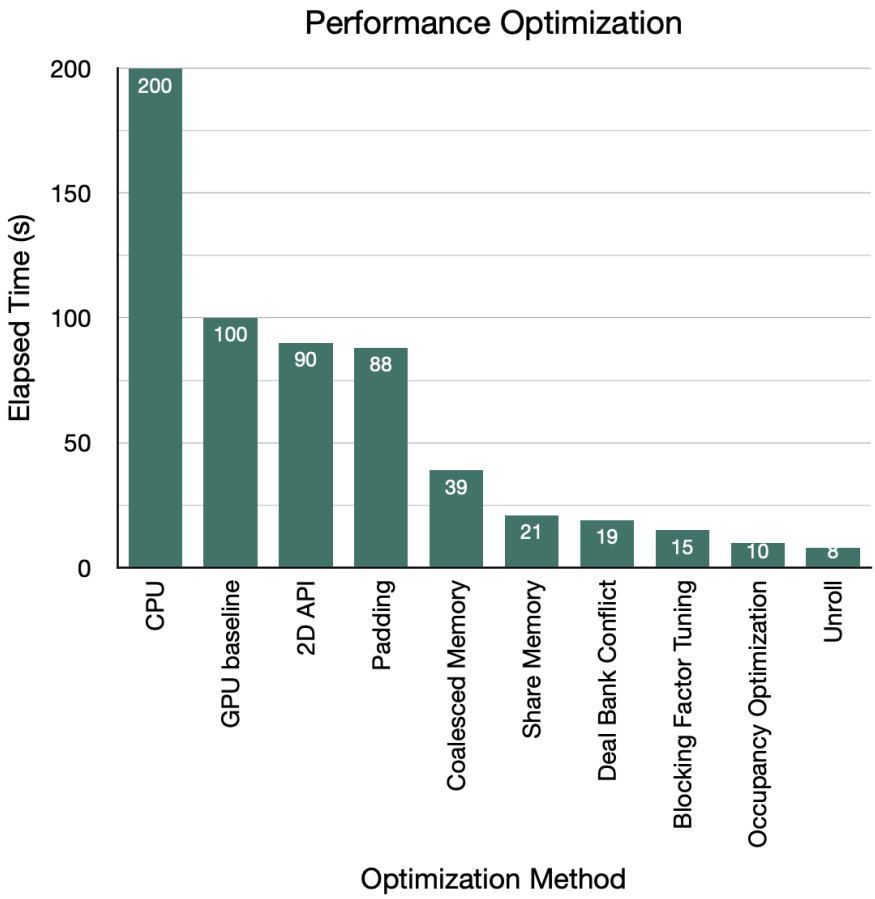

Figure 6: Example chart of performance optimization¶

d. Weak scalability **(hw3-3)**

   Observe weak scalability of the multi-GPU implementations

e. Time Distribution **(hw3-2)**

   Analyze the time spent in:

   - computing
   - communication
   - memory copy (H2D, D2H)
   - I/O of your program w.r.t. input size.

f. Others

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

4. Experiment on AMD GPU

a. Use the method we taught in lab3 to run your GPU version code on AMD GPU.

b. Compare the difference between Nvidia GPU and AMD GPU.

c. Share your insight and reflection on the AMD GPU experiment.

d. You need to run the single GPU and multi GPU version on AMD, note that when running multi GPU on AMD node, the judge might get some error, if you encounter this, just mention it in the report.

e. The experiment on AMD GPU would only be part of your report score, it won't affect your correctness and performance score.

5. Experience & conclusion

a. What have you learned from this homework?

b. Feedback (optional)

6 GRADING

---

1. **[40%] Correctness**

An unknown number of test cases ("cXX.1") will be used to test your implementation.

- CPU (15%)
  - You get 15 points if you passed all the test cases, $max(0, 15 - k)$points if there are k failed test cases.
  - Time limit for each case: (960 seconds) / (number of CPU cores).
- Single-GPU (15%)
  - You get 15 points if you passed all the test cases, $max(0, 15 - k)$ points if there are k failed test cases.
  - Each test case has a 30s time limit.

- Multi-GPU (10%)
    - We provide some public test cases c[01-07].1
    - For final verdict, we will use 7 hidden test cases similar to c[03-07].1 .

2. **[20%] Performance (*Single-GPU version only*)**
    - We have 30 performance test cases named pXXk1, XX = 11 ~ 40
    - Each test case has a 30s time limit.
    - Basically, larger XX test cases require longer time.
    - You will get "max(***XX-10, 0***)" points if you pass test cases p11k1 ~ pXXk1. Otherwise, zero.
    - For example, if you pass test cases p11k1 ~ p23k1, p25k1 and fail other test cases. You will get 13 points.
    - If XX > 20, then extra points will still count. (but the max point of this homework is still 100)
3. **[20%] Demo**
    - A demo session will be held remotely. You'll be asked questions about the homework.
4. **[20%] Report**
    - Grading is based on your evaluation, discussion and writing. If you want to get more points, design or conduct more experiments to analyze your implementation.

7 Submission

Upload the files below to eeclass. (**DO NOT COMPRESS THEM**)

- hw3-1.cc
- hw3-2.cu
- hw3-3.cu
- Makefile (optional)
- hw3_{student_ID}.md

8 Final Notes

- Type hw3-1-judge(apollo), hw3-2-judge, hw3-3-judge to run the test cases.
- If you want to submit your Makefile, **please combine it into a single file**. All of your code should be compiled by make hw3-1, make hw3-2 and make hw3-3 with the same Makefile.
- Scoreboard:
  - https://apollo.cs.nthu.edu.tw/pp24/scoreboard/hw3-1/
  - https://apollo.cs.nthu.edu.tw/pp24/scoreboard/hw3-2/
  - https://apollo.cs.nthu.edu.tw/pp24/scoreboard/hw3-3/
- Use the hw3-cat command to view the binary test cases in text format.
- Resources are provided under /home/pp24/share/hw3-*/ on Apollo
  - Makefile - example Makefile
  - cases/ - sample test cases
- However, the final result is based on Apollo (GTX 1080), so please make sure to fine tune your performance on Apollo.
- Contact TA via pp@lsalab.cs.nthu.edu.tw or eeclass if you find any problems with the homework specification, judge scripts, example source code or the test cases.
- You are allowed to discuss and exchange ideas with others, but you are required to write the code on your own. You'll get **0 points** if we found you cheating.