CS542200 Parallel Programming

HW 4-2: Blocked All-Pairs Shortest Path (Multi-cards)

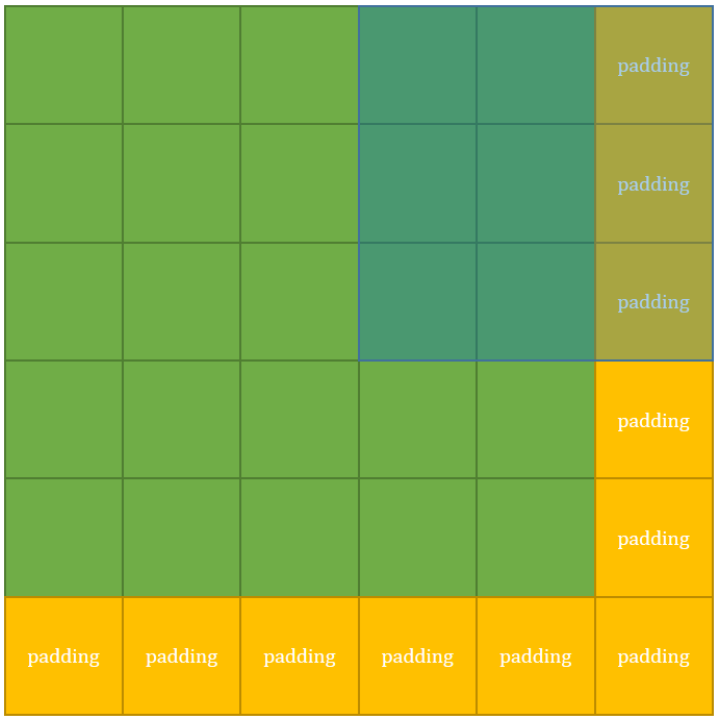
109065517 林佳縈

1. Implementation

a. Data Division

For this part, I'd like to divide them into the following steps to demonstrate how I deal with the data.

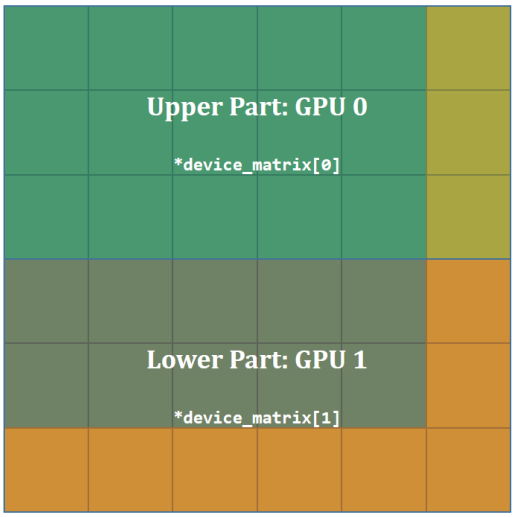
- *Determine adjacency matrix size: (padding):*
  In the first step, after reading in how many vertices I have, I'll then count the size of which the adjacency matrix should be. In order to obtain better performance, in this part, I'll do padding.
  Take the following graph for example, say I have vertex number 5, so I then create a 5 by 5 adjacency matrix, which is the green part. And about the blocking factor, say we have 3, since blocking factor is 3, I have to thus extend the original adjacency matrix to the size which width and height can be divided by the blocking factor. Therefore, the final adjacency matrix is a 6 by 6 matrix, where the yellow part is for padding and blue filter is to act as blocking factor.



- *Put data into an 1D adjacency matrix:*
  After determining the width and height for the adjacency matrix, I then put the input data into the matrix. For computation simplicity. I

have the adjacency matrix as a 1D matrix with padding part set to INF.

- *(cuda) Memory Copy to corresponding GPU:*
  To better make use of the GPU resources, for memory copy part, I have the first GPU keep the upper part of the matrix while the second keep the lower part. (For visual simplicity, I will show the adjacency matrix as a 2D one, while in reality it is 1D.) This is how I locate the entire data to 2 GPUs correspondingly.



b. Communication Implementation

Measuring the time distribution by **nvprof**, I found that phase1 and phase2 didn't actually cost too much computation resource. Therefore, have two GPUs both compute phase1 and phase2. Yet, for phase3, which is the most computation resource consumer, I have the two GPUs compute different part of the matrix.

Communication is basically done by the following

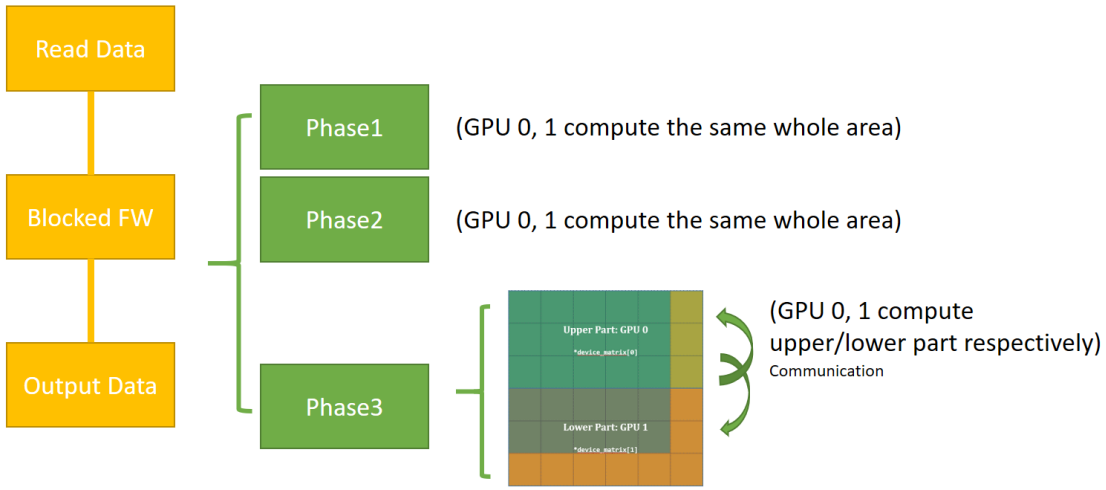
for each iteration, each phase will do return the updated value of the adjacency matrix to the original adjacency matrix in host by cudaMemcpyDeviceToHost.

Host matrix will collect all the updated value and then again distribute data that should be count by cudaMemcpyHostToDevice.

c. Implementation

The following diagram demonstrates how the whole program is executed. First of all, read data. The data is read into a 1D adjacency matrix with width and height set to V*V, where V is the number of nodes + padding. The entire matrix is set to INF except the diagonal, which is set to 0. Secondly, for each GPU resources, a V*V device_matrix is allocated to it, and thus load the host's adjacency matrix value into it.

Third, the blocked Floyd Warshall part. For phase1 and phase2, each GPU computes the same part while for phase3, GPU 0 and 1 computes the upper and lower part correspondingly.
After each iteration, there will be a communication taken place. The updated value will be transferred to the host matrix and the new uncalculated value will be load to device matrix for a new computation.
Last, output the result.



▲ Diagram: Implementation

2. Experiment & Analysis

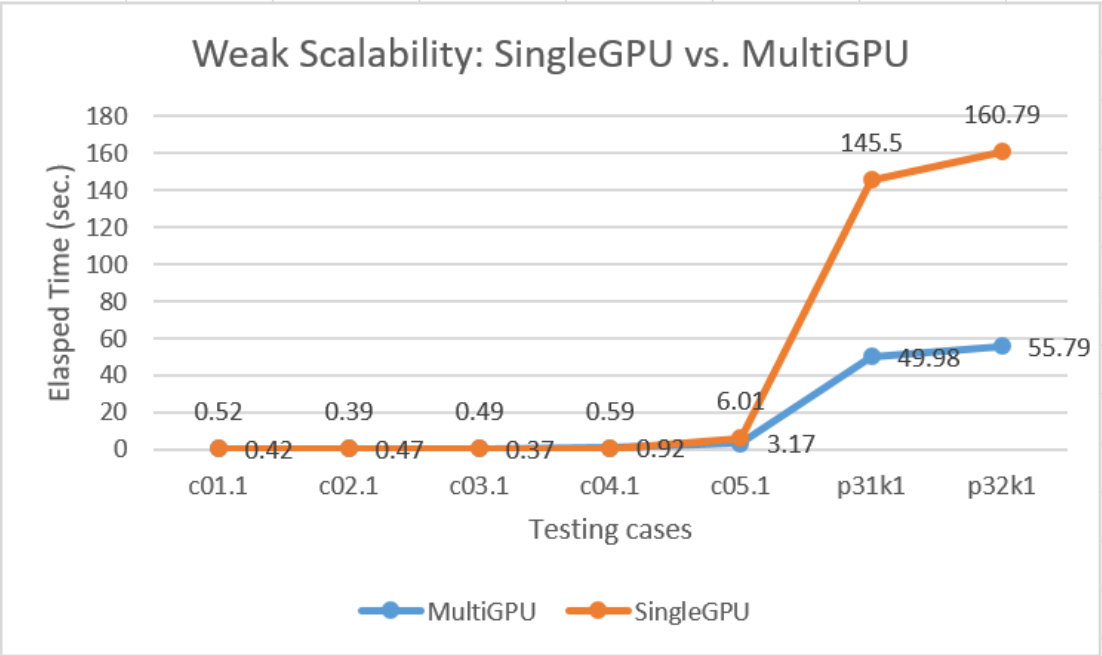
a. System Spec

During the entire implementation and testing phases, I performed them all on hades server.

b. Weak Scalability

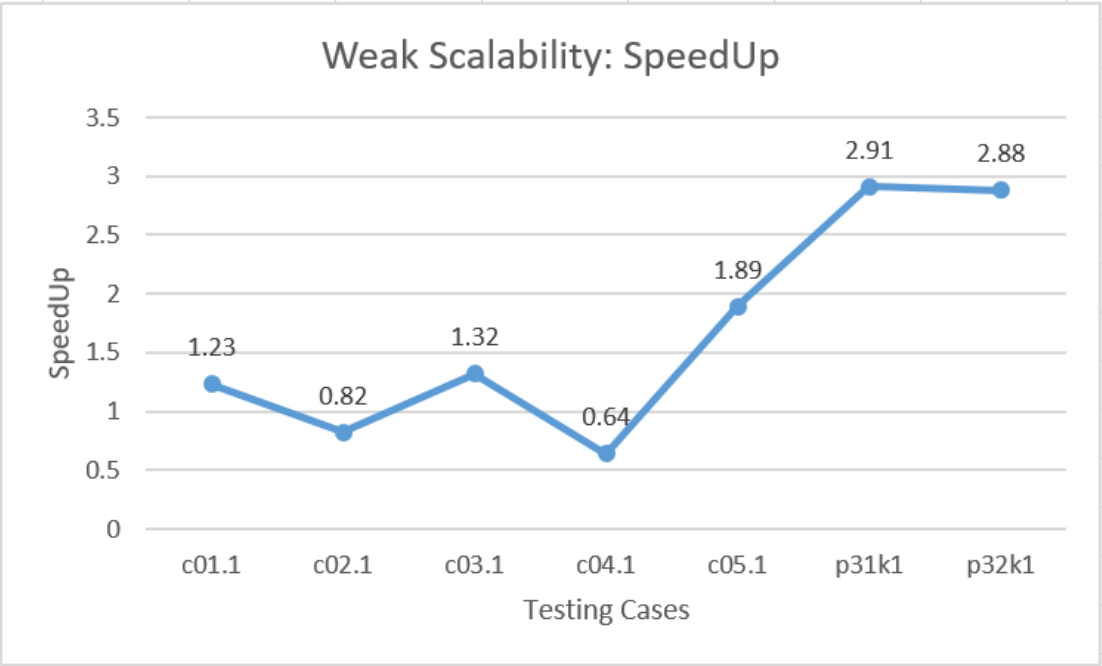
For weak scalability test, the following analysis shows the results for multi-GPU and single-GPU version tested on same testing cases correspondingly. Testing cases are: c01.1~c05.1, p31k1, p32k1 in order to show different execution time with regards to different problem sizes.
The following chart shows the number of vertices corresponding to each testing cases.

Cases	c01.1	c02.1	c03.1	c04.1	c05.1	p31k1	p32k1
# of vertices	5	160	999	5000	11000	31000	32000



▲ Weak Scalability: MultiGPU vs. SingleGPU

For the above analysis graph, we can tell that for bigger problem size, such as testing cases p31k1 and p32k2, multi GPU still has advantage on less computation time spent.



▲ Speed Up

How I calculate speedup level:

$$SpeedUp = T_e(single\ GPU) / T_e(multi\ GPU),$$

$$T_e = execution\ time$$

The above chart shows the speed up level by replacing single GPU with multi GPU. For testing cases with large number of vertices, speedup is

significant.

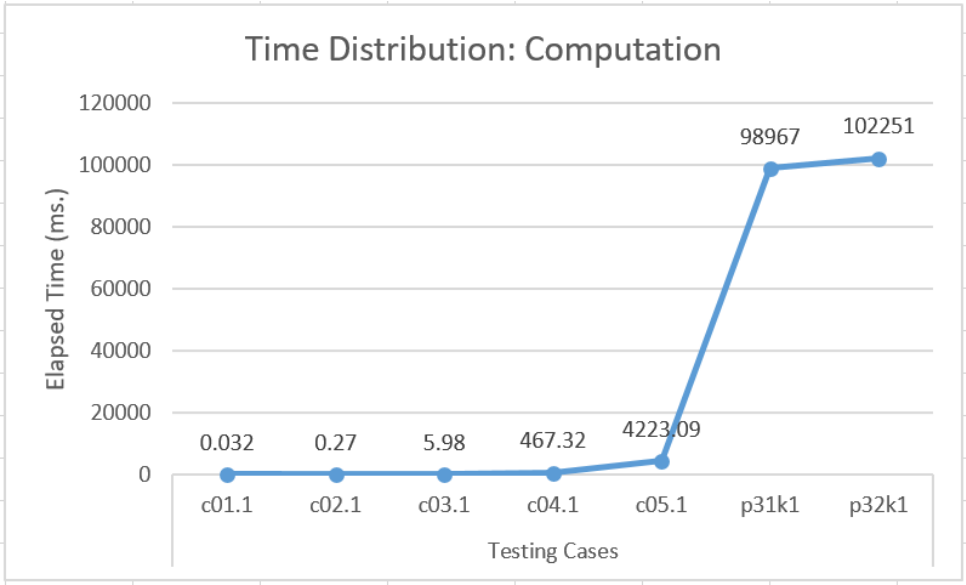
c. Time Distribution

To analyze the time spent on the following part, I chose testing cases of c01.1~c05.1, p31k1 and p32k1 to demonstrate the analysis. As for how I obtain the time spent, I create cuda events by wrapping up the part where I want to calculate the time spent. The structure is like this:

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);
...
cudaEventRecord(stop );
cudaEventSynchronize(stop);
```

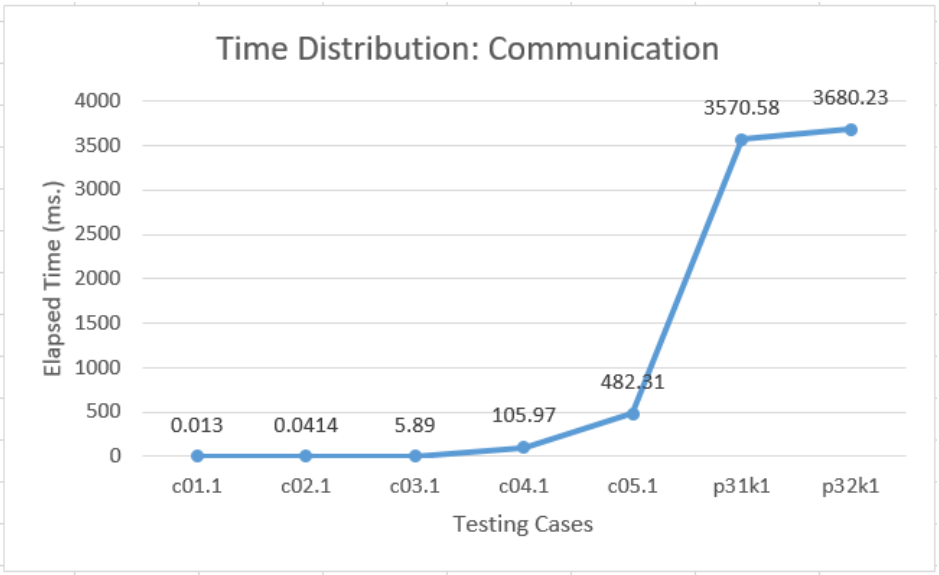
- Computation

  To measure computation time, I wrap up the part of where kernels are launched. From the first kernel phase1 to phase3.



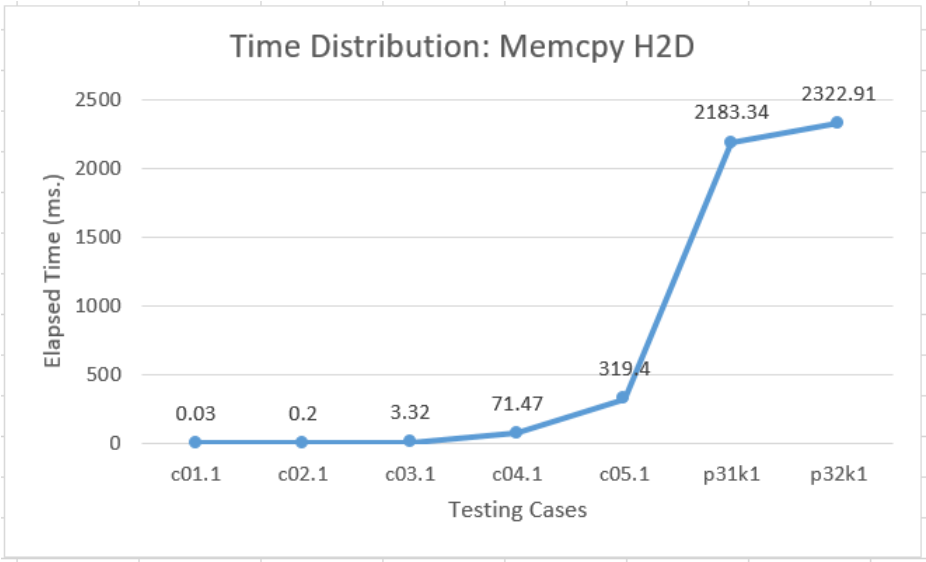
- Communication

  For communication, I wrap up for the part of GPUs send back updated to data and host matrix send the new uncalculated data to GPUs for each iteration.



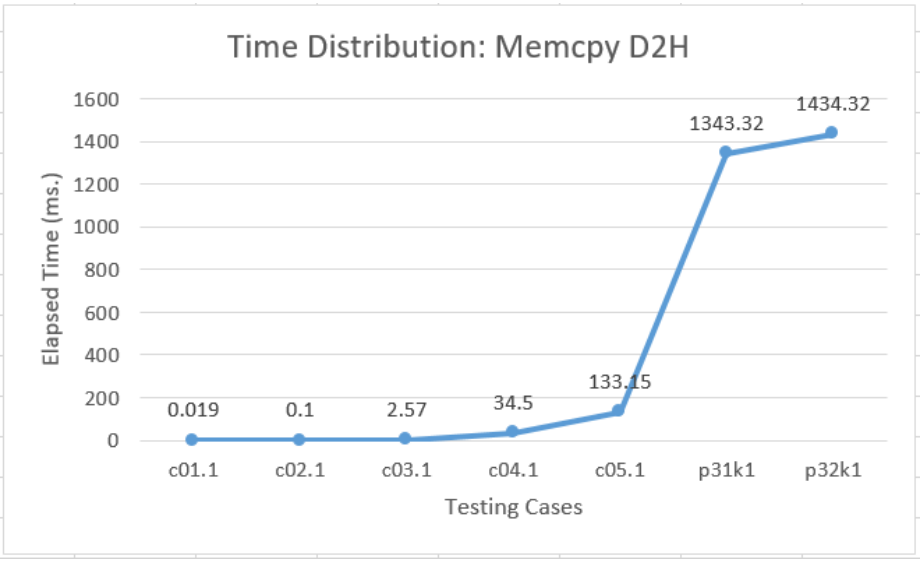
- Memory Copy
  - Host to Device (H2D)

    To measure host to device time spent, I wrapped up the part where I have cudaMemcpyHostToDevice instruction.



  - Device to Host (D2H)

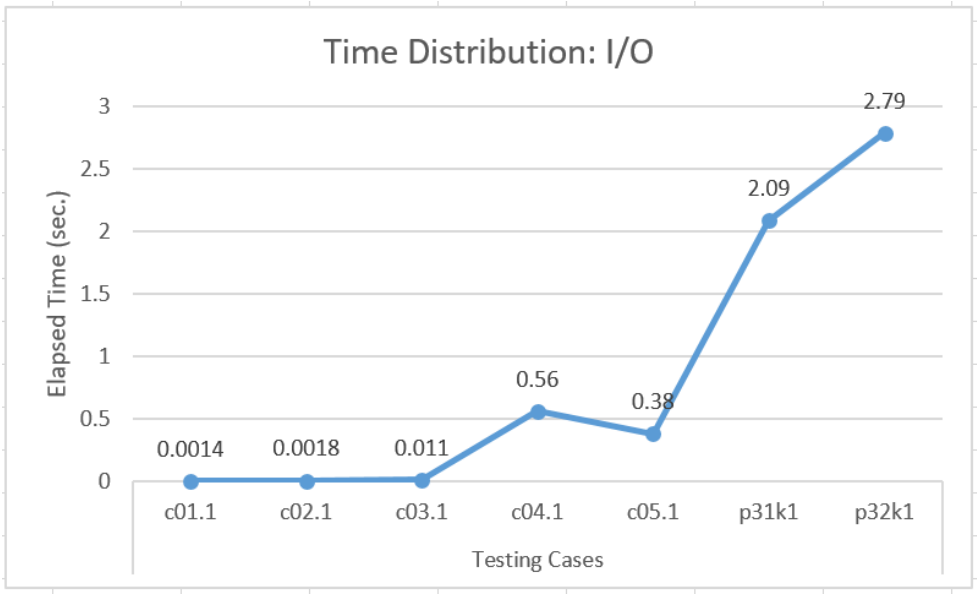
    To measure device to host time spent, I wrapped up the part where I have cudaMemcpyDeviceToHost instruction.



- I/O

  For I/O time spent, I use the clock_t provided by <time.h> to

```
#include clock_t start, end;
double IO_time _spent;
start = clock();
... /* Do something */
end = clock();
IO_time_spent = ((double) (end - start)) / CLOCKS_PER_SEC;
```

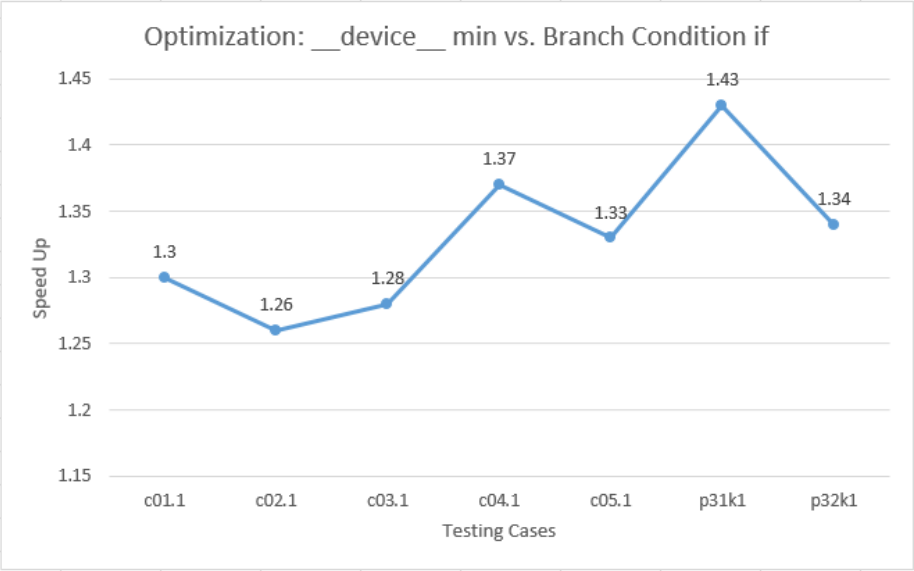


measure. The logic is the same as cudaEvent but it measures the code other than cuda part instead.

d. Others

- Branch Replacement

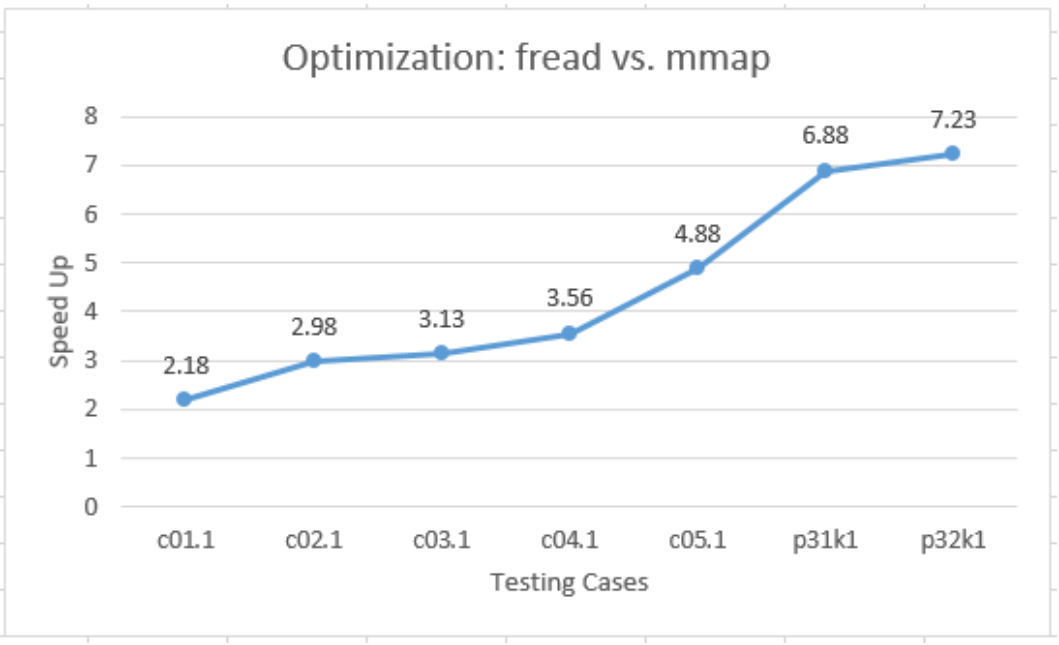
  As branches cost a lot of computation resources (time) and each time when I use nvprof to check bottleneck of the whole program, it is always thanks to phase3. Therefore, the first optimization I did is to replace the "if" conditional statement in phase3 by __device__ min provided by Cuda Math API. The evaluation is as below:



- I/O Optimization

  Reading data is a place that I found can accelerate the program. At first, I used the most common way of reading data, that is by fstream. But soon I found it is, for cases with large vertices number, not a really great way to deal with the part. Then, I replaced the original fstream with mmap. The speedup level can be seen below:
  I picked testing cases c01.1~c05.1, p31k1, p32k1.

  SpeedUp = Time(fread) / Time(mmap)



3. **Experience & Conclusion**

   At first, I thought the structure of multiGPU implementation would be completely different from the original implementation. Yet, I found that the keypoint of how to make the implementation this time way faster is still the job that should be done in HW4-1. Also, during hw4-1, it seems that we can simply use **nvprof** to do time measurement, but for hw4-2 it doesn't really work out as **nvprof** would get its own time overlapping so on and so forth. During implementation, I felt like sometimes to obtain better performance, the implementation will therefore be not that straightforward which required sometimes a bit brain twisting.