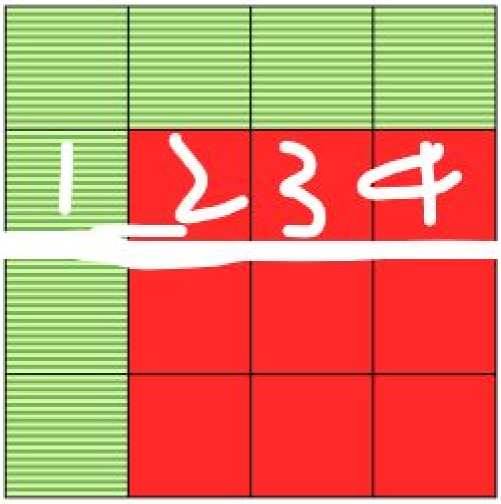
HW 4-2: Blocked All-Pairs Shortest Path (Multi-cards)

106033233 資工21 周聖諺

1. Implementation
    1. Data Division: I divide the whole matrix into upper and lower parts. As the following figure shows, The GPU that handles the upper part should relax the path during phase 3 and then, pass the next pivot row to the other GPU when the pivot row is in the upper part.(The green cells are the current pivot row and pivot column) When the pivot row is in the lower part, another GPU should compute the pivot row and transfer the data to another GPU.



Phase 3

    2. Communication: I use `cudaMemcpyPeer` to transfer the next pivot row to another GPU after the phase3 finishes. It will fill up the PCIE bus as much as possible.

3. I modify the input and output code to read the whole file at once. It speeds up the program effectively.

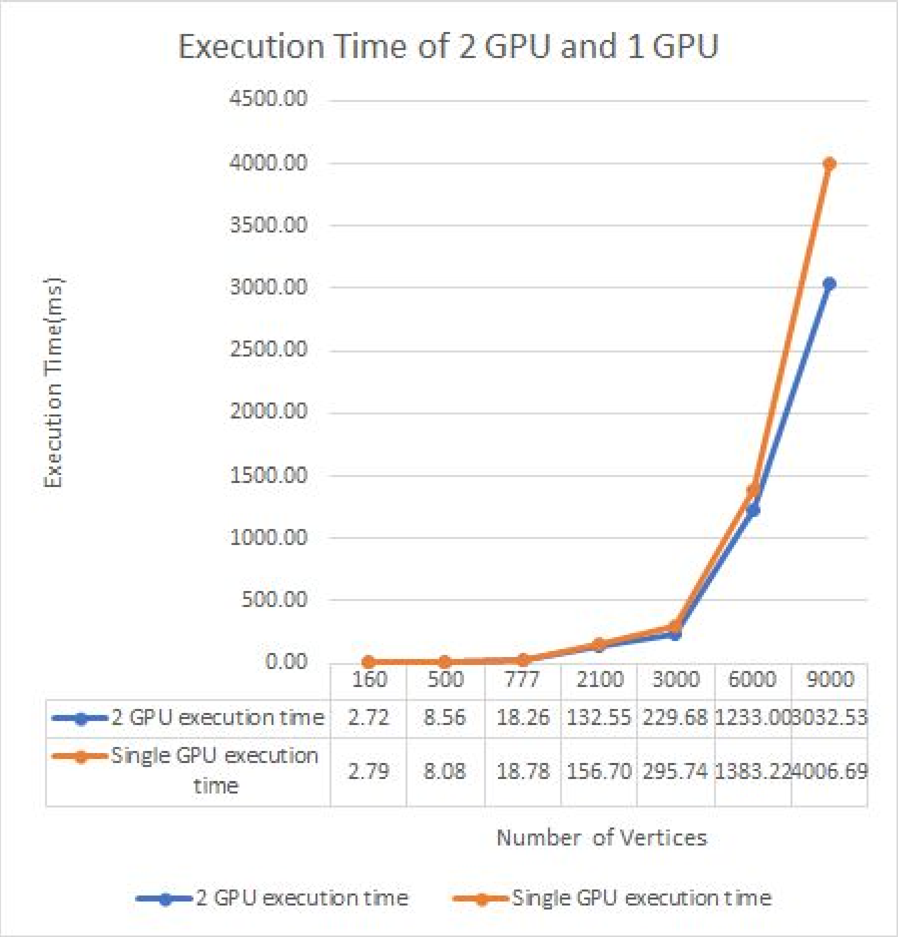
2. Experiment & Analysis

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

A. System Spec

(1) CPU: Intel(R) Core(TM) i9-7960X CPU @ 2.80GHz 32 cores
(2) RAM: 62.4 GB
(3) GPU: GeForce RTX 2080 TI *4
(4) Disk: 1.9T

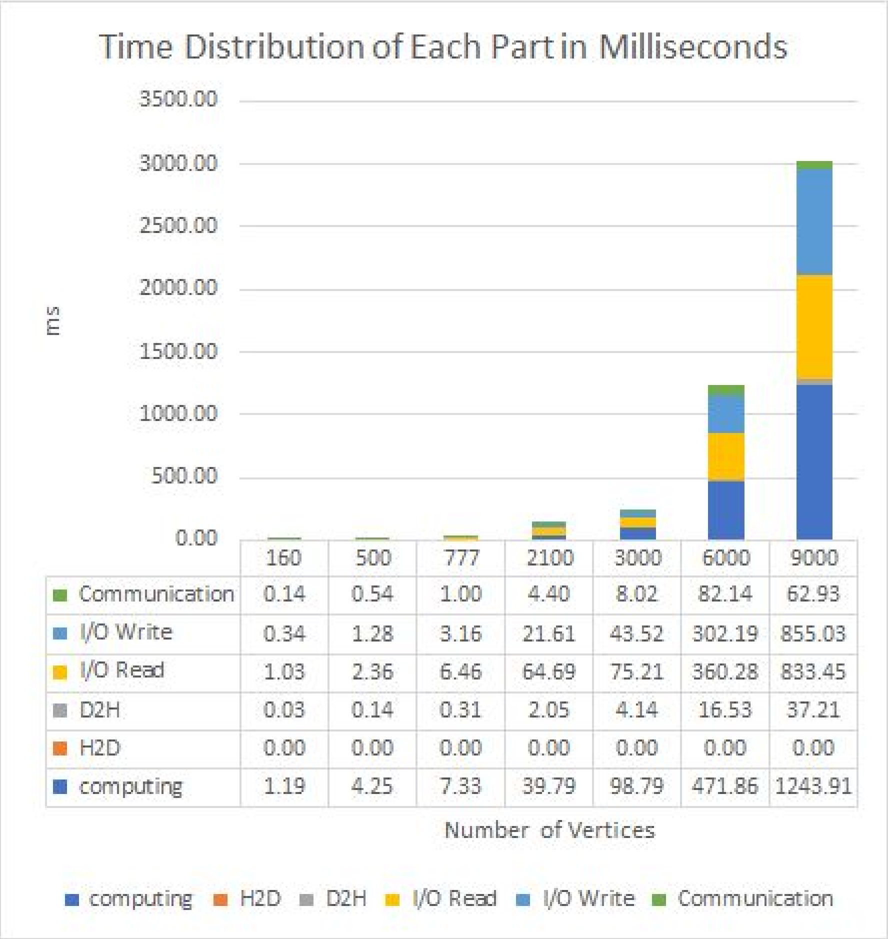
B. Weak Scalability



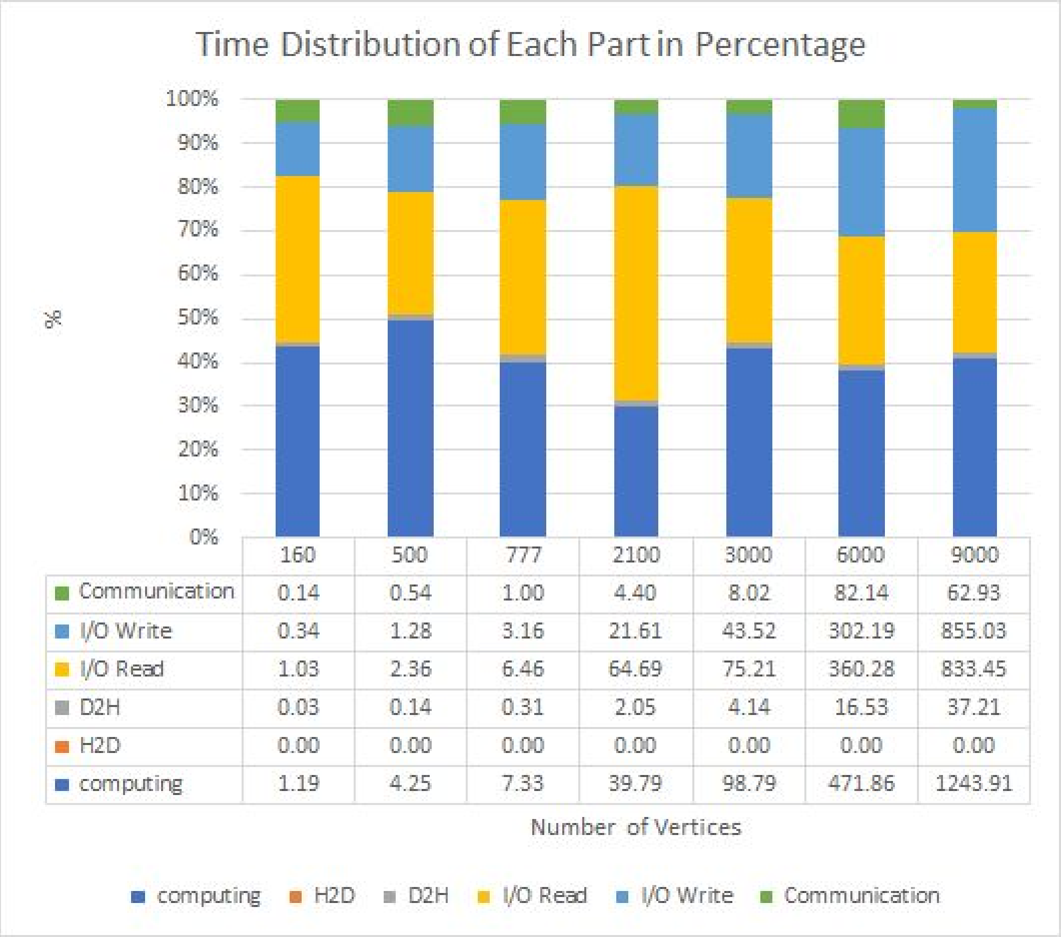
	160	500	777	2100	3000	6000	9000
2 GPU execution time	2.72	8.56	18.26	132.55	229.68	1233.00	3032.53
Single GPU execution time	2.79	8.08	18.78	156.70	295.74	1383.22	4006.69

It seems that the scalability isn’t very good. 2 GPUs only speed up 1.32 times faster than a GPU. It may be due to the small testcase, since the gap between 2 lines gets larger as the file gets bigger.

C. Time Distribution

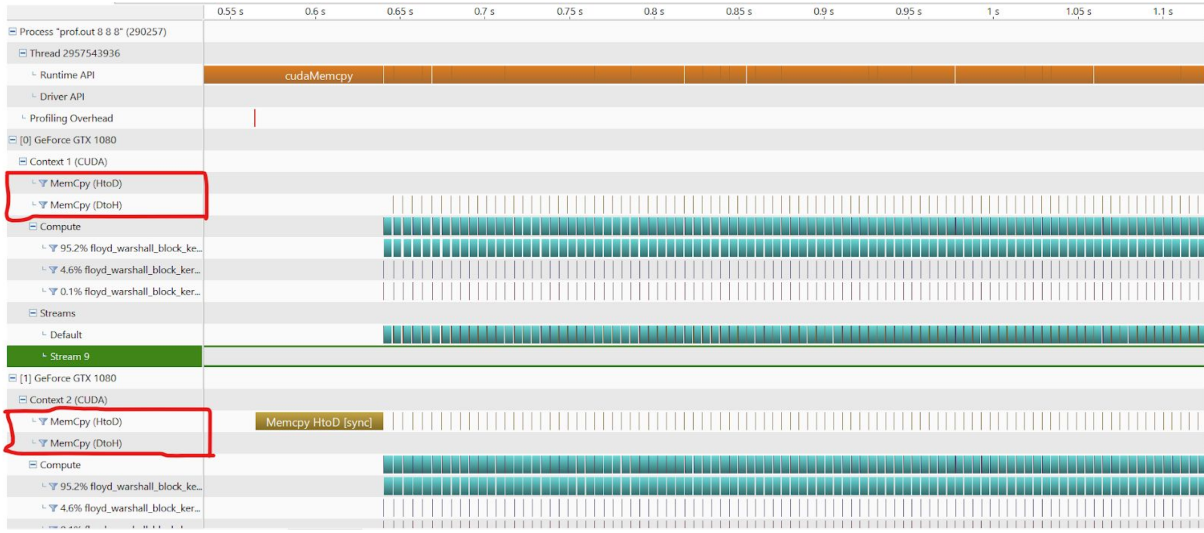


	160	500	777	2100	3000	6000	9000
🟩 Communication	0.14	0.54	1.00	4.40	8.02	82.14	62.93
🟦 I/O Write	0.34	1.28	3.16	21.61	43.52	302.19	855.03
🟨 I/O Read	1.03	2.36	6.46	64.69	75.21	360.28	833.45
⬜ D2H	0.03	0.14	0.31	2.05	4.14	16.53	37.21
🟧 H2D	0.00	0.00	0.00	0.00	0.00	0.00	0.00
🟦 computing	1.19	4.25	7.33	39.79	98.79	471.86	1243.91



	160	500	777	2100	3000	6000	9000
Communication	0.14	0.54	1.00	4.40	8.02	82.14	62.93
I/O Write	0.34	1.28	3.16	21.61	43.52	302.19	855.03
I/O Read	1.03	2.36	6.46	64.69	75.21	360.28	833.45
D2H	0.03	0.14	0.31	2.05	4.14	16.53	37.21
H2D	0.00	0.00	0.00	0.00	0.00	0.00	0.00
computing	1.19	4.25	7.33	39.79	98.79	471.86	1243.91

As the above charts show, most of the time is taken by computation and I/O, although I've improved the data accessing pattern. Nearly ½ to ⅓ of time spent on I/O read. Computation also takes lots of time. However, surprisingly, communication takes the least amount of time. I also use nvprof to profile the time distribution of the program. The profiler gives the same result that communication takes less than 10% of total time.



The figure above is the profiling result of the program. The rows that are wrapped by red boxes are the times of data transfer. Its width is nearly the same as the phase2, that is only 4.6%.

Well, why do we still get a bad speed up ratio while the communication between GPUs isn't a bottleneck? I guess that it's due to the size of the graph. Since the graph isn't large enough to exhaust the GPU in phase3 at once and there are still some idle processors, parallelizing the computation into 2 GPUs doesn't speed up the program too much.

D. Others

(1) An interesting thing is that originally, I put the code of relaxing paths in another __device__ function called `block_calc`. Theoretically, it doesn't need to synchronize all threads in a block during the phase3. However, when I use `block_calc` function to relax the paths, it always needs to synchronize after calling it. Once I put the code into phase3 function(that is to say, inline the function manually), it doesn't need to synchronize anymore. It is a weird thing when I am coding cuda.

(2) Inline device function manually and use 2D shared memory

Both case running with blocking factor 64 and 32 block dimension

	C17	C18	C19
Inline device functions & 2D shared memory	0.416s	0.486s	0.474s
Original	0.588s	0.930s	0.595s

I use 2D shared memory instead of 1D memory and inline the `block_calc`, `block_calc_rev_async` device functions.

(3)I/O Improvement:

Both case running with blocking factor 64 and 32 block dimension

	C17	C18	C19
Improved I/O	0.588s	0.930s	0.595s
Original	0.657s	0.952s	0.668s

I read the whole input file at one time instead of reading one line each time. It gives a great improvement.

3. Experience & conclusion

Coding CUDA is so tricky to find out what’s wrong with the program. All you can do is try-and-error. Although the program gets faster than the previous version, it still seems to have some bugs.