# Part 1:

| # of rows | average execution time for col_wise (micro seconds) | average execution time for row_wise (micro seconds) |
|---|---|---|
| 10 | 1.9 | 1.6 |
| 100 | 83.2 | 98.2 |
| 1000 | 7343.6 | 9512.2 |
| 10000 | 1078361.9 | 969740.4 |
| 50000 | 39511692.5 | 23956931.4 |

## Report Questions:

1. Spatial locality is when elements are located in close proximity to one another. It is especially common in arrays, which use a specific form of spatial locality called sequential locality, which is when elements are stored sequentially/linearly in memory. Spatial locality allows elements to be accessed quickly because they are easier to find in memory and allows for elements to be cached easilier because when a miss occurs and a word is pulled into the cache, the rest of the elements will be pulled as well.

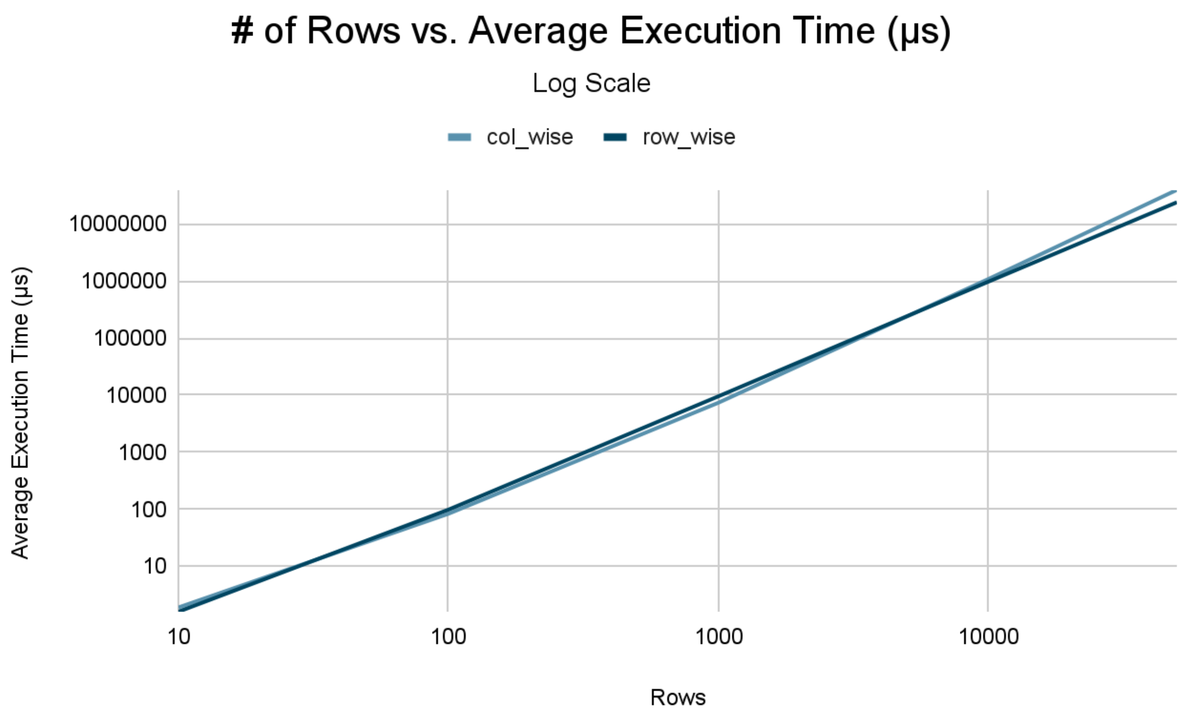
2. Based on my knowledge of spatial locality, the row_wise.c program should perform faster since the inner for loop accesses each element in the specific array of the 2D array (ex. A[0][0], A[0][1], …) then switches to the next array stored in the 2D array (ex. A[0][j], A[1][j], …,) so each consecutive element that is accessed is stored consecutively in memory. This is unlike the col_wise.c program, which switches the order of the for loops- forcing the program to bounce around the memory in order to access the elements of each array stored in the 2D array.

3. As the row size increases, both the row_wise.c and col_wise.c programs increase in execution time (as expected.) At first, the row_wise program was a little bit slower than the col_wise program, which was unexpected. But, when the row size becomes really high, the col_wise program's average execution time increases faster than that of the row_wise program, and the spatial locality optimization in the row_wise program can be realized.

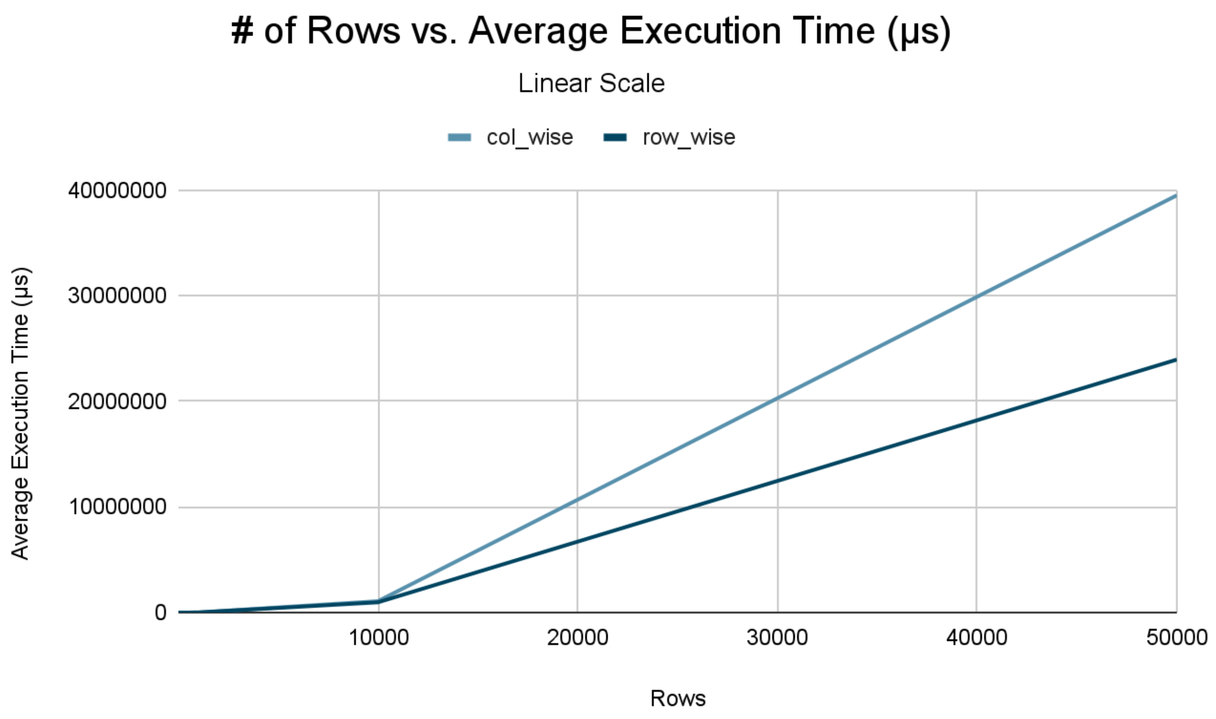
4. The answer I gave for question 3 is the same as the answer that I gave for question 2 for when the row size is high. When the row size was low, the unoptimized code performed better than the spatially optimized code, which was unexpected.

5. I think that the program behaves the way it does because at lower row amounts, the compiler is able to optimize how the arrays contained in the 2D array are stored in memory, which allowed col_wise.c to perform better than row_wise.c until this compiler optimization was no longer possible.

6. I think we executed each program 10 times to rule out the possibility of a fluke execution time, which could've ended up skewing the results. Taking the average for all of the tests allowed the experiment to remain impartial because outliers would affect the data less.



# of Rows vs. Average Execution Time (µs)
Log Scale
col_wise
row_wise
Average Execution Time (µs)
10000000
1000000
100000
10000
1000
100
10
10
100
1000
10000
Rows



# of Rows vs. Average Execution Time (µs)
Linear Scale
col_wise
row_wise
Average Execution Time (µs)
40000000
30000000
20000000
10000000
0
10000
20000
30000
40000
50000
Rows

## Part 2:

1. On silo, the L1d cache (which is for data) is 768 KB, the L1i cache (which is for instructions) is also 768 KB, the L2 cache is 12 MB, and the L3 cache is 192 MB.

2. The total amount of RAM on the silo machine is 515388 Mebibytes. 1 Mebibyte is $2^{50}$ bytes, while a Megabyte is $10^6$ bytes.

3. RAM/Cache size has an effect on execution time because RAM and cache are both very fast, although cache is much faster than RAM. Because of this, it is more optimal to use these to access data faster than accessing the same data stored on a disk. So, the more space that is available to store data on faster storage mediums such as RAM and cache, the more information will be able to be accessed using these faster mediums. This can affect execution time because if there is a lot of data that needs to be accessed really quickly and there isn't enough space in fast modes of storage like RAM and cache, then it could force the program to resort to reading from slower forms of storage.

# Part 3:

| # of rows | average execution time for col_wise (micro seconds) | | | | | | average execution time for row_wise (micro seconds) | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | cache hits | | | cache misses | | | cache hits | | | cache misses | | |
| Level of cache | L1d & L1i | L2 | L3 | L1d & L1i | L2 | L3 | L1d & L1i | L2 | L3 | L1d & L1i | L2 | L3 |
| 10 | 104365 | x | x | 2473 | x | 4401* | 105884 | x | x | 2766 | x | 4115* |
| 100 | 298220 | x | x | 3327 | x | 4191* | 294515 | x | x | 3383 | x | 4181* |
| 1000 | 18559102 | 189518 | 1610448 | 126516 | 62333 | 35812 | 19133352 | 8609 | 225048 | 161637 | 5913 | 5883 |
| 10000 | 1981533701 | 47315693 | 281608302 | 116259850 | 116974905 | 116617598 | 1896233264 | 51171 | 23149321 | 12429330 | 126198 | 134254 |
| 50000 | 49834701036 | 2093504257 | 2652920560 | 2869879360 | 5231563120 | 5229503936 | 47515767753 | 1235555 | 568220754 | 312652771 | 3087589 | 3166156 |

## Report question:

1. perf stat -e L1-dcache-load-misses, L1-dcache-loads, L1-icache-load-misses, L1-icache-loads,LLC-loads, LLC-load-misses, LLC-stores,cache-misses, cache-references, l2_cache_req_stat.ic_dc_miss_in_l2 ./cw

   For tests with a low amount of rows, some of the tests would output "not counted" or "not supported." This is why there are no L2 cache hits and misses and L3 cache hits results for the 10 and 100 row tests for the on both col_wise.c and row_wise.c. This could've been the fault of the "NMI watchdog," which perf was complaining about for these tests- although I do not have the necessary permissions to change this. I used the cache-misses event value for the L3 cache misses because I read that this can be used as a substitute for LLC cache (which on silo would be L3.) For tests where the value was present, I used the cache-references event value and subtracted the amount of cache-misses from it for the L3 cache. For the tests that show the L2 cache misses, I used this value. For the L2 cache hits, I divided the L2 cache misses by the percentage of total L2 cache loads, and subtracted the this value by the L2 cache misses. For the L1d and L1i caches, I added up the miss rates for both of these for the L1 miss values, and I added up the L1 cache loads and subtracted the total L1 cache misses from it.

2. The cache hits values signify the amount of times the programs looked for a piece of data in one of the 3 levels of cache and was able to find it there. The cache misses values signify the amount of times the programs looked for a piece of data in one of the 3 levels of cache and was unable to find it. Upon a miss, the program would then load this piece of information into the cache to access it more easily next time it needs it.

3. As the spacial locality of a program improves, there will be less cache misses in all levels and more cache hits in all levels. This is due to the fact that if a data structure stores its elements in close proximity (such as an array storing its elements sequentially,) once a word is loaded into one of the levels of cache it will contain many of the next elements in the array. This is why the row_wise.c program was able to have less cache misses, more cache hits, and an overall lower execution time when the rows get higher- since the for loops access information in one of the individual arrays stored in the 2D array and then move to the next array, the information in each array stored in the 2D array are loaded into cache so the program is able to access this information more efficiently in the cache than in col_wise.c.