CS542200 Parallel Programming
Homework 3: All-Pairs Shortest Path

109062113 葛奕宣

1. Implementation

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

b. How do you divide your data in hw3-2, hw3-3?
Since my block_size is 78 and threads per block is (26,26). So one thread is responsible for 9 data, each data in distance of 1/3*block_size.
Eg.
Thread(0,0) is responsible for:
(0,0), (0,1/3*blocksize), (0, 2*1/3*blocksize),
(1/3*blocksize, 0), (1/3*blocksize, 1/3*blocksize), (1/3*blocksize, 2*1/3*blocksize)…… total 9 data.
In 3-3, I let each GPU do half of the block in phase 3, upper half and lower half.

c. What's your configuration in hw3-2, hw3-3? And why? (e.g. blocking factor, #blocks, #threads)
I choose B(block_size) as 78, threads per block as (26*26). #block is V after padding divided by block_size. The reason for 78 is to utilize the space of share memory. Total shared memory is 49152, we need 2 matrix in phase 2, 3. So, it is sqrt(49152/4/2). Max block_size is 78. And the thread size limit is 1024. So I choose 26, 1/3 of 78. I've tested (78,15) thread size, where each thread is responsible for only 6 data. But the result is worse.

d. How do you implement the communication in hw3-3?
Because each gpu do half of the blocks in phase 3. I need to copy the calculated result after half of the total round is finished. Otherwise, the calculation in phase 1 will be wrong since the data isn't the newest. So, I copy the data back to host and copy the data back to device. The ideal method is to copy from device to device. But I got a worse result.

e. Briefly describe your implementations in diagrams, figures or sentences.
3-1:
I use normal Floyd-warshall alg. Parallel the calculation of second for loop(each i) using openMP.

```
void floyd(){

    for(int k=0;k<V;k++){
        #pragma omp parallel for schedule(dynamic)
        for(int i=0;i<V;i++){
            #pragma omp simd
            for(int j=0;j<V;j++){
                matrix[i*V+j]=  min(matrix[i*V+j],matrix[i*V+k]+matrix[k*V+j]);
            }
        }
    }

}
```

3-2:
I first pads the V in align with block_size.

In each phase, the program first arrange share data, calculate, then write back to device memory.

In phase 1, I use 1 block to calculate the result. Inside a block, a thread is responsible for 9 data.

In phase 2, I use (2* V/blocksize) blocks to calculate the result. Each block is responsible for 1 block(besides the block calculated in phase 1).

In phase 3, I use (V/blocksize, V/blocksize) blockes to calculate all blocks besides the block calculated in phase 1 and 2.

3-3:

The only difference between 3-2 and 3-3 is phase 3. I use 2 gpu to divide the task into 2 half. One gpu calculate the upper half, one is another.

Profiling Results (hw3-2)

I use p12k1 and observe the result from phase3.

occupancy	0.651183
sm efficiency	99.93%
shared memory load throughput	2575.4GB/s
shared memory store throughput	258.14GB/s
global load throughput	48.103GB/s
global store throughput	114.47GB/s

Experiment & Analysis

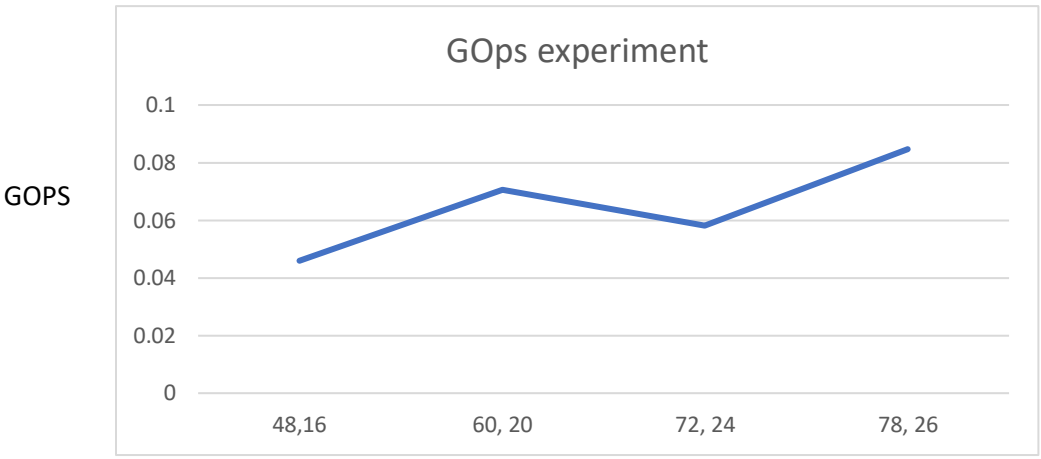
System Spec:
hades

Blocking Factor (hw3-2)

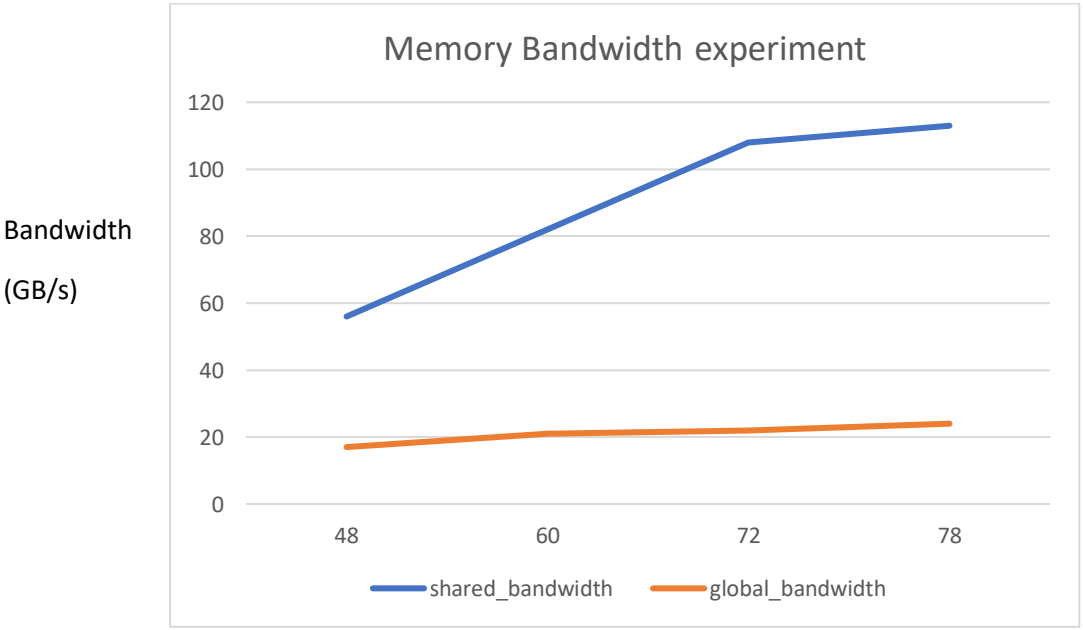
I use c05.1 as testcases.

Bandwidth is calculated by adding load and store throughput.

GOps is calculated by integer instruction / execution time.



For GOPs, when blocking factor increase, the number of threads increase, so the instruction done per time is positive correlated to them. However, we can see 60,20 performs better than 72,24. This may result in different padding result according to the testcases.

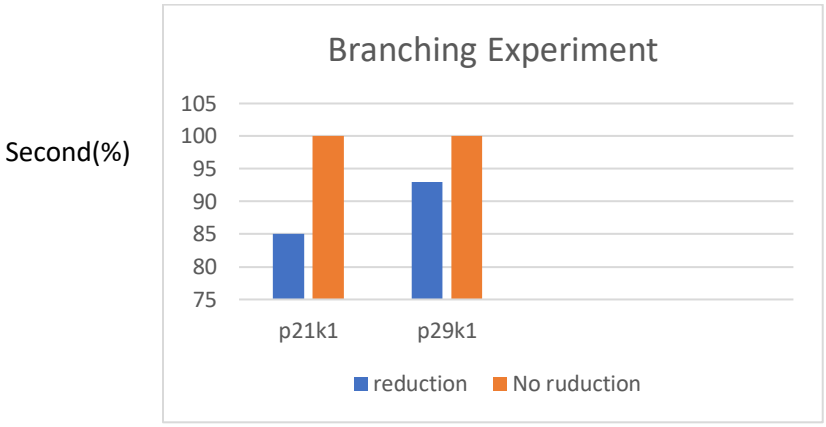


We can see by bandwidth experiment. When Blocking factor is large, the more we utilize the share memory, so the bandwidth raises. Since the use of global memory doesn't change in my program, the bandwidth remains the same.

Optimization (hw3-2)

1. Utilize the share memory space. Experiment is similar to the blocking factor above.

2. Use unroll to reduce branching and change if else statement to bitwise operation. There's no branching in my kernels.



3. Use memcpyAsync. Overlaps the copy time.

4. I tried the pitch and 2Dmemcpy, but the result is worse.

5.Cut down the use of metrics in share memory. In phase 3, I use 9 register instead of share memory for the local value.

```
/*calculation*/
#pragma unroll
for(int k=0;k<block_size;k++){
    ans1 = min(ans1, share_row[y][k]+share_col[k][x]);
    ans2 = min(ans2, share_row[y][k]+share_col[k][x+block_size_index]);
    ans3 = min(ans3, share_row[y][k]+share_col[k][x+block_size_index*2]);
    ans4 = min(ans4, share_row[y+block_size_index][k]+share_col[k][x]);
    ans5 = min(ans5, share_row[y+block_size_index][k]+share_col[k][x+block_size_index]);
    ans6 = min(ans6, share_row[y+block_size_index][k]+share_col[k][x+block_size_index*2]);
    ans7 = min(ans7, share_row[y+block_size_index*2][k]+share_col[k][x]);
    ans8 = min(ans8, share_row[y+block_size_index*2][k]+share_col[k][x+block_size_index]);
    ans9 = min(ans9, share_row[y+block_size_index*2][k]+share_col[k][x+block_size_index*2]);
}
```

6.Padding the input matrix so that it is aligned with the block_size. This is to reduce the branch condition inside kernel. Since we have enough blocks, the extra computation overhead is lower than the warp divergence resulted by branch conditions.

Weak scalability

I use p31k1 as testcase. Use nvprof to get data.

	Single phase 3 Computational time	Total time	Memcpy time
Multi-GPU	28.3s	24.9s	1.7s
Single-GPU	56.6s	24.3s	0.6s

There's no much difference between multi and single gpu. The single phase 3 computational time is truly cut into half in each gpu. However, the memcpy time in Multi-GPU spend on communicating data yields a large overhead compared with single gpu.

Time Distribution

Testcases: p31k1. Use nvprof and clock_gettime() to get the data.

- computing
  - phase1: 23.6ms
  - phase2: 429ms
  - phase3: 22s
- communication
  - CudalaunchKernel: 3.6ms
- memory copy (H2D, D2H)
  - H2D: 327ms
  - D2H: 292ms
- I/O of your program w.r.t. input size. (Input size: V=31000, E=15125277)
  - Input: 1.13s
  - Output: 13.22s

We can see the phase3 occupies 95% of the total execution time. So optimization should start with phase 3. This program is bounded by computational time and output.

Experience & conclusion

This homework assignment is really hard. I spend a lot of time optimizing my program and testing the configuration. I found out profiling tools are really useful when dealing with problems like bank conflict, share memory using……. I am more familiar with cuda now :).