

NTHU CS542200 Parallel Programming

Homework 3: All-Pairs Shortest Path

111062625 蔡哲平

1. Implementation

- CPU version (hw3-1)

I implemented the *Blocked Floyd-Warshall Algorithm*, denoted as BFW, in this part. As for the thread library, I used OpenMP to parallelize my program. A block is partitioned into columns. I applied a dynamic scheduling technique to distribute each column to a thread using `#pragma omp parallel for schedule (dynamic, 1)`.

- Single-GPU version (hw3-2)

In this part, I implemented BFW using one GPU. Each phase corresponds to a kernel and each GPU block is responsible for a block in BFW.

- ◆ Block factor

Since I accelerated my program using shared memory, the value of the blocking factor, denoted as B , has a maximal value. It will exceed the shared memory size if B is too big, therefore, I set it to 32.

- ◆ Thread num

Each thread is responsible for an element in a block, therefore, the number of threads is set to $B*B$.

- ◆ Matrix size

Matrix size, denoted as mtx_size , defines the number of elements in one row in the 2D-array $dist$. Since the number of vertices, denoted as vtx_num , may not be divisible by B , there should be some bounding check in a kernel. However, using branches in a kernel may cause thread divergence, hence, I used more memory space to prevent this problem. Below is the formula of mtx_size :

$$mtx_size = \text{ceil}(vtx_num / B) * B$$

- ◆ Phase 1

In this phase, there is only one block needed to be calculated, which is the pivot block, hence, there is only one GPU block issued. Each thread is responsible for one element in a block. At first, each thread will load its element from the global memory to the shared memory. Afterward, all calculations will be done in the shared memory, which can significantly reduce the runtime. Each element should be calculated in B rounds and since each round is dependent on the previous round, synchronization is needed at the start of each round. Finally, after the calculation is done, copy the content from the shared memory back to

the global memory.

◆ Phase 2

In this phase, it will calculate all blocks in the pivot row and the pivot column, and those blocks are dependent on themselves and the pivot block. Since there is a dependency between them and the pivot block, it's necessary to load the pivot block into the shared memory. It will issue $2 \times \text{round}$ GPU blocks in this phase. *blockIdx.x* has the value 0 or 1, which denotes whether this block is in the pivot column or not, and *blockIdx.y* denotes the index of the block in the row(column).

◆ Phase 3

In this phase, it will calculate the remaining blocks. Total $(\text{round}-1) \times (\text{round}-1)$ GPU blocks will be issued. Since each block in this phase is dependent on a block in the pivot column and a block in the pivot row, it's necessary to load them into the shared memory.

● Multi-GPU version (hw3-3)

This version is very similar to the single-GPU version. The only difference is that it will use two GPUs in phase 3 to calculate different parts of the whole matrix. GPU₀ is responsible for the top half of the matrix, which are the blocks above the pivot row. In contrast, GPU₁ is responsible for the bottom half of the matrix, which are the blocks below the pivot row. Since GPU₀ needs the correct pivot row to derive the correct top half of the matrix, GPU₁ must send its pivot row to GPU₀ at the start of each round.

◆ Communication

For communication between GPUs, I used *cudaMemcpy(D2D)* to copy the pivot row from GPU₁ to GPU₀.

◆ Simple case explanation

Before the iterations start, it will first load the content in the host memory into each device's memory, denoted as *dist₀* and *dist₁* respectively. At the start of each round, GPU₁ will copy the current pivot row in *dist₁* into the corresponding row in *dist₀*. This way, GPU₀ will be able to derive the correct data in each phase. Below are some figures demonstrating each round. Blocks in green denote the pivot block, which will be calculated in phase 1; Blocks in blue denote the pivot column and the pivot row, which will be calculated in phase 2, and the pivot row is also boldly outlined; Blocks in yellow denote the other blocks, which will be calculated in phase 3. Note that GPU₀ only

calculates the blocks above the pivot row and GPU₁ only calculates the blocks below the pivot row in phase 3. Lastly, the green check means that the block has the correct result after this round. In contrast, the red cross means that the result is incorrect. After all iterations are finished, copy dist_0 back to the host memory since it has the correct result.

V	V	V	V
V	X	X	X
V	X	X	X
V	X	X	X

Round 0: dist_0

V	V	V	V
V	V	V	V
V	V	V	V
V	V	V	V

Round 0: dist_1

V	V	V	V
V	V	V	V
X	X	X	X
X	X	X	X

Round 1: dist_0

X	V	X	X
V	V	V	V
V	V	V	V
V	V	V	V

Round 1: dist_1

V	V	V	V
V	V	V	V
V	V	V	V
X	X	X	X

Round 2: dist_0

X	X	X	X
X	X	V	X
V	V	V	V
V	V	V	V

Round 2: dist_1

V	V	V	V
---	---	---	---

X	X	X	X
---	---	---	---

V	V	V	V
V	V	V	V
V	V	V	V

Round 3(final): dist₀

X	X	X	X
X	X	X	V
V	V	V	V

Round 3(final): dist₁

2. Profiling results (hw3-2)

The biggest kernel is phase 3 and below is the profile result of test case c21.1

Invocations	Metric Name	Metric Description	Min	Max	Avg
Device "GeForce GTX 1080 (0)"					
Kernel: phase2(int*, int)					
313	achieved_occupancy	Achieved Occupancy	0.901231	0.928379	0.914634
313	sm_efficiency	Multiprocessor Activity	84.72%	88.62%	87.03%
313	shared_load_throughput	Shared Memory Load Throughput	1597.8GB/s	1750.7GB/s	1694.3GB/s
313	shared_store_throughput	Shared Memory Store Throughput	66.130GB/s	423.19GB/s	101.83GB/s
313	gld_throughput	Global Load Throughput	65.216GB/s	71.457GB/s	69.155GB/s
313	gst_throughput	Global Store Throughput	32.608GB/s	35.728GB/s	34.578GB/s
Kernel: phase3(int*, int)					
313	achieved_occupancy	Achieved Occupancy	0.968929	0.969342	0.969145
313	sm_efficiency	Multiprocessor Activity	99.86%	99.90%	99.89%
313	shared_load_throughput	Shared Memory Load Throughput	2136.3GB/s	2150.3GB/s	2145.0GB/s
313	shared_store_throughput	Shared Memory Store Throughput	131.97GB/s	570.14GB/s	174.21GB/s
313	gld_throughput	Global Load Throughput	130.79GB/s	131.65GB/s	131.33GB/s
313	gst_throughput	Global Store Throughput	43.597GB/s	43.884GB/s	43.775GB/s
Kernel: phase1(int*, int)					
313	achieved_occupancy	Achieved Occupancy	0.123920	0.124708	0.124597
313	sm_efficiency	Multiprocessor Activity	2.33%	3.10%	2.83%
313	shared_load_throughput	Shared Memory Load Throughput	12.194GB/s	15.549GB/s	14.339GB/s
313	shared_store_throughput	Shared Memory Store Throughput	306.71MB/s	3.3152GB/s	567.95MB/s
313	gld_throughput	Global Load Throughput	260.14MB/s	331.71MB/s	305.91MB/s
313	gst_throughput	Global Store Throughput	260.14MB/s	331.71MB/s	305.91MB/s

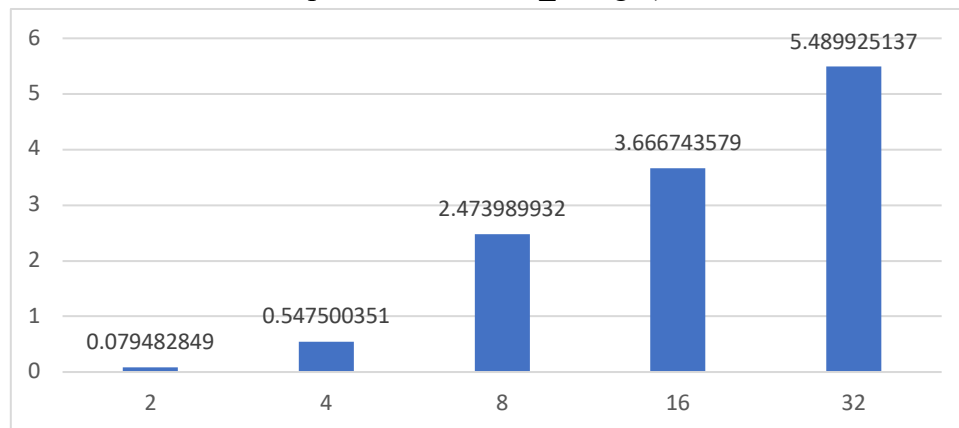
3. Experiment & Analysis

i. System spec (NCHC)

All experiments are conducted on the hades.

ii. Blocking factor

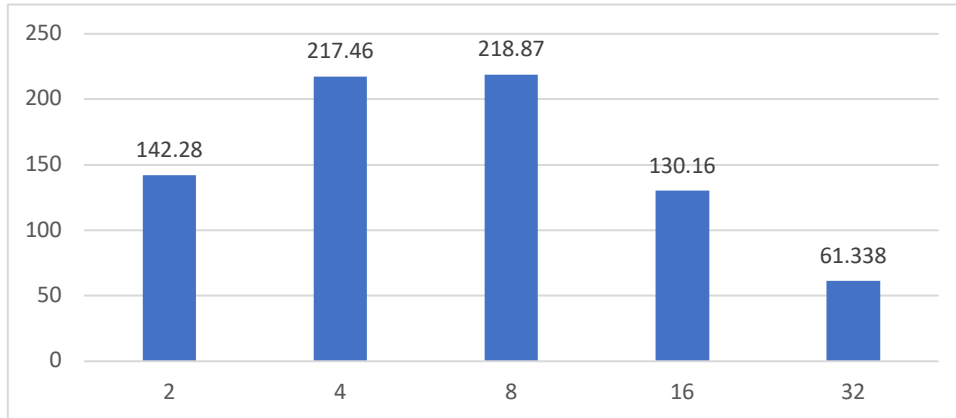
- Experiment 1: Observe how the blocking factor affects GOPS. (Test case: c21.1, command: `nvprof --metrics inst_interger`)



GOPS of different blocking factors
[X: blocking factor, Y: GOPS]

Observation 1: According to the above figure, as the value of the blocking factor increases, GOPS also increases, which means more instructions are executed at a time interval. It's because using a lower blocking factor needs to move data between the global memory and the shared memory much more frequently, leading to significant overhead.

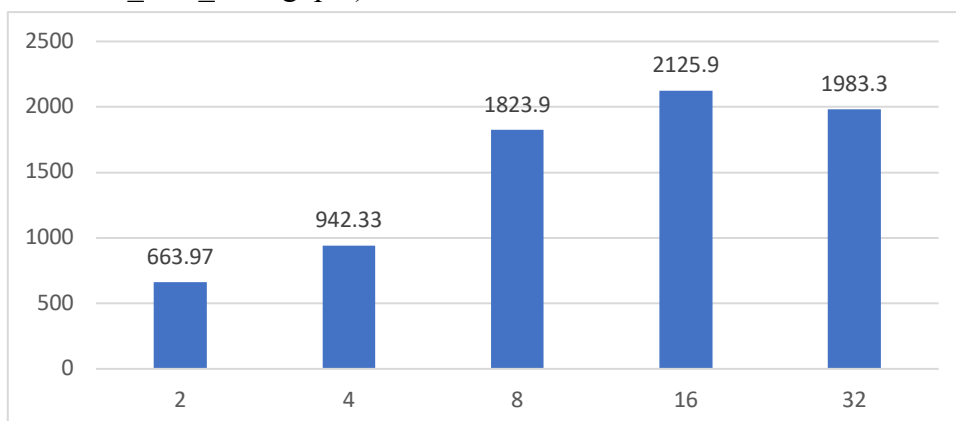
- Experiment 2: Observe how the blocking factor affects the global memory bandwidth. (Test case: c21.1, command: `nvprof --metrics gld_throughput`)



*Global memory bandwidth of different blocking factors
[X: blocking factor, Y: bandwidth (GB/s)]*

Observation 2: According to the above figure, the higher the blocking factor, the lower the global memory bandwidth. This is due to the use of a higher blocking factor can reduce the number of accesses to the global memory. However, there is a drop when B equals 2, which I couldn't figure out why.

- Experiment 3: Observe how the blocking factor affects the shared memory bandwidth. (Test case: c21.1, command: `nvprof --metrics shared_load_throughput`)



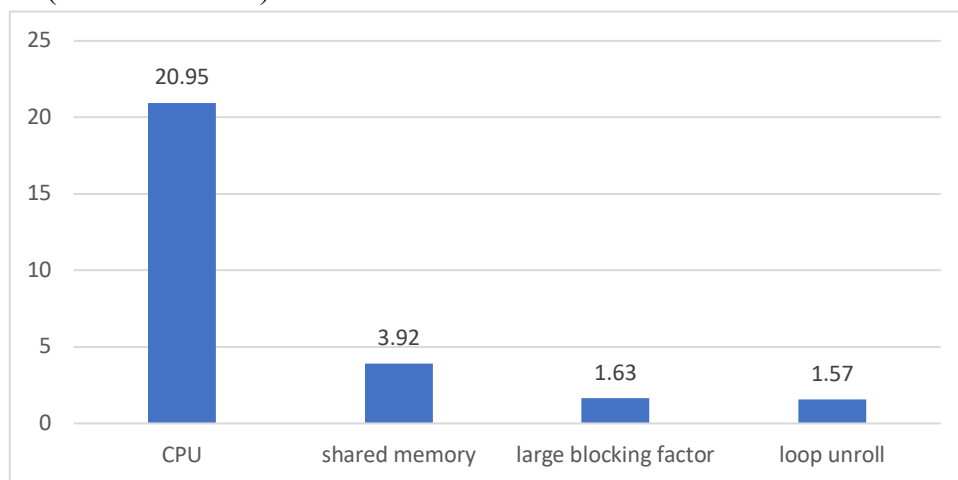
*Shared memory bandwidth of different blocking factors
[X: blocking factor, Y: bandwidth (GB/s)]*

Observation 3: According to the above figure, we can see that as the blocking

factor increases, the shared memory bandwidth also increases. It's because using a higher blocking factor can improve the efficiency of using the shared memory.

iii. Optimization (hw3-2)

- Below are the optimization techniques I used
 - ◆ Blocking factor tuning
 - ◆ Shared memory
 - ◆ Loop unrolling
- Experiment 4: Observe how the optimization technique affects the runtime. (Test case: c21.1)



*Runtime of different optimization
[X: optimization, Y: runtime (sec)]*

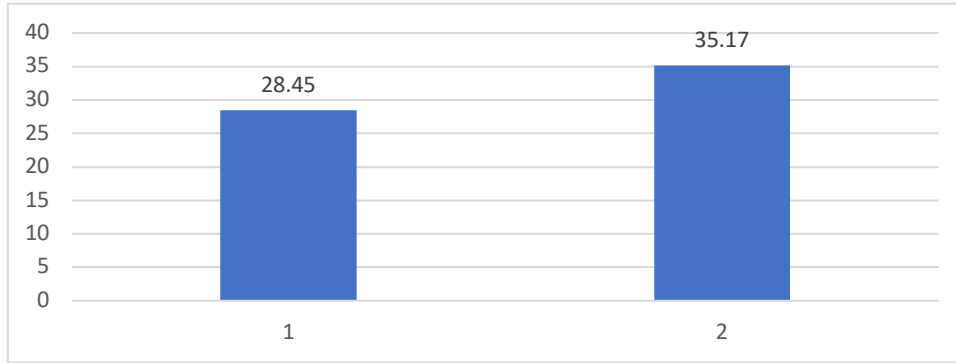
- Observation 4: According to the above figure, we can see that GPU with shared memory applied has a significant runtime decrease compared to the CPU version. Note that the figure didn't show GPU without the shared memory because I used shared memory in my first version of the code, and I found that it was hard to transform it into GPU without the shared memory version. Other optimization such as using different blocking factors and loop unrolling can also slightly improve the performance.

iv. Weak scalability (hw3-3)

The calculation method of weak scalability is based on the fact that the calculation amount of each calculation unit is the same. In this case, compare the execution time required by different numbers of calculation units to complete the calculation. If the execution time is the same, it means that the weak scalability is good.

- Experiment 5: Observe the weak scalability of the multi-GPU version.

(Testcase: single-GPU used p17k1(17000 vertices), multi-GPU used p24k1(24000 vertices); $24000 / 17000 = 1.412 = \sqrt{2}$)



Weak scalability
[X: GPU number, Y: runtime (sec)]

- Observation 5: According to the above figure, we can see that runtime of the multi-GPU version is higher than the single-GPU version, which means the weak scalability isn't good. Since it requires communication between the GPUs, the runtime will of course increase.

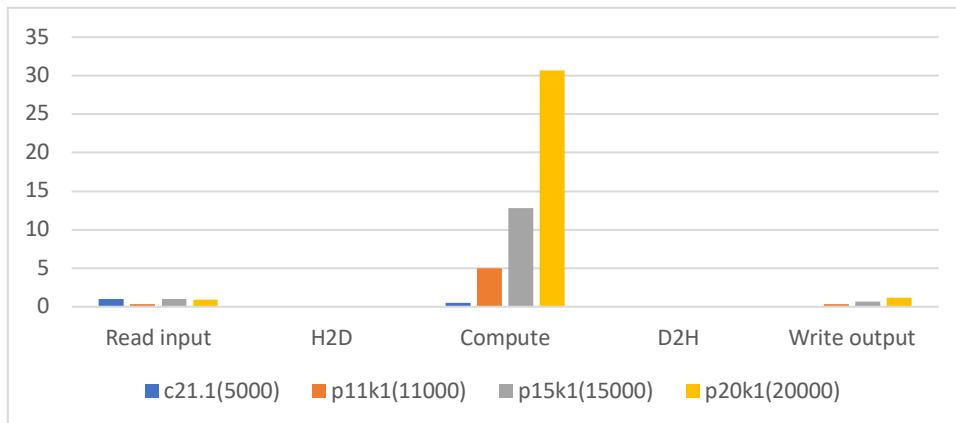
v. Time distribution (hw3-2)

- Analyze the time spent below with respect to input size

- ◆ Read input
- ◆ H2D memory copy
- ◆ Compute
- ◆ D2H memory copy
- ◆ Write output

(Runtime of communication is 0 in the single-GPU version)

- Experiment 6: Observe the time distribution of different input sizes. (Test cases: c21.1, p11k1, p15k1, p20k1)

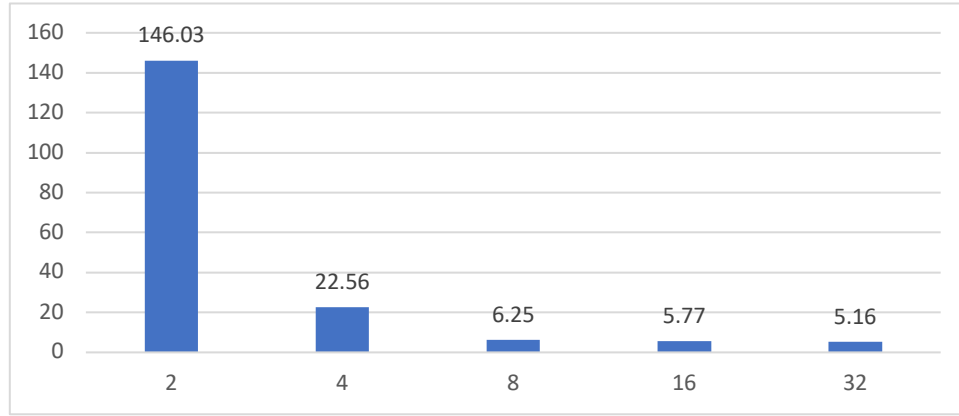


Time distribution of different input sizes
[X: stages, Y: runtime (sec)]

- Observation 6: According to the above figure, we can see that my program's bottleneck is at the BFW computation. In my opinion, it's because of some memory access overhead. Unfortunately, I didn't come up with a good solution.

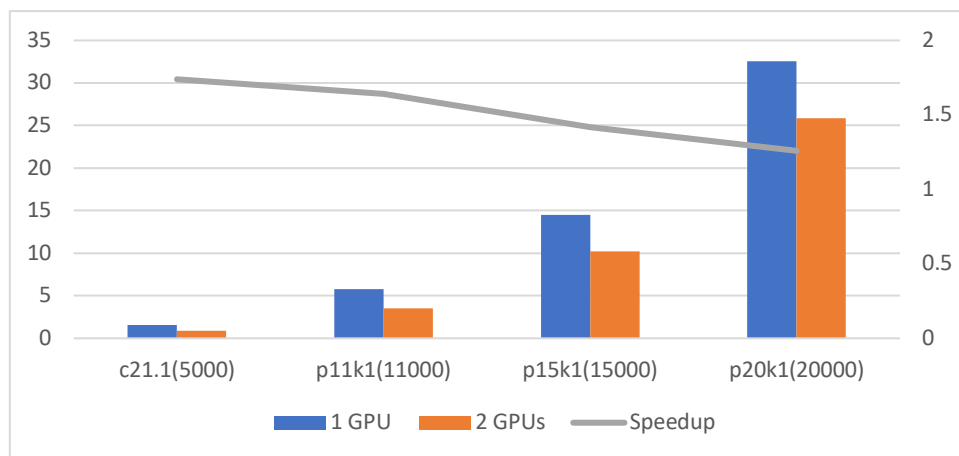
vi. Others

- Experiment 7: Observe the runtime of using different blocking factors (Test case: p11k1)



*Runtime of different numbers of GPUs
[X: blocking factor, Y: runtime (sec)]*

- Observation 7: According to the above figure, we can see that setting B to 2 led to a very long runtime compared to others. Moreover, using a larger blocking factor can shorten the runtime a lot. In my opinion, it's because using a larger blocking factor can utilize the shared memory more efficiently, reducing the number of global memory access.
- Experiment 8: Observe the runtime and the speedup of using different numbers of GPUs on several test cases. (Test cases: c21.1, p11k1, p15k1, p20k1)



*Runtime of different numbers of GPUs
[X: test case, LY: runtime (sec), RY: speedup]*

- Observation 8: According to the above figure, we can see that two GPUs did not speed up twice. And as the number of vertex increases, the speedup decreases. In my opinion, it's because of the communication overhead between two GPUs.

4. Experiences & Conclusion

This homework is very difficult to improve performance. Moreover, I don't have enough time to make my program pass more performance test cases, which is quite frustrating. However, writing a CUDA program is very interesting, and it can significantly reduce the runtime. Hope there's one day I can master the CUDA program and combine it with my research field, which is EDA algorithms, to parallelize them and improve their performance.