Parallel Programming - All Pairs Shortest Path

1. Implementation
    a. Algorithm, Data and Implementation (hw3-1)

    For the algorithms, I am still using the blocked Floyd Warshall algorithm, the same as the rest of the homework. I am using openmp and vectorization for hw3-1. I am using openmp to parallelize the part where each vertex inside a block needs to calculate its shortest distance to another vertex in either the pivot block or its own block (phase 1). Next, I also use the vectorization technique from our previous assignment, since the vectorization is for integers, I can load 4 values of the matrix at once, based on the result in the scoreboard, it seems like this technique has improved the execution time. In addition, since in the class, unroll is said to be an optimization technique as well, I also try to unroll the for loop, when initializing the distance matrix (to either 0 or INF).

```
for (int k = Round * B; k < (Round + 1) * B && k < n; ++k) {

    #pragma omp parallel
    {
        simd d, b, cmp;
        __m128i a;
        #pragma omp for collapse(2) schedule(static)
        for (int i = block_internal_start_x; i < block_internal_end_x; ++i) {
            for (int j = block_internal_start_y; j < block_internal_end_y; j+=4) {
                a = _mm_set1_epi32(Dist[i][k]);
                b.v = _mm_loadu_si128((__m128i const*) &Dist[k][j]);
                d.v = _mm_loadu_si128((__m128i const*) &Dist[i][j]);
                b.v = _mm_add_epi32(a, b.v);
                cmp.v = _mm_cmplt_epi32(b.v, d.v);
                if(cmp.vs[0]) Dist[i][j] = min(b.vs[0], INF);
                if(cmp.vs[1]) Dist[i][j + 1] = min(b.vs[1], INF);
                if(cmp.vs[2]) Dist[i][j + 2] = min(b.vs[2], INF);
                if(cmp.vs[3]) Dist[i][j + 3] = min(b.vs[3], INF);
            }
        }
    }
}
```

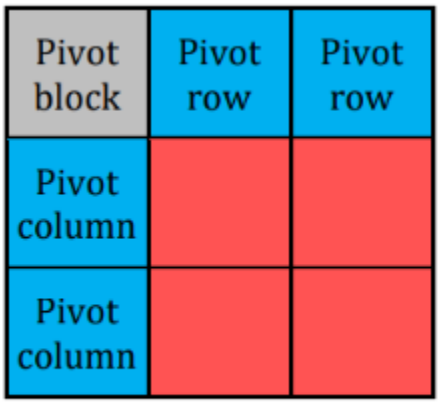
```
#pragma unroll 64
for(int i = 0; i<n; ++i){
    Dist[i] = (int*)malloc(n * sizeof(int));
    for(int j = 0; j < n; ++j){
        if (i == j) {
            Dist[i][j] = 0;
        } else {
            Dist[i][j] = INF;
        }
    }
}
```

b. Configuration for hw3-2, hw3-3, and Reason

The maximum number of threads for each block is 1024, and I want to use 2 dimensions of blockSize so I chose to execute each phase with kernel block size **(32, 32)**. However, from what I configured when listening to the class lecture, the optimal value of using the shared memory, when we need to store 3 integer matrix is with “TILE WIDTH” 64. so each thread in the block will calculate the result for 4 pixels, which means that the **blocking factor** is **64**.

Next, to avoid the bank conflict, linear mapping is the easiest way in this case, so I assigned the pixel in round-robin fashion. Thread 0 will be responsible for matrix (0,0), (32, 0), (0, 32) and (32,32). However, there will be a segmentation error if the number of vertices is not multiple of 64, so in order to avoid branching inside the for loop kernel function, I use the padding method to deal with the problem, so if the matrix size is not multiple of 64, I increase the matrix size to the nearest multiple of 64, i.e: 10 -> 64. Although there will be some unnecessary calculations, but it is better than branching in the kernel function. Every thread only needs to move the value that it needs for its own calculation from global memory to shared memory.

Grid size:



(c) Phase 3

(round is the matrix (row/column) size after padding divided by the blocking factor, i.e: in the figure above, the round is equal to 3)

Phase 1 : (1,1) -> because only need to calculate the pivot block each round

Phase 2 : (round, 1) -> because only need to calculate the blue part of the above figure and I let each thread calculate 1 of the pivot row and 1 of the pivot column.

Phase 3 : (round, round) -> because need to calculate all of the remaining blocks, and skip the calculation when the blockIdx.x = round (blue) or blockIdx.y = round (blue).

c. Data and Implementation (hw3-2, hw3-3)
Another optimization that I have done is, the comparison inside the for function inside the kernel function, instead of using if to compare the current distance with the previous distance, I use min(), to directly take the minimum value. Moreover, for copying the memory from host to device, I used cudaMemcpyAsync, so that I can overlap some of the CPU calculation, and I also pinned the host memory using cudaMallocHost.

The details for each phase:
For phase 1, since the values needed are all from the pivot block, so each thread only need to move 4 value to the shared memory. Additionally, since in this phase, the value inside the pivot block will change every iteration inside the kernel function, we need to sync the thread, to assure

that all values have been successfully updating before continuing to the next iteration. Then, I also unroll the for loop inside the kernel function.

For phase 2, initially, I want to separate this phase into 2 sub-phases, namely 1 horizontal and 1 vertical, but this way the pivot block will be needed to be loaded into the shared memory twice, and I want to separate both of them into different streams, but after running nvprof, the result is that most of the time spent in waiting the streams synchronization, so at last I try to combine both horizontal and vertical to one. As a result, one thread will be needed to load 3 values from 3 blocks (pivot block, pivot row, and pivot column), then since one thread is responsible for 4 matrix values in each block, so one thread will be needed to copy 3*4 = 12 values to the shared memory. Next, since each thread in the block is independent of the others, we do not need to sync threads before continuing to the next iterations. Then, I also use unroll and use min() instead of if, for the “for” loop inside the kernel function in the same fashion as phase 1.

For the last phase, each thread still needs to load values from 3 blocks, 2 blocks from phase 2 (blue), and 1 for itself (red), which means that the total is the same as in phase 2, 3*4 = 12. For phase 3, I actually try to do them in different streams as well, but the result is the same as before (as described in phase 2).

The difference in hw3-3 compared to hw3-2:

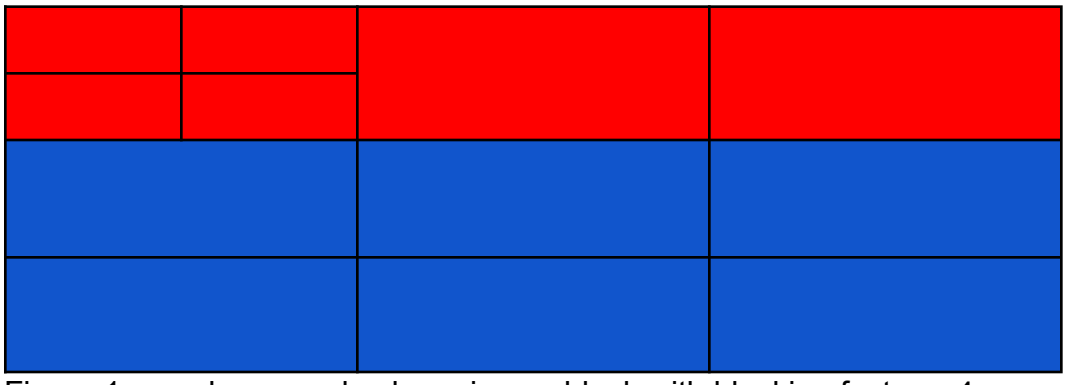


Figure 1a: each row and column is one block with blocking factor = 4

For hw3-3, I used openmp to let one thread manage 1 GPU, and the total block of the data is divided into 2. GPU 0 will get the upper part (red) and GPU 1 will get the lower part (blue). For simplicity, if the total round is not

divisible by 2, then GPU 1 will get 1 more round/row (blue), compared to GPU 0(red). Then, since the time spent in doing phase 1 and 2 are faster compared to the communication time to pass the data across the GPU, both GPUs will calculate their own part of phase 1 and 2, and for phase 3, it will be divided into 2, and then after finishing calculating the phase 3. Before going to the next iteration, since the pivot block is the only block needed in the next round phase 1, the GPU responsible for the pivot block will need to send the corresponding pivot block to the other GPU, so that the other GPU can calculate the phase 1 correctly, which means that for each iteration, the data passes between the GPU is n_after_padding * Blocking factor * sizeof(int), which is the size of one pivot row.

Next, after finishing calculating the shortest path for all vertices, each of the GPUs will send its own part of the data to the host.

d. Communication in hw3-3
For hw3-3, I use the p2p communication method, so that the memory copy does not have to go through CPU/Host for moving the data across different GPUs. so the cudaMemcpy's last parameter is cuda device to device. Then, since every GPU will recalculate phase 1 and 2, we only need to pass the data of the pivot block needed in phase 1, across the GPU.

2. Profiling Results (hw3-2)
The test case I used is p20k1 and p11k1, because the time spent in the scoreboard is not too long and not too short (~9s) for the former. As for p11k1, it was because p20k1, would have encountered a time limit when doing the blocking factor experiment. Next, my biggest kernel is the kernel of phase 3 .

p20k1	Occupancy	SM Efficiency	Shared Memory Throughput		Global Load/Store Throughput	
			Load (GB/s)	Store (GB/s)	Load (GB/s)	Store (GB/s)
Min	0.937	99.93%	3273.7	269.61	16.375	64.318
Max	0.941	99.99%	3403.8	277.52	18.376	66.730
Avg	0.939	99.98%	3364.9	274.59	16.859	65.545

p11k1	Occupancy	SM Efficiency	Shared Memory Throughput		Global Load/Store Throughput	
			Load (GB/s)	Store (GB/s)	Load (GB/s)	Store (GB/s)
Min	0.935	99.85%	3144.1	256.81	64.696	64.696
Max	0.938	99.95%	3410.9	261.29	69.413	69.413
Avg	0.937	99.93%	3270.6	259.27	68.886	68.886

3. Experiment and Analysis
    a. System Spec
    Hades

    b. Blocking Factor (hw3-2) :
    Test case: p11k1

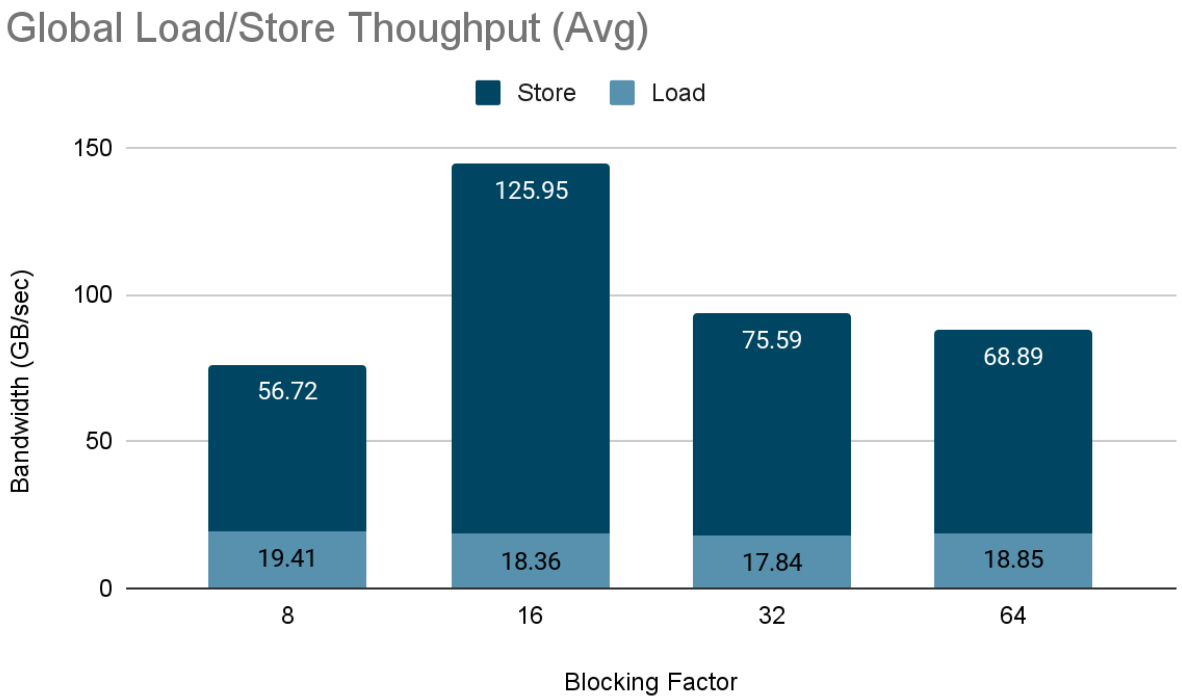


Figure 3b-1

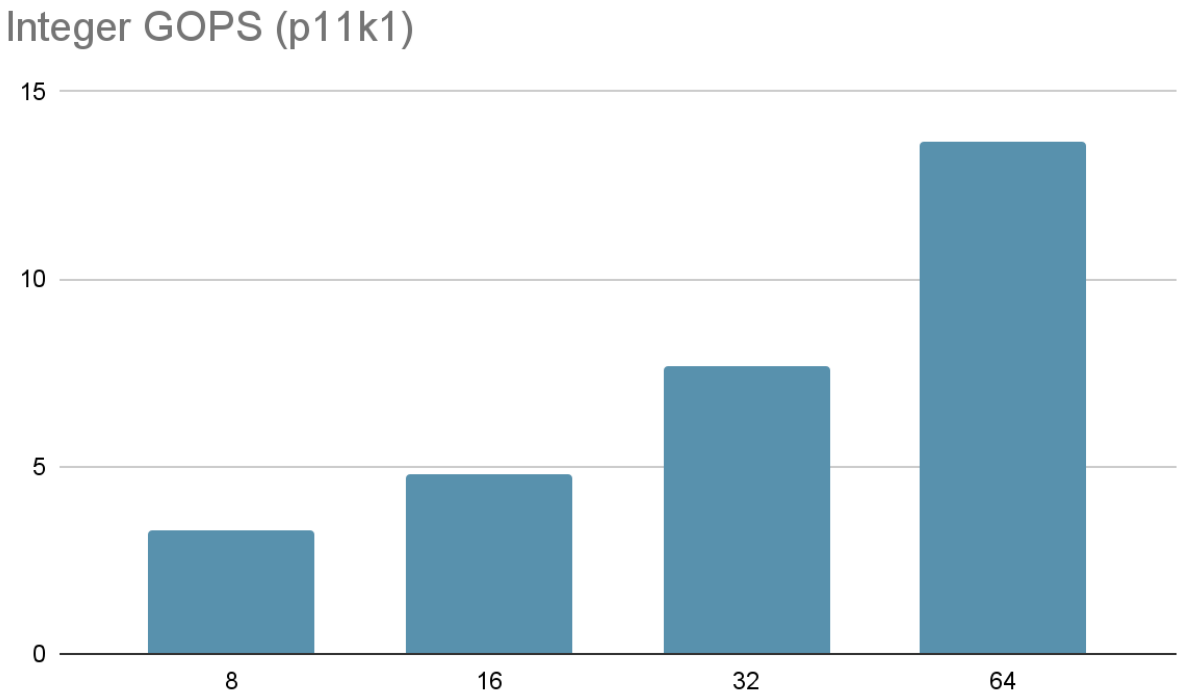


Figure 3b-2

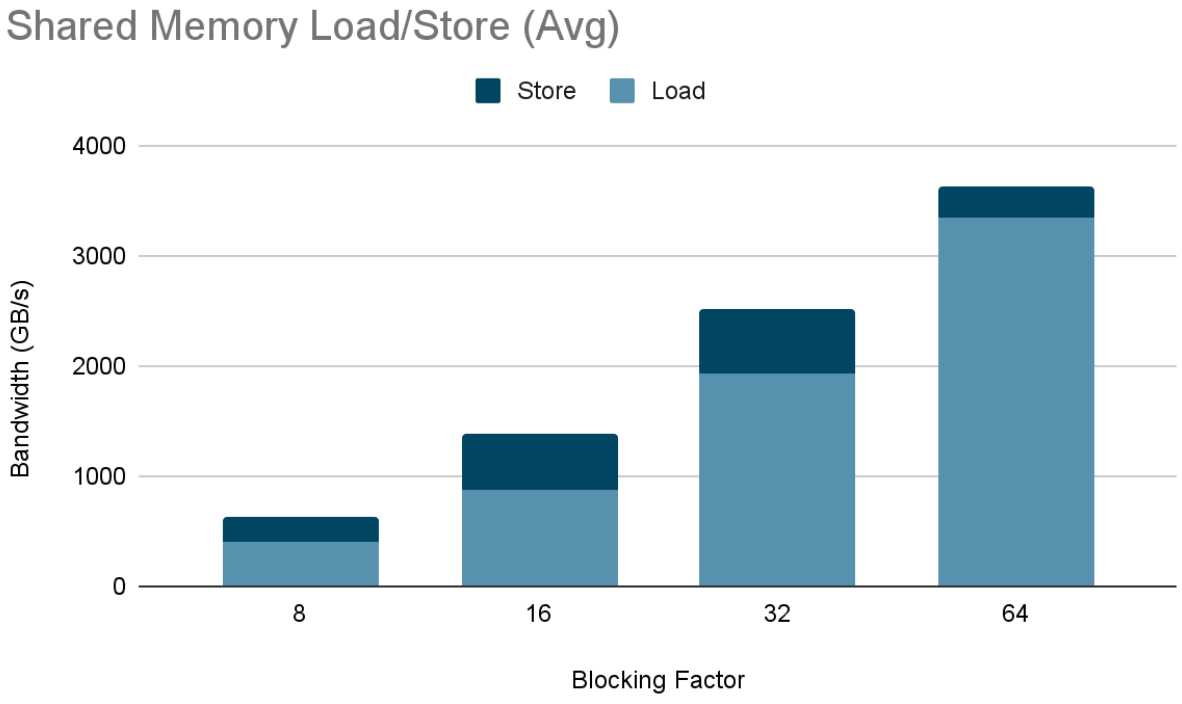


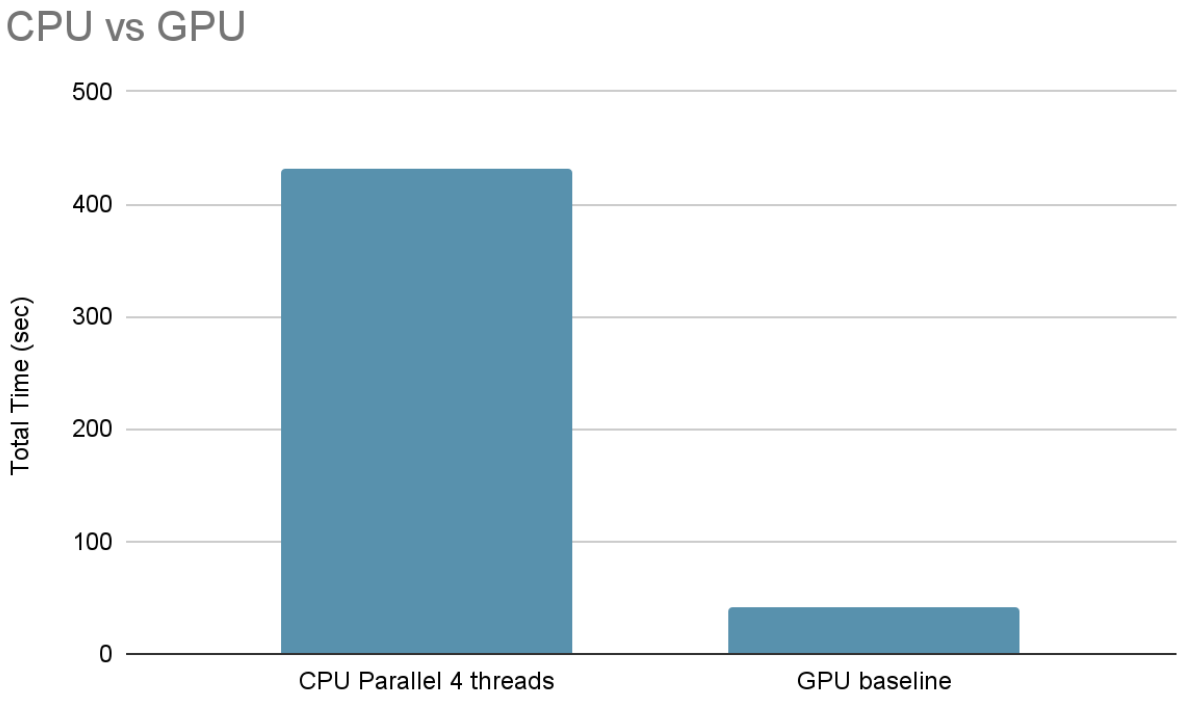
Figure 3b-3

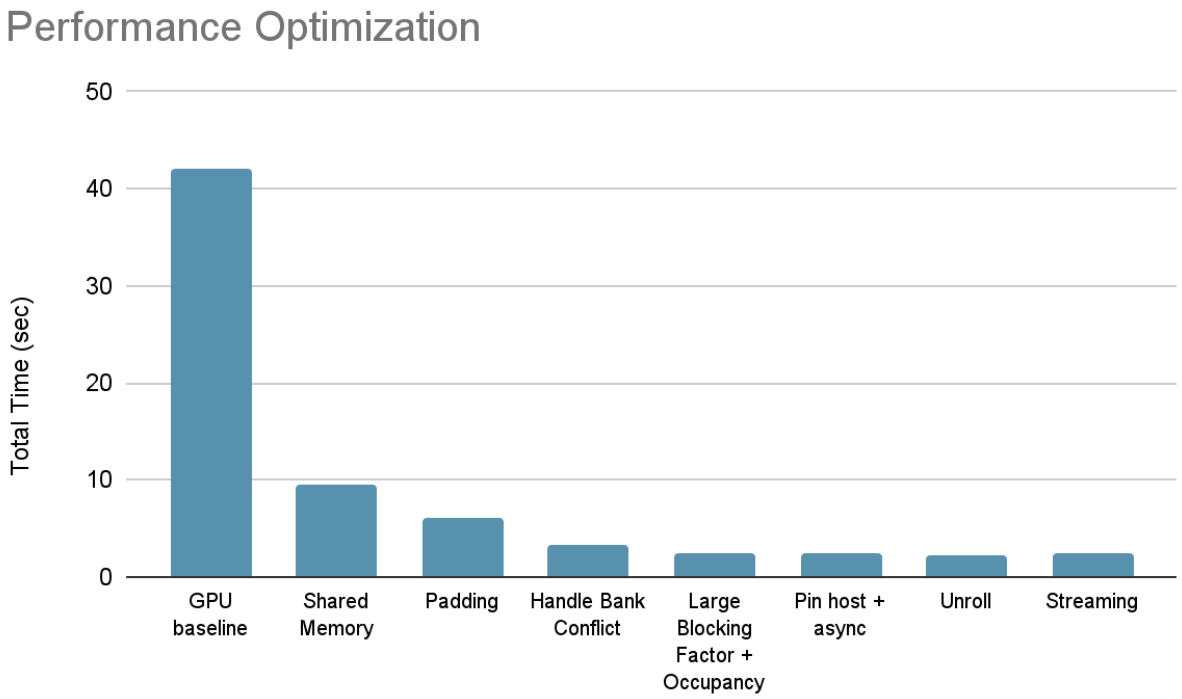
Note:
Since, the maximum number of threads is 1024, which means that the maximum block size for the kernel is (32,32), the maximum blocking factor I am using is 64, (each thread is responsible for calculating 4 values).

Conclusion:
From all of the figures above, we can see that the blocking factor of 64 has the best result, it has the highest integer operation and highest shared memory load/store bandwidth.

c. Optimization (hw3-2)
For the CPU version, it was from hw3-1 with 4 threads.

GPU Optimization:
The details are also written in the implementation section for hw3-2. In addition, I also access the memory in a coalesced manner, threads in the same warp will access the memory sequentially instead of striding.

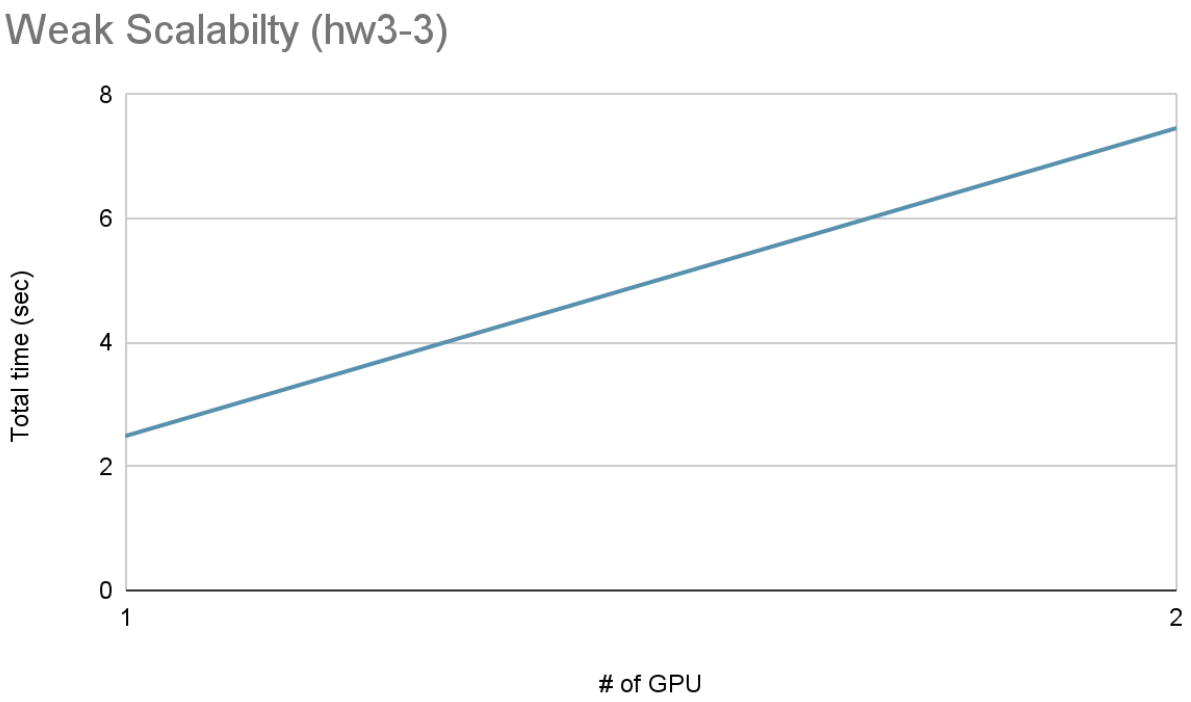




Conclusion:

Using shared memory instead of always loading data from global memory is a very important optimization in this assignment. Next, using padding instead of always checking the boundary is also important. For streaming, I did not implement it in the final version of hw3-2, because, I think the time spent waiting for the stream to synchronize is taking too long. Then, a large blocking factor also affects the performance a lot, as when the blocking factor is large, the occupancy also increases.

d. Weak Scalability (hw3-3)
Fixed problem size for each GPU with problem size 11000 vertices per GPU.



Below is the profiling for the problem size of 11000 vertices for each GPU.
Single GPU:

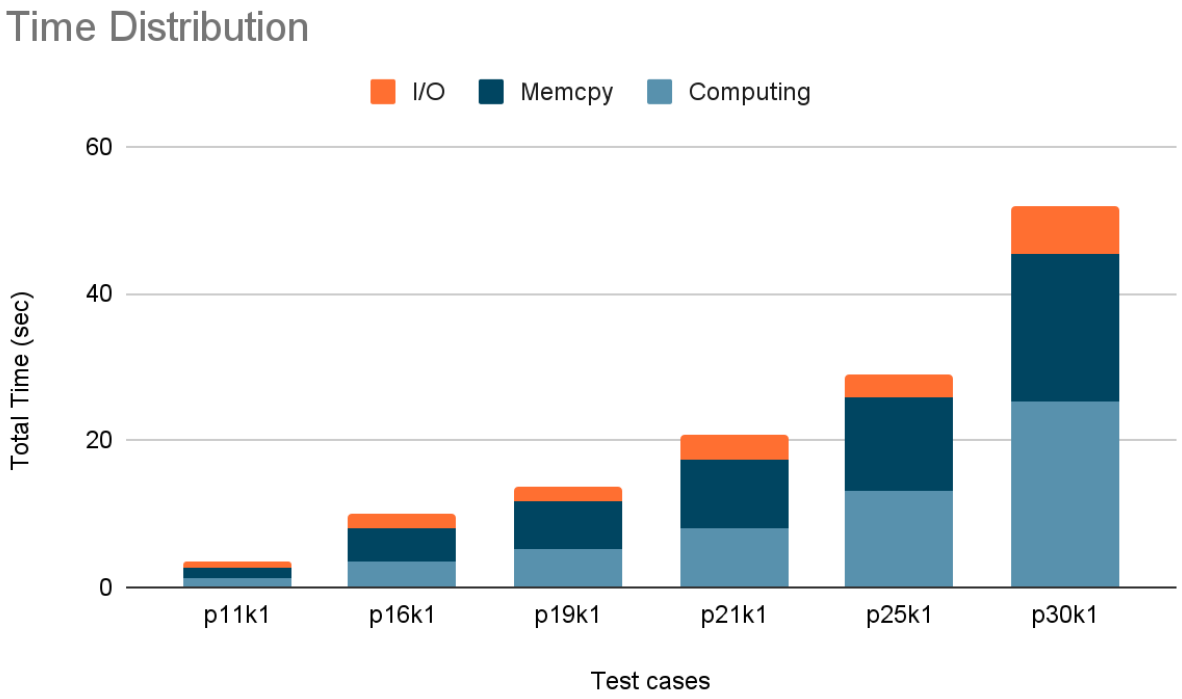
Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	85.79%	1.15240s	172	6.7000ms	6.5265ms	6.8914ms	phase3(int*, int, int)
	5.57%	74.879ms	1	74.879ms	74.879ms	74.879ms	[CUDA memcpy HtoD]
	5.50%	73.932ms	1	73.932ms	73.932ms	73.932ms	[CUDA memcpy DtoH]
	2.81%	37.773ms	172	219.61us	211.46us	230.63us	phase2(int*, int, int)
	0.32%	4.2545ms	172	24.735us	23.840us	25.728us	phase1(int*, int, int)
API calls:	84.77%	1.34206s	1	1.34206s	1.34206s	1.34206s	cudaMemcpy
	15.06%	238.47ms	1	238.47ms	238.47ms	238.47ms	cudaHostAlloc
	0.08%	1.2755ms	516	2.4710us	2.3060us	22.393us	cudaLaunchKernel
	0.03%	517.85us	2	258.93us	5.7390us	512.11us	cudaFree
	0.03%	438.21us	1	438.21us	438.21us	438.21us	cudaMalloc
	0.01%	171.06us	101	1.6930us	105ns	79.258us	cuDeviceGetAttribute
	0.01%	156.69us	1	156.69us	156.69us	156.69us	cuDeviceTotalMem
	0.00%	29.533us	1	29.533us	29.533us	29.533us	cudaMemcpyAsync
	0.00%	20.598us	1	20.598us	20.598us	20.598us	cuDeviceGetName
	0.00%	4.4860us	1	4.4860us	4.4860us	4.4860us	cuDeviceGetPCIBusId
	0.00%	771ns	3	257ns	169ns	423ns	cuDeviceGetCount
	0.00%	617ns	2	308ns	156ns	461ns	cuDeviceGet
	0.00%	267ns	1	267ns	267ns	267ns	cuDeviceGetUuid

2 GPUs:

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	88.38%	9.03565s	688	13.133ms	12.917ms	13.482ms	phase3(int*, int, int, int)
	2.96%	302.88ms	2	151.44ms	150.97ms	151.91ms	[CUDA memcpy HtoD]
	2.89%	295.82ms	2	147.91ms	147.90ms	147.92ms	[CUDA memcpy DtoH]
	2.82%	288.12ms	344	837.57us	836.23us	870.67us	[CUDA memcpy PtoP]
	2.78%	284.27ms	688	413.18us	401.57us	429.35us	phase2(int*, int, int)
	0.17%	16.897ms	688	24.560us	23.105us	33.825us	phase1(int*, int, int)
API calls:	83.23%	9.15667s	346	26.464ms	5.3520us	4.27454s	cudaMemcpy
	8.32%	915.31ms	2064	443.46us	2.3270us	142.32ms	cudaLaunchKernel
	4.66%	512.55ms	1	512.55ms	512.55ms	512.55ms	cudaHostAlloc
	1.84%	202.93ms	2	101.46ms	4.2610ms	198.67ms	cudaDeviceEnablePeerAccess
	1.84%	202.04ms	2	101.02ms	1.5768ms	200.46ms	cudaMalloc
	0.10%	11.387ms	3	3.7958ms	5.5480us	9.5601ms	cudaFree
	0.00%	486.29us	2	243.14us	240.99us	245.29us	cuDeviceTotalMem
	0.00%	415.96us	202	2.0590us	167ns	91.208us	cuDeviceGetAttribute
	0.00%	45.861us	2	22.930us	11.528us	34.333us	cudaMemcpyAsync
	0.00%	45.237us	2	22.618us	18.089us	27.148us	cuDeviceGetName
	0.00%	11.509us	2	5.7540us	2.1120us	9.3970us	cuDeviceGetPCIBusId
	0.00%	7.0310us	2	3.5150us	2.5330us	4.4980us	cudaSetDevice
	0.00%	1.4000us	3	466ns	261ns	870ns	cuDeviceGetCount
	0.00%	1.2970us	4	324ns	172ns	693ns	cuDeviceGet
	0.00%	644ns	2	322ns	285ns	359ns	cuDeviceGetUuid

Conclusion:
The scalability is not that good, as seen from the nvprof, the time for cudaMemcpy took around 9 seconds, it increases compared to the single GPU version (only 1 call with 1 second), so most of the time is spent on the communication time.

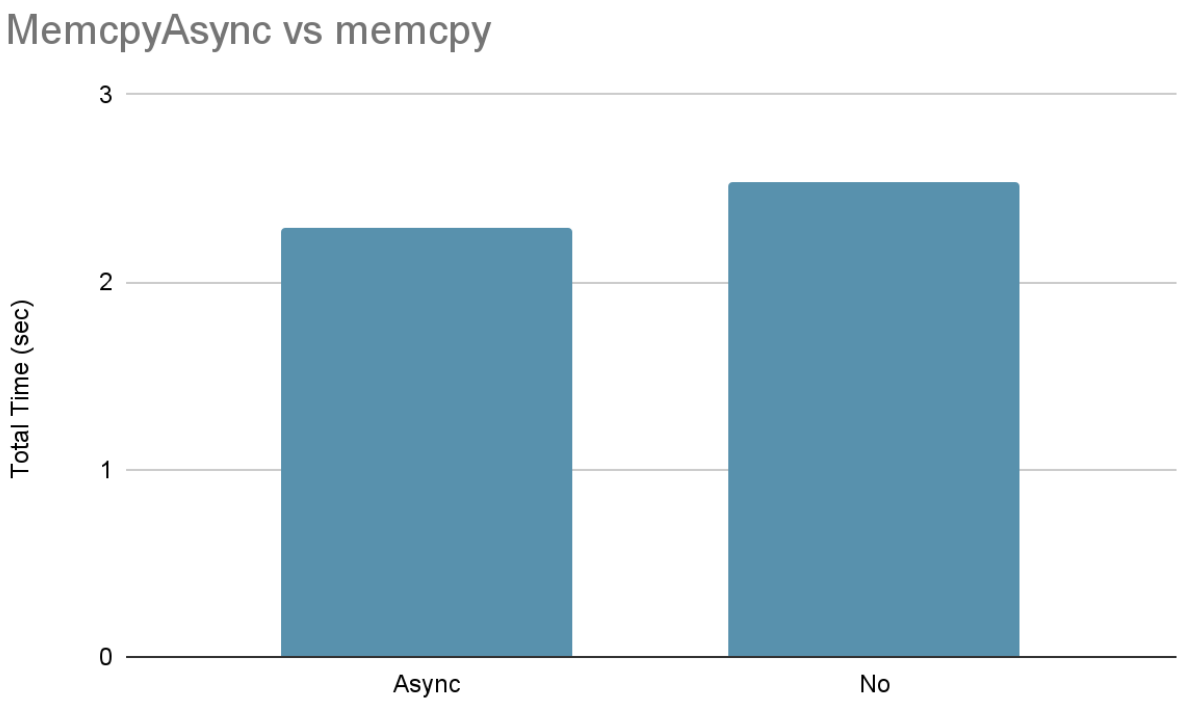
e. Time Distribution (hw3-2)
Computing:
CUDA computation time (phase 1, phase 2, phase 3) (from nvprof)
I/O time: time to read and write the file.
Memcpy: CUDA memcpy's time includes cuda host-to-device memcpy, cuda device-to-host time memcpy

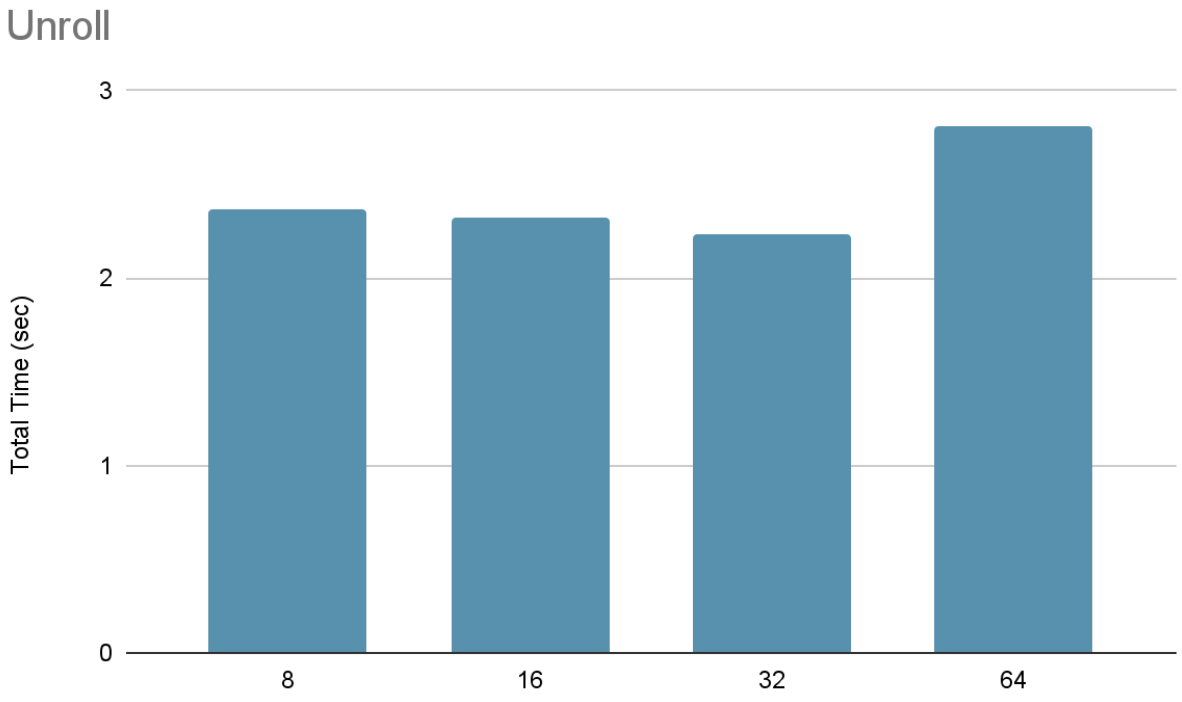


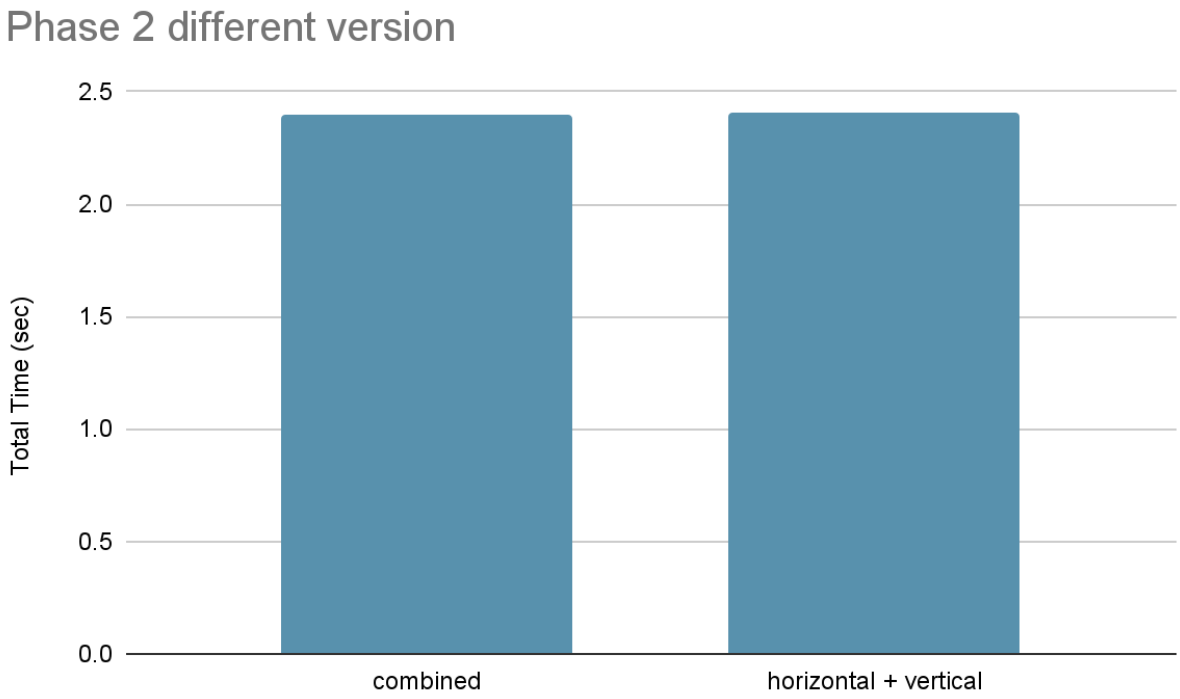
Conclusion:

Based on the time distribution, most of the time is spent on memory copy and computing time. I think this is reasonable because if the number of data to compute increases, then the number of data needed to be copied into the device and back to the host also increases. Additionally, I use cuda memcpy async when doing memcpy from host to device, so the time is of the cudaMemcpy only includes the memcpy from device to host.

f. Others
Unroll







Conclusion:

For unroll, I chose to unroll 32, because as seen from the plot, it has the best result, then I also use memcpyAsync, for the memory copy from host to device, because then I can overlap some CPU operation while waiting for the memcpy to finish. Lastly, I also do an additional experiments for testing 2 different versions of phase 2. The combined version means that I only use 1 kernel to do both pivot row and column so that the pivot block will only be copied once. While for the second version, I do the pivot row and column with the different kernels, so the pivot block is copied twice and from the figure result, I used the combined version in the final hw3-2.