CS542200 Parallel Programming

Homework 4-1: Blocked All-Pairs Shortest Path

109065517 林佳縈

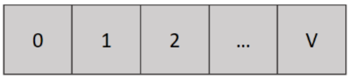
1. Implementation

a. Data division

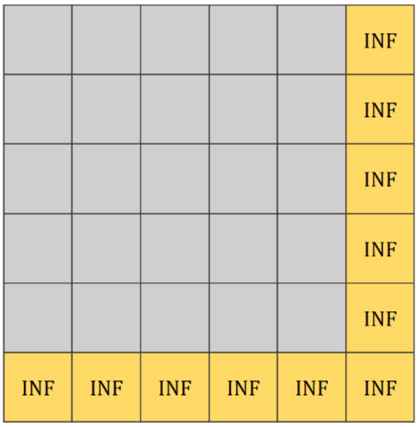
For how I divide my data, I'd like to introduce them in the following corresponding sections.

Host to Device

When the program received the input file, it will be stored like this:



input data



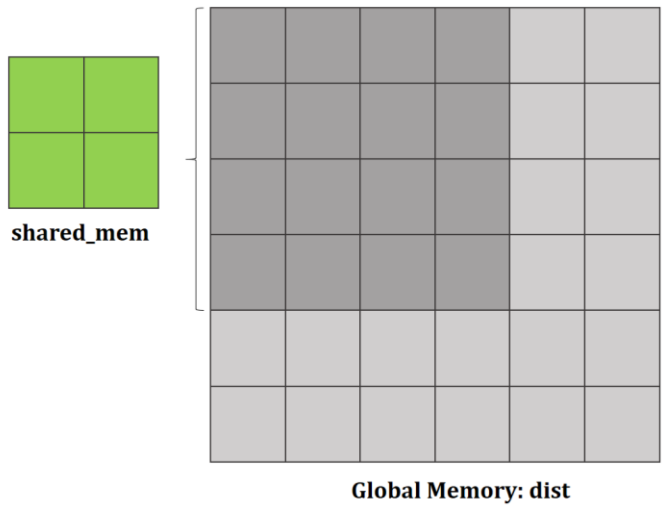
host_ptr: adjacency matrix

The gray part of size V*V will store the adjacency matrix, while the light brown part is used to cases such that if number of nodes can't be evenly divided by the blocking factor, which the out-of-range part will then be an unexpected number (mostly, 0), it has to make sure that the out-of-range part will never affect the correct output. It's like padding, with value INF stored inside.

Note that, the above 2D graph is drawn for visual simplicity, the program stored the adjacency in 1D way.

Phase 1

For phase 1, after loading the entire host_ptr to device_ptr with cudaMemcpy2D, I loaded the complete device_ptr to phase 1's kernel. But since only a pivot block has to be calculated, I then load the corresponding part into shared_memory for acceleration.



Suppose we have blocking factor as 4, the dark-gray of global memory denotes the part that to load into shared memory. Since I have a 2D thread, so I assigned it like this:

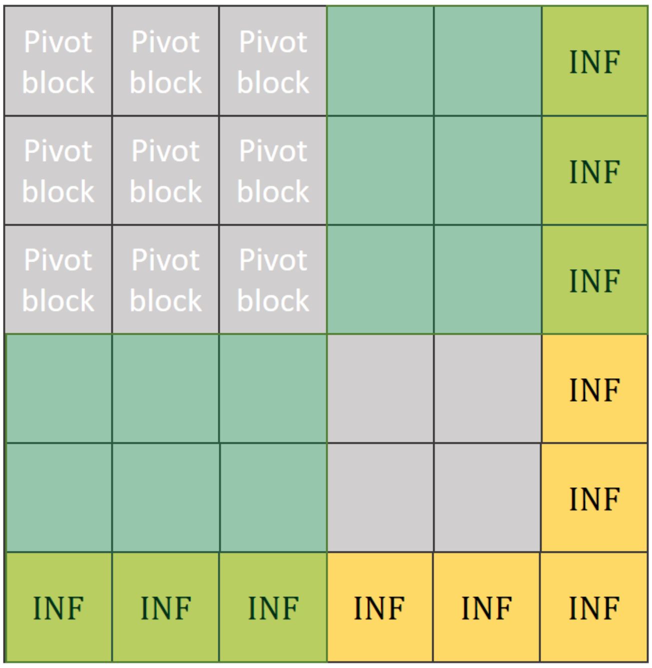
```
int tid = threadIdx.y * B + threadIdx.x;
shared_mem[tid] = dist[j*V+i];
```

Phase 2

For phase 2, pivot row and pivot column should be calculated, which is drawn as the blue-filtered area. Same as the previous logic, padding with value INF is add to the matrix to prevent wrong answer; and data will be divide and loaded into shared memory by the following:

Since the pivot block is already calculated, so I calculated the shared memory index with:

```
shared_mem[tid + B*B] = dist[j*V + i];
shared_mem[tid] = dist[(Round*B+threadIdx.y)*V + (Round*B +
threadIdx.x)];
```

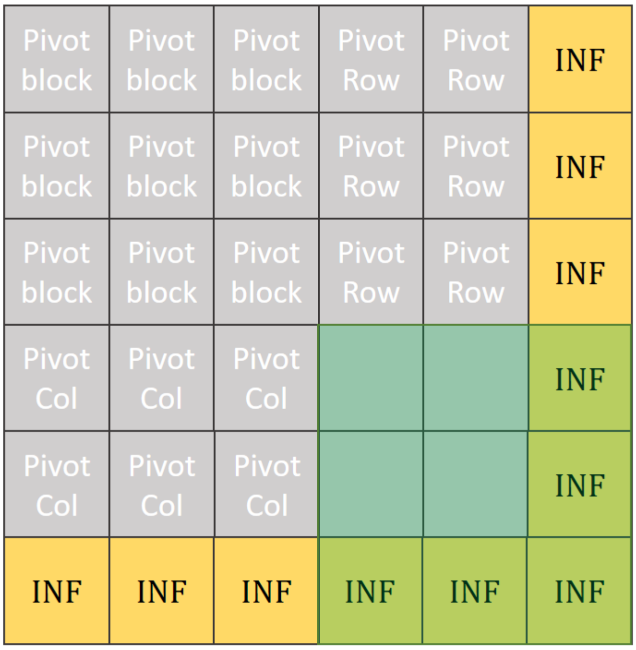


Device adjancency matrix: dist

Phase 3

For phase 3, the blue-filtered part is calculated. Since part other than Pivot block, pivot row and pivot column, I load the global memory matrix to the following index on the shared memory:

shared_mem[tid], shared_mem[tid+B*B], shared_mem[tid+B*B*2];



Device adjancency matrix: dist

b. Configuration

For my configuration, I have the following settings:

- blocking factor: 16
- blocks: initialize with {1,1}, but will be altered during different phase.
- threads: 16

Blocking Factors:

The reason I set blocking factor 16 is because it is the most stable one I've tried. During many trials, I have tested blocking factors with 2, 4, 8, 16, 32. When it's set 2 or 4, for cases with larger nodes/edges, I could easily be a timeout; while if it's set to be 8, there're some cases with unexpected errors, that's to say, wrong answer; also, for blocking factor 32, since there seems to be no significant improvements on performance, I then didn't choose to have 32 as my blocking factor. Therefore, after few trials, I found 16 to be most stable one.

Blocks

Blocks' numbers depend basically on different phases, and the following is how I plan to count the shortest path:

- ***<u>Phase1:</u>***

{1,1}

- ***<u>Phase2:</u>***

{1, 0~round}, {0~round, 1}
{1, round-(0~round)-1}
{round-(0~round)}

- ***<u>Phase3:</u>***

{round-(0~round)-1, round-(0~round)-1}
{(0~round), (0~round)}
{round-(0~round)-1, round-(0~round)-1}
{0~round, round-(0~round)-1}
{round-(0~round)-1, (0~round)}

Threads

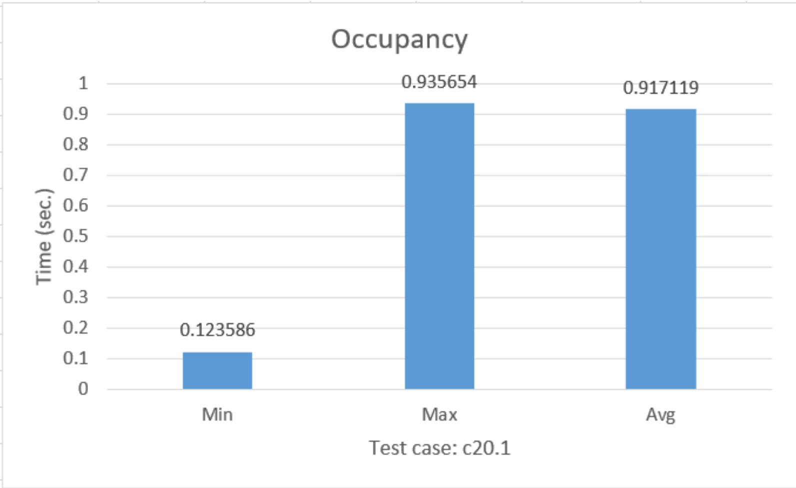
For threads number, I chose {16,16}. First of all, I've tried to make it with one-dimension, but then found that to pass performance cases, I should make best use of threads (blocks also). So I changed it to two dimension. Before the final decision on having my threads number as 16, I've also tried {8,8} once, but then found 16 also works, so I just simply think that the more the better.

2. Profiling Results

For time profiling, I chose testing case c20.1 to see the results.

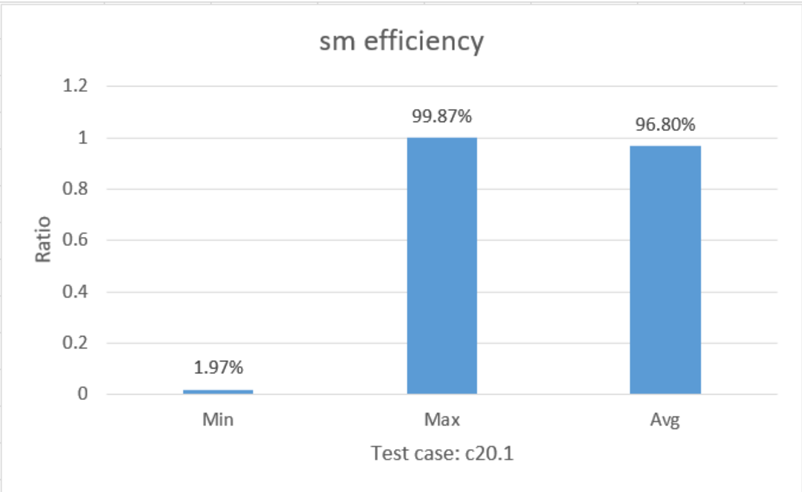
a. occupancy

For occupancy: --metrics achieved_occupancy



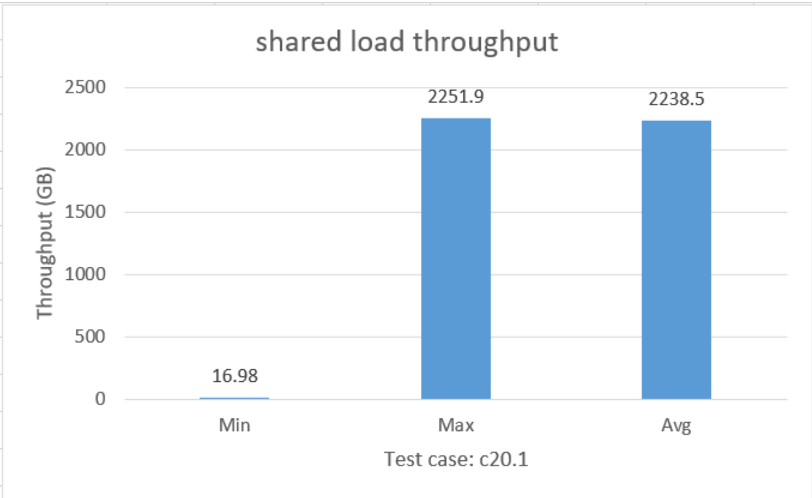
b. sm efficiency

For sm efficiency: --metrics sm_efficiency

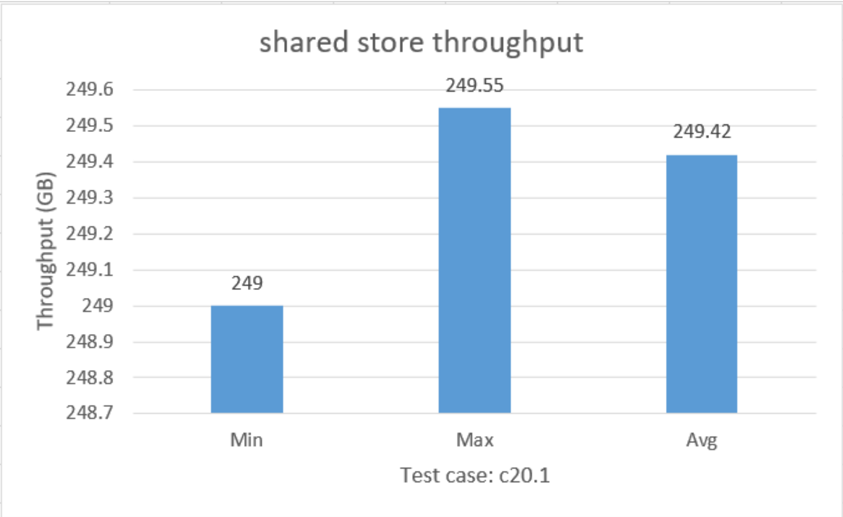


c. shared memory load / store throughput

For shared memory load throughput: --metrics shared_load_throughput

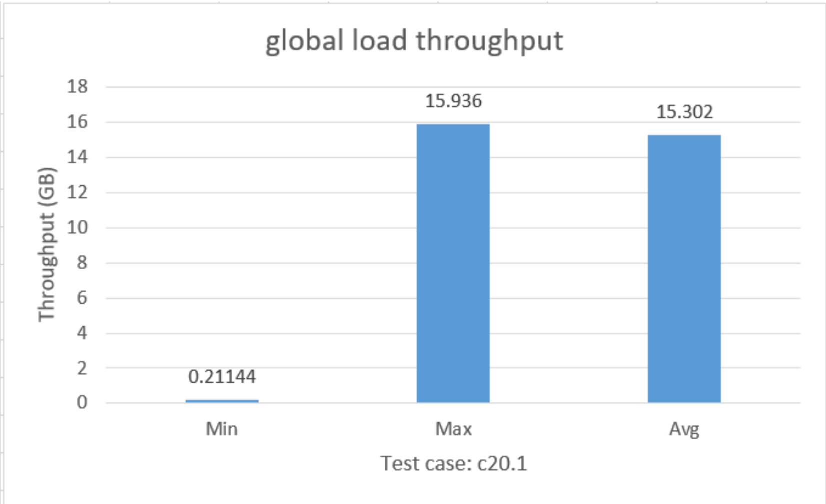


For shared memory store throughput: --metrics shared_store_throughput

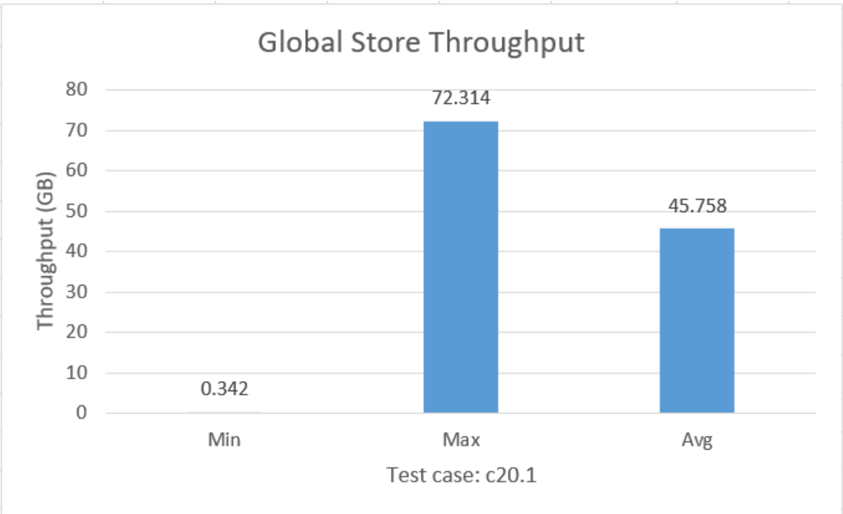


d. global load / store throughput

For global load throughput: --metrics gld_throughput



For global store throughput: --metrics gst_throughput



3. Experiment & Analysis

a. System Spec

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

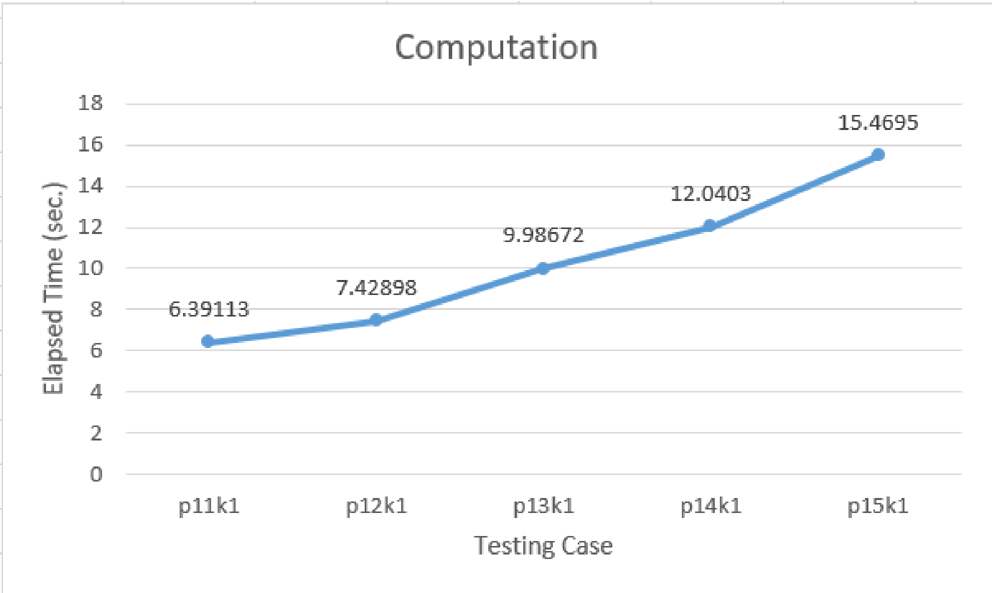
b. Time Distribution

To measure time distribution, I measured them separately. For measuring computing and memory copy, I use nvprof to see the profiling results and also use the cuda event to see the total computation time taken in a program; while for I/O, I measured it with clock_t by library <time.h>.

- Computing

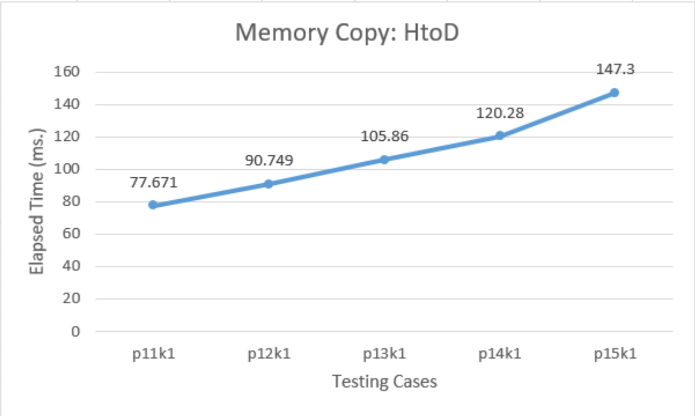
```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);
...
cudaEventRecord(stop );
cudaEventSynchronize(stop);
```

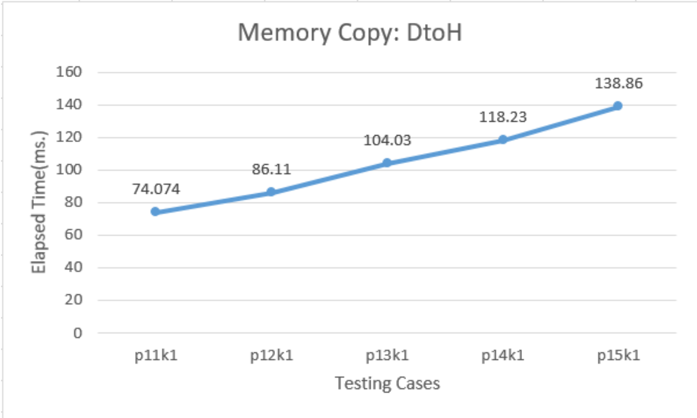
This is how I compute the computation time spent, by creating cuda events and set start and stop flags that wrap up the part of code of which time spent I'd like to compute. For testing cases, I chose cases of p11k1~p15k1, and this is the result:



- Memory Copy (H2D, D2H)

For memory copy, I simply use **nvprof** to check the time spent. For testing cases, I chose p11k1~p15k1 and this is the result:



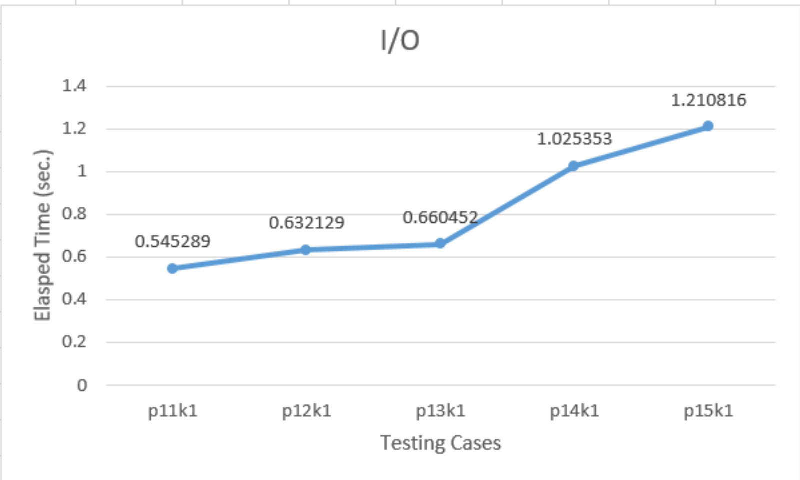


- I/O

For I/O time spent. I use the clock_t provided by <time.h> to measure. The logic is same as cudaEvent but it measures the code other than cuda part instead.

```
#include <time.h>
clock_t start, end;
double cpu_time_used;
start = clock();
 ... /* Do something */
end = clock();
IO_time_spent = ((double) (end - start)) / CLOCKS_PER_SEC;
```

This is how I measure the I/O time spent. To actually see the difference, I chose testing cases of c01.1, the least nodes/edges, and p11k1~p15k1. This way, we can compare the difference of time spent w.r.t quantities of nodes/edges.



c. Blocking Factor

For blocking factor, I chose testing case c03.1 due to server time limit. I calculated GOPS by Inst_integer/Elapsed Time.

Blocking factor	Inst_integer	Elasped time	GOPS
8	15829	0.44	35975
16	37632	0.37	101708
32	147456	0.28	526628

d. Optimization

- **Shared Memory**

Profiling by nvprof, the third kernel (phase 3) charges for nearly 98% of the total computation time. To reduce the total time, I put some variables in the third kernel into shared memory:

Before:

*int tmp = share_p_r[x * bsz+k] + share_p_c[k * bsz + y];*

if (tmp < dij) dij = tmp;

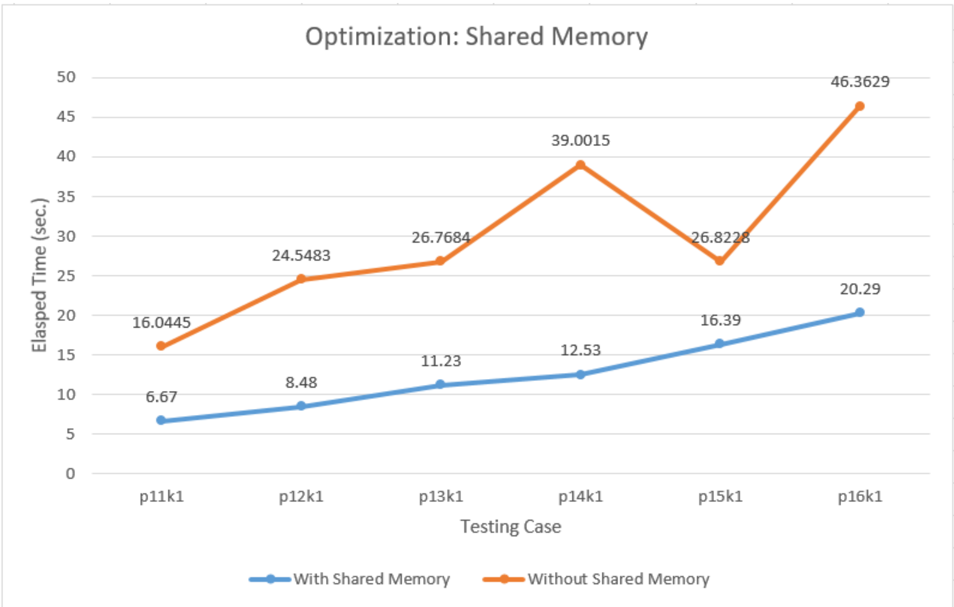
, where dij is actually from global memory. Therefore, the *dij=tmp* statement is thus a huge bottleneck.

After:

*int tmp = shm[kn1*B + in1+B*B*2] +shm[jn2*B+kn2+B*B]*

shm[tid] = min(shm_mem[tid], tmp);

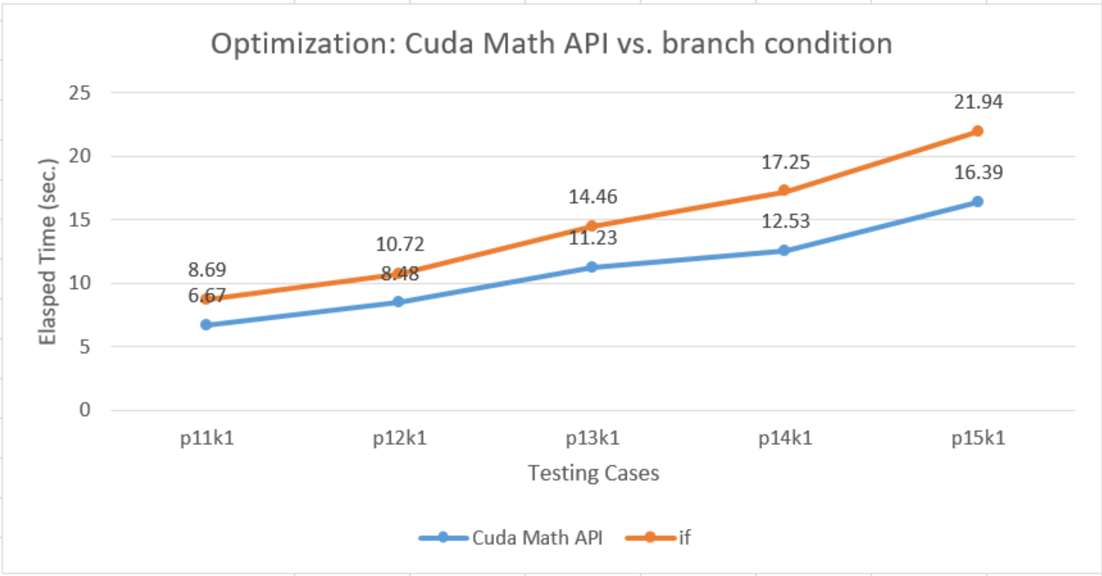
, by assigning every used value into share_memory, this is the time reduced:



So as can be interpreted from the graph, we can tell that the blue line which indicates the time spent for kernel three is way efficient than the orange line.

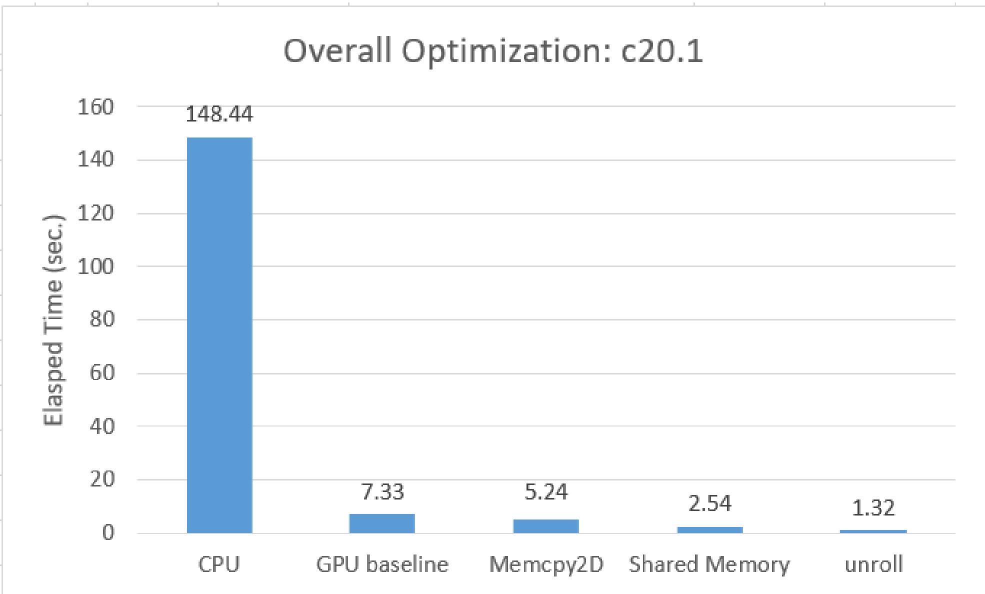
- **Branches replacement**

  As branches cost a lot of computation resources (time) and each time when I use **nvprof** to check bottleneck of the whole program, I see kernel for phase3 is just a huge bottleneck that spent a whole lot of time. After some revision, such as using share memory instead of global, I found the "if" statement is still quite wasting time; therefore, I tried to replace the if statement by __device__ min provided by Cuda Math API.



- Overall Optimization Result

  To show the overall Optimization result, I have the following chart. I test the CPU version on Apollo, while the other starting from GPU



baseline, I tested on hades server.

4. Experience & Conclusion

I heard that the goal of this assignment is to have us actually see how fast coding with GPU could be, and yes, I've experienced it. Also, accompanied with the more efficient algorithm, the total time reduced made me in awe. But I'd say, the toughest job in this assignment for me is to have my mind twisted when transferring the sequential code to cuda version. It's way not simple, not only the part of calculating index of different threads, blocks or so, but the whole structure is not that easy to think of or to say, implement for me. But yes, half goal is reached!