Part 1.

1.

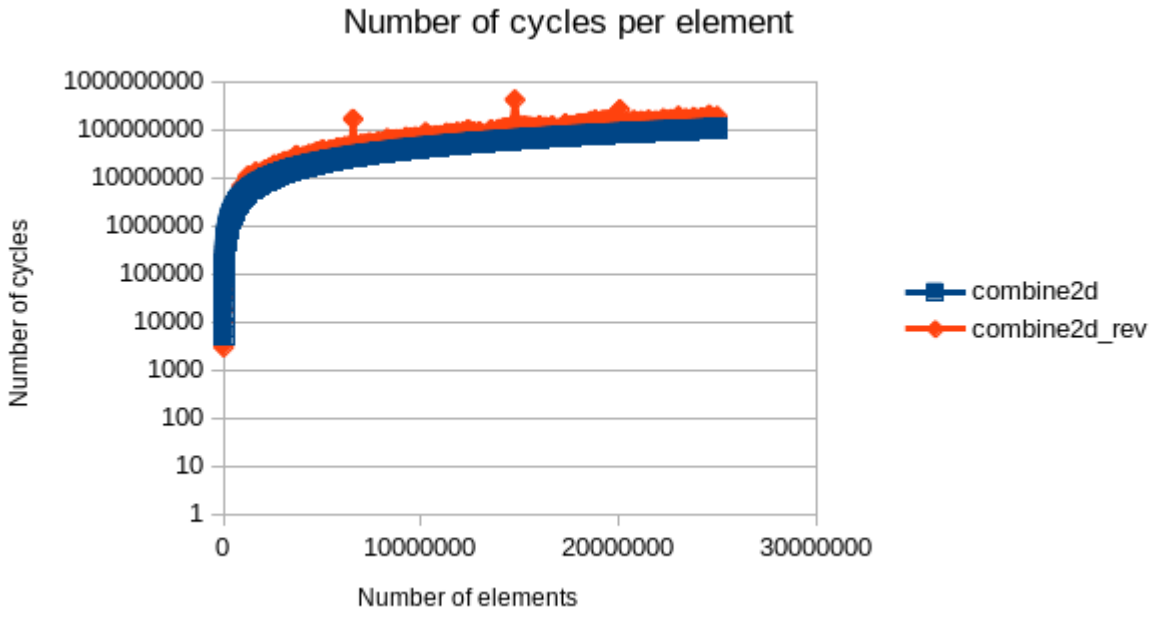
- In my case, the program stopped execution when the array size reached 47000 (BASE=0, ITERS=4700, DELTA=10). I received the output as:

  Hello World -- 2D Combine

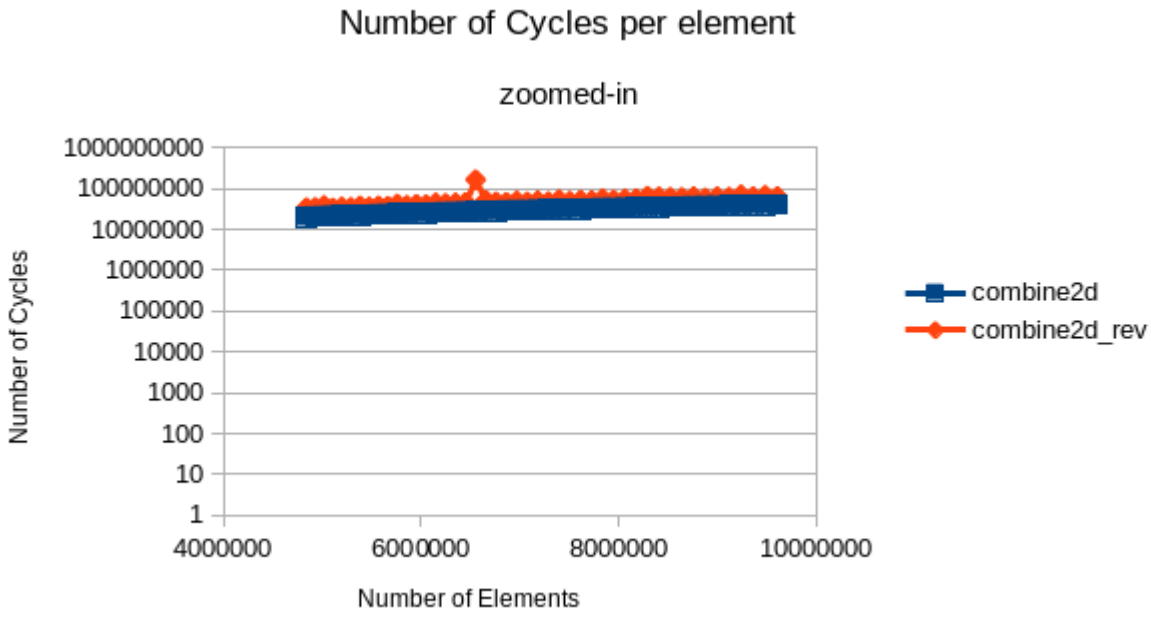
  COULDN'T ALLOCATE STORAGE
  Segmentation fault

- By running multiple tests on both the functions over a range of array sizes, I observed that the function combine2d is faster than combine2d_rev by roughly 1.8 times. Some of the results I obtained from my tests were as follows:

  (Operation: Multiplication,BASE=0, ITERS=250, DELTA=20)



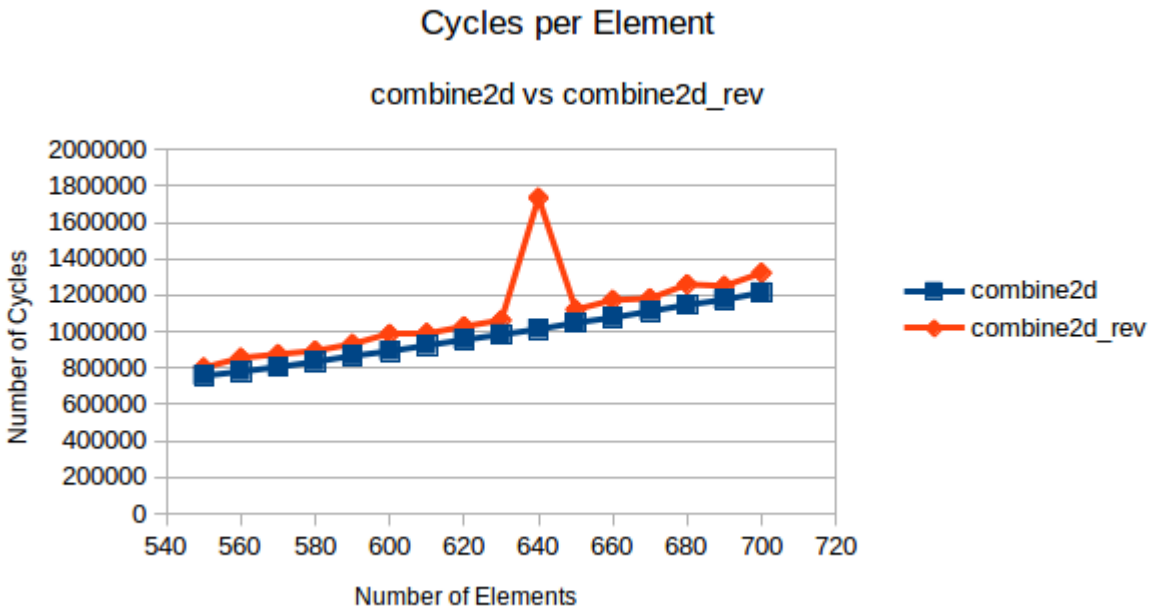
As we can see, the combine2d function has fairly constant CPE over the whole range on which it was tests. However, combine2d_rev shows strange spikes in between and has a larger CPE as well.



On zooming in over a small range, we can clearly see a significant increase in number of cycles at a particular point.
The reason behind this observation is that combine2d is row major multiplication whereas combine2d_rev is a column major multiplication. As a result, till the point where the cache size is larger than the array size, both the functions perform similarly. However, as soon as the array size becomes larger than the cache size, combine2d_rev takes a hit. This is because the elements fetched from column that are loaded in the cache along with the neighboring elements (principal of spatial locality) get evicted as soon as the cache is full and next element in the column is loaded in the cache. As a result, we get a series of misses and the data needs to be fetched from the memory. Hence, the number of cycles increase significantly.
In row-major, the elements are stored row wise and hence we get a lot of hits in the cache over the complete range and hence the CPE is fairly constant over the complete range.

Speed-up of combine2d over combine2d_rev = 1.802

- At the array size of nearly 640, the strange things begin to happen in combine2d_rev. This is because the array size becomes greater than the cache size and hence we encounter successive misses. The data has to be accessed from memory which takes a large number of cycles. Hence, we observe such spikes in the graph.



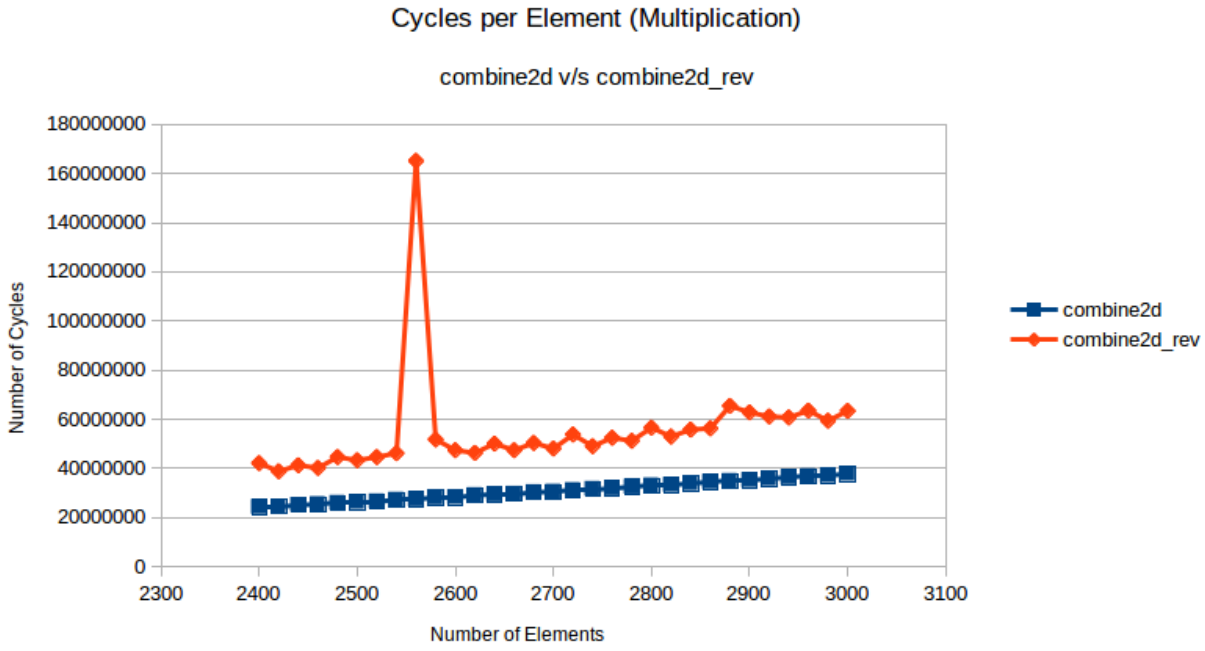
2.

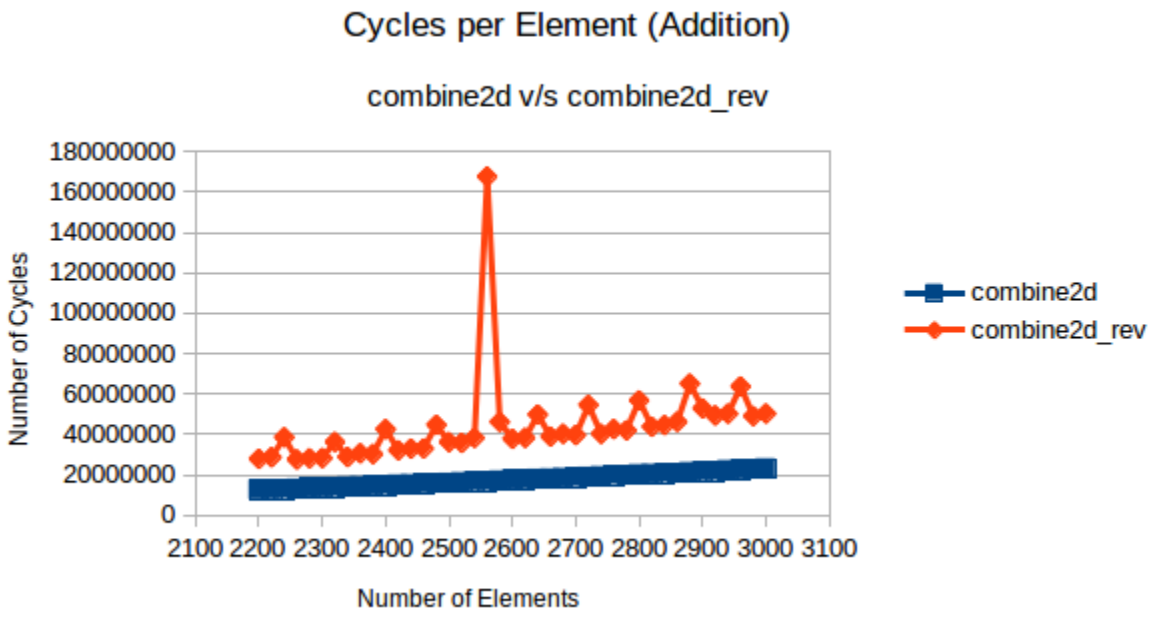
- For small range:
  CPE of combine2d:
  CPE of combine2d_rev: 38009.337
- For large range:
  CPE of combine2d: 13184.017
  CPE of combine2d_rev: 36328.770

As we know, multiplication is way slower than addition, hence for combine2d, which is a row-major function, CPE of addition is significantly smaller than CPE of multiplication.

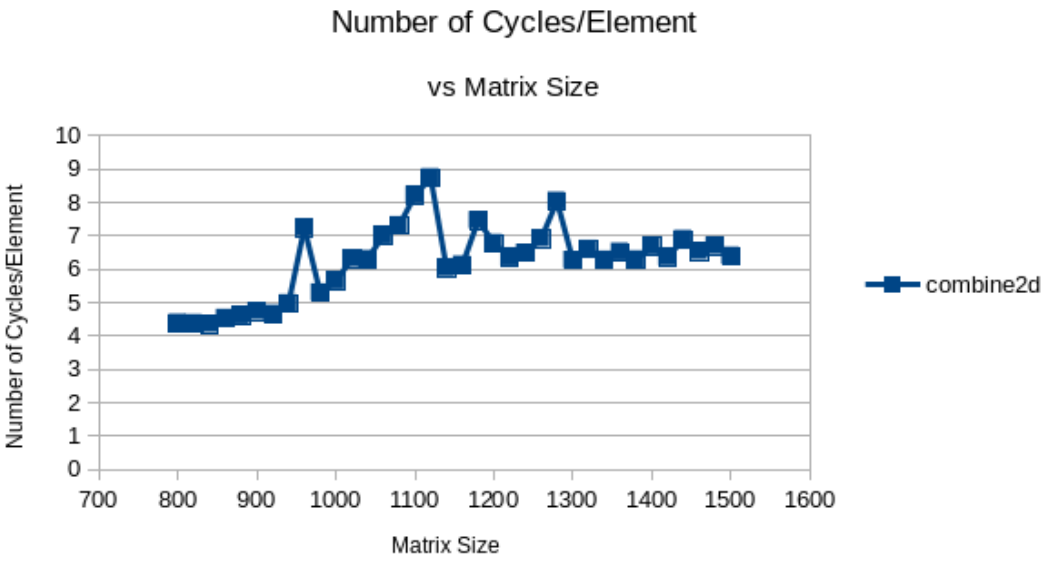
However, for combine2d_rev, which is a column major function, CPE of both addition and multiplication is almost identical (addition being slightly smaller since addition takes less cycle than multiplication). This is due to the fact that the number of misses are a lot in column-major operations when the cache size is smaller than the array size.

The zoomed-in graphs for narrow ranges in case of multiplication and addition are attached. The same were used to make the aforementioned observation.

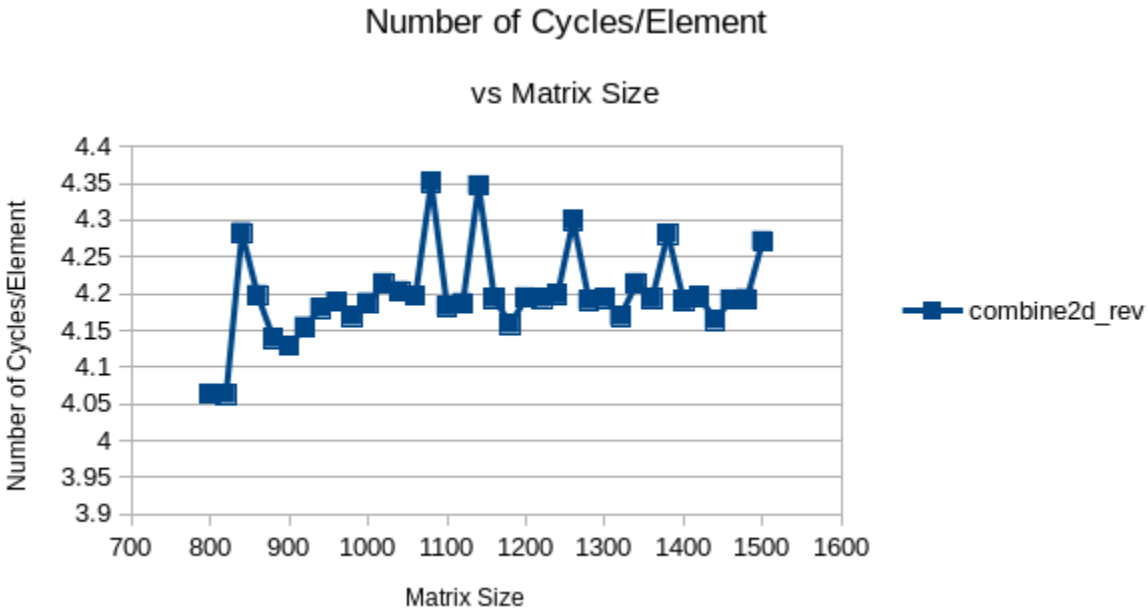

Cycles per Element (Multiplication)
combine2d v/s combine2d_rev
Number of Cycles
180000000
160000000
140000000
120000000
100000000
80000000
60000000
40000000
20000000
0
2300
2400
2500
2600
2700
2800
2900
3000
3100
Number of Elements
combine2d
combine2d_rev


Cycles per Element (Addition)
combine2d v/s combine2d_rev
Number of Cycles
180000000
160000000
140000000
120000000
100000000
80000000
60000000
40000000
20000000
0
2100 2200 2300 2400 2500 2600 2700 2800 2900 3000 3100
Number of Elements
combine2d
combine2d_rev

3.



In case of combine 2d, the elements are stored in a row major format and hence the performance improves for a while till the time the elements fit in the cache block. However, as soon we move to the next row, we empty the cache and the next row is fetched from the memory. Hence after a while the performance gets worse.

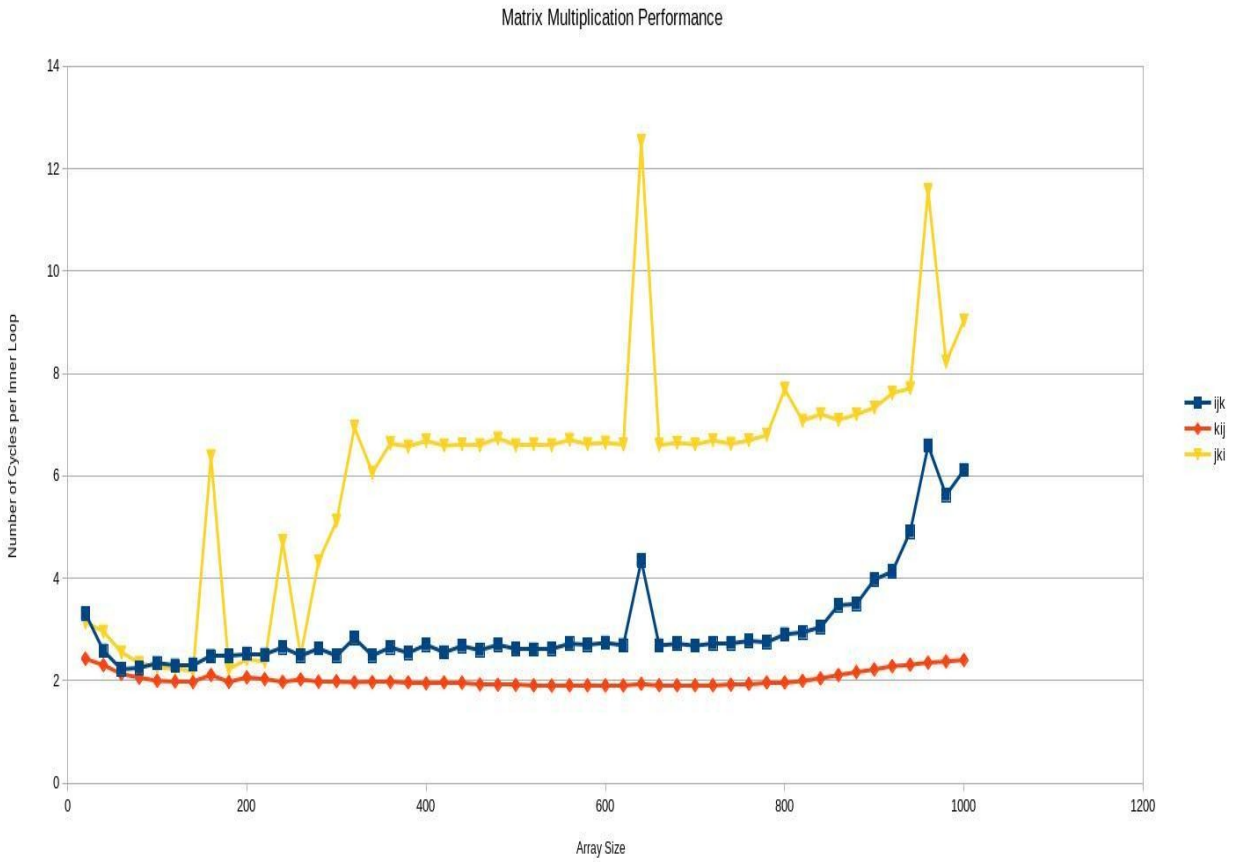


In case of combine2d_rev however, the behaviour more jumpy once the matrix size is greater than the cache size since we encounter successive misses when going from one

element of the column to the next as well as when changing columns. Hence the behaviour is more jumpy.

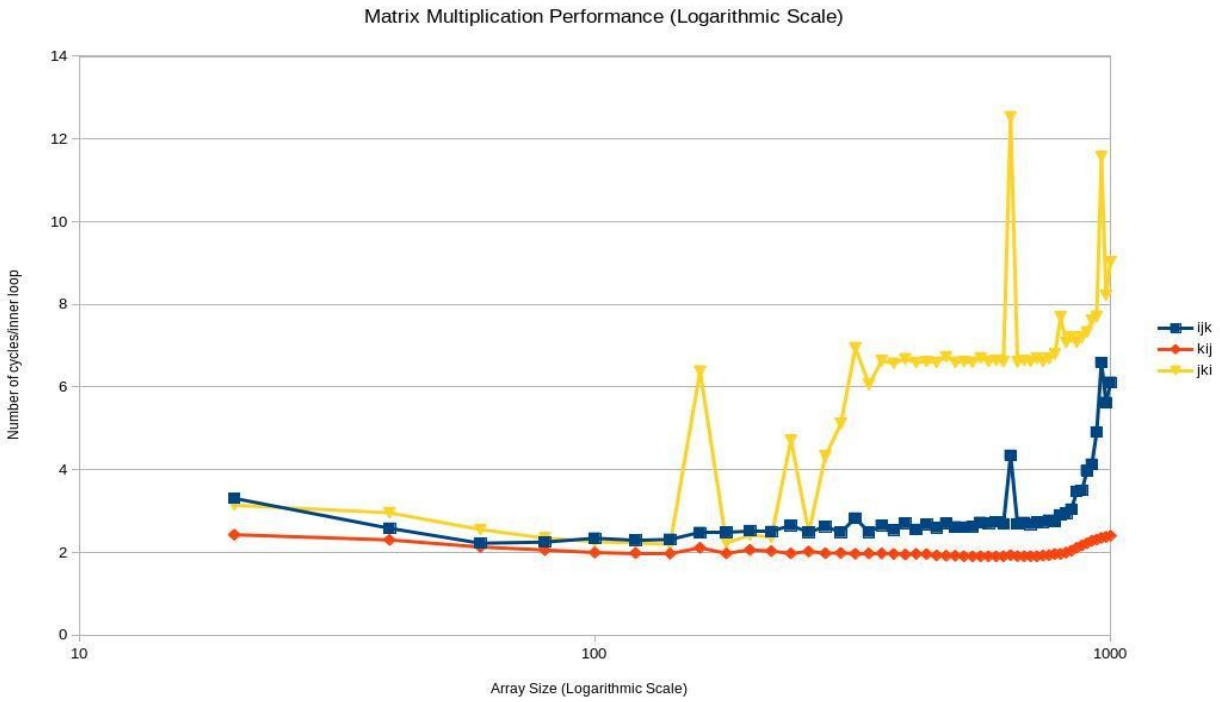
Part 2.

I ran the code for various sizes of array and tried to visualize the results. The result for matrix performance as a function of array size is given below:

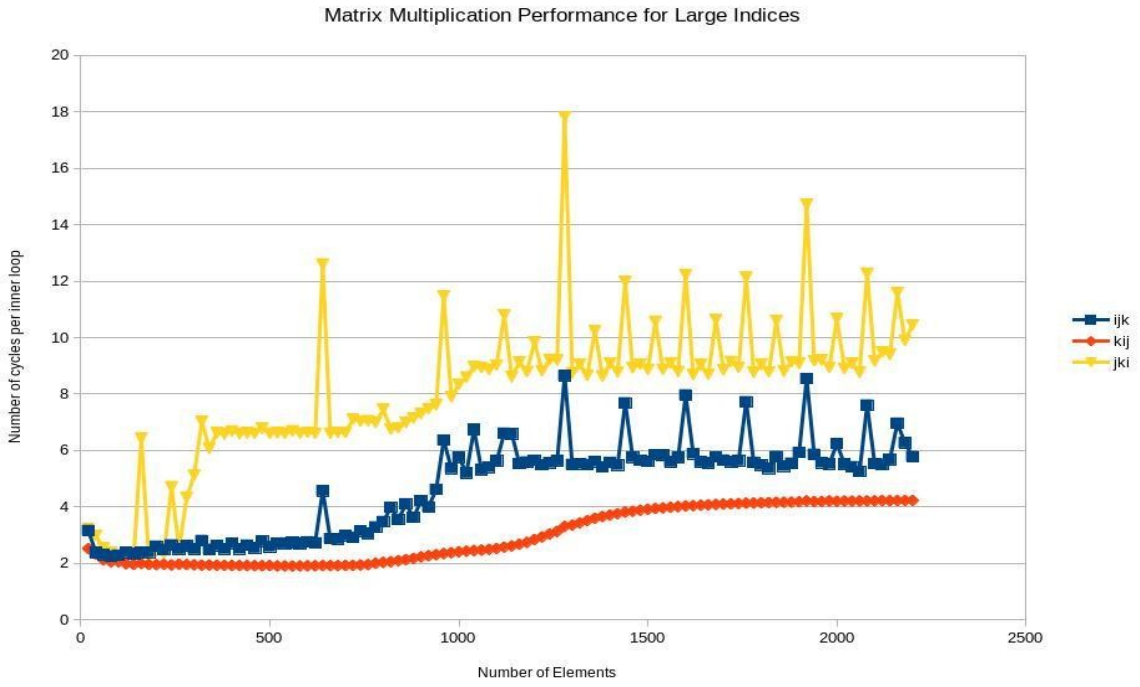


The above graph is without logarithmic scale

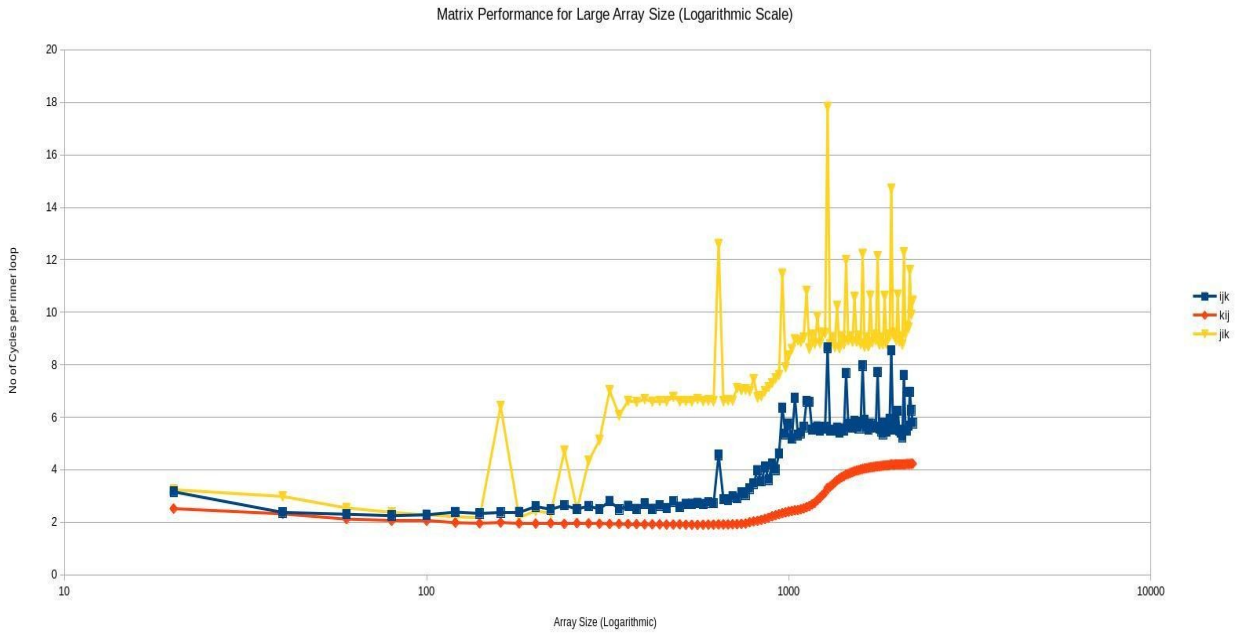
The graph shows the performance in terms of “No of Cycles per inner loop” as a function of “Array Size”. Here it can be seen that “kij” looping pattern gives the best performance and “jki” looping pattern gives the worst. This is in coherence with the book and the plot looks highly similar to the graph given in the book. Some of the peaks are due to the noise in the data and can be ignored. The logarithmic scale graph is attached below:



I ran the tests for very large array size too (above 2k). The graph for that is attached below:

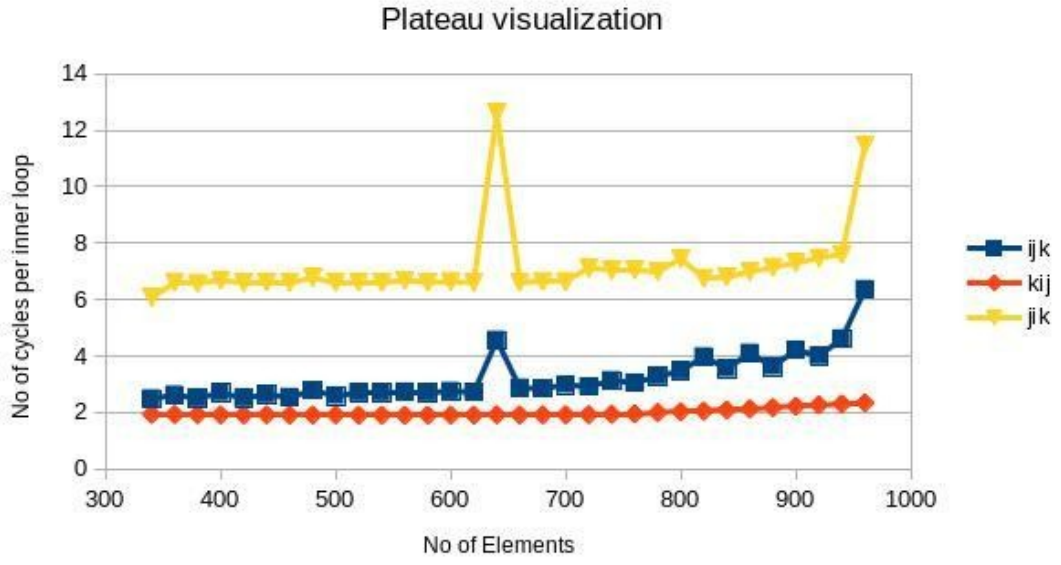


Logarithmic Scale:



As seen from the above graphs we can say that there are at least one plateaus for each looping pattern. The spikes shown in the graphs are due to the cache misses that happen most in the case of “jik”. As we studied that “kij” looping pattern utilizes the cache most efficiently, therefore it provided the lowest miss rate and hence the plot for it is fairly smooth as compared to others.

We have used above 2k array size to visualize the plateaus as they are mostly visible in that range only when the no cycles are significantly large for each looping pattern. The plateau



region can be seen below:
The no of cycles per iteration for each looping pattern here is:

- ijk: 200000000 to 4000000000

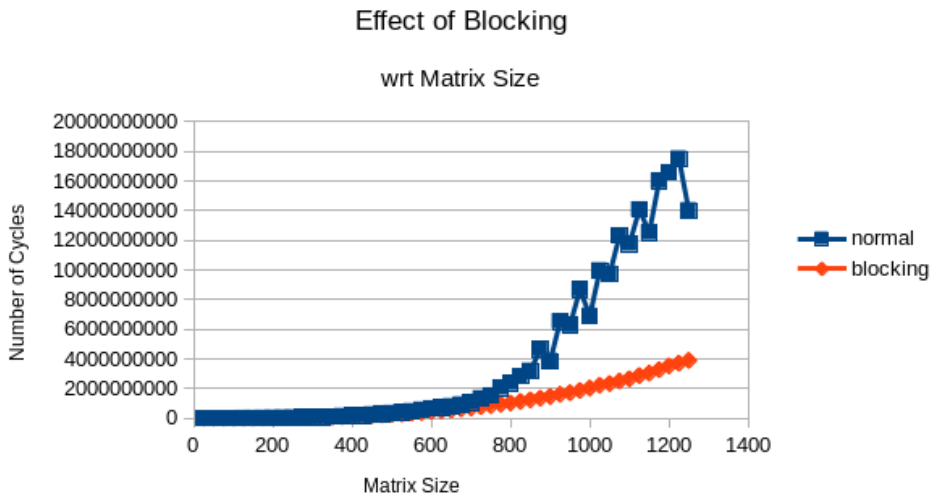
-
- kji: 200000000 to 1000000000
-
- jik: 200000000 to 6000000000

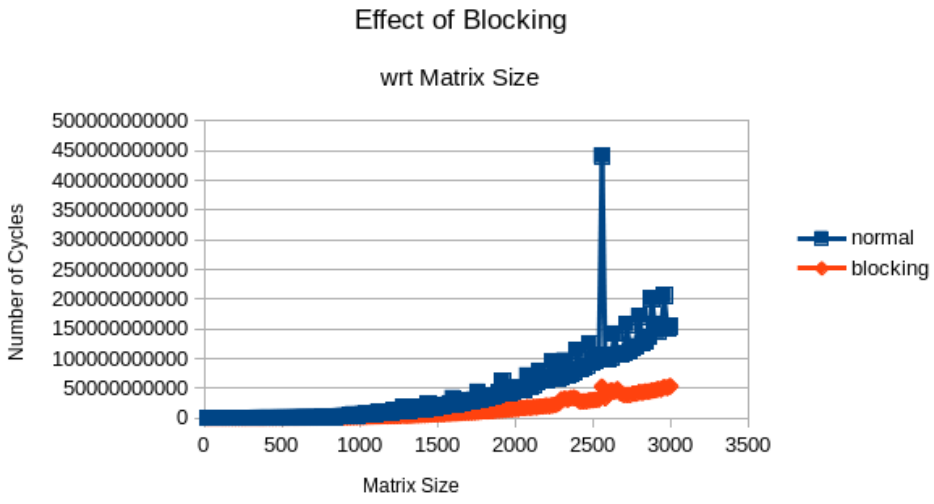
So from the above plots we can conclude that the no cycles taken by “kji” looping pattern is the least where as no cycles taken by “jik” looping pattern is the greatest. The transitions are occurring when the size of array is much larger than the cache size and the looping pattern is producing significant number of cache misses.

As the cache misses are the least for the case of “kji” looping pattern, it has a smooth transition for most of the array sizes and the transitions are the most for “jik” looping pattern.

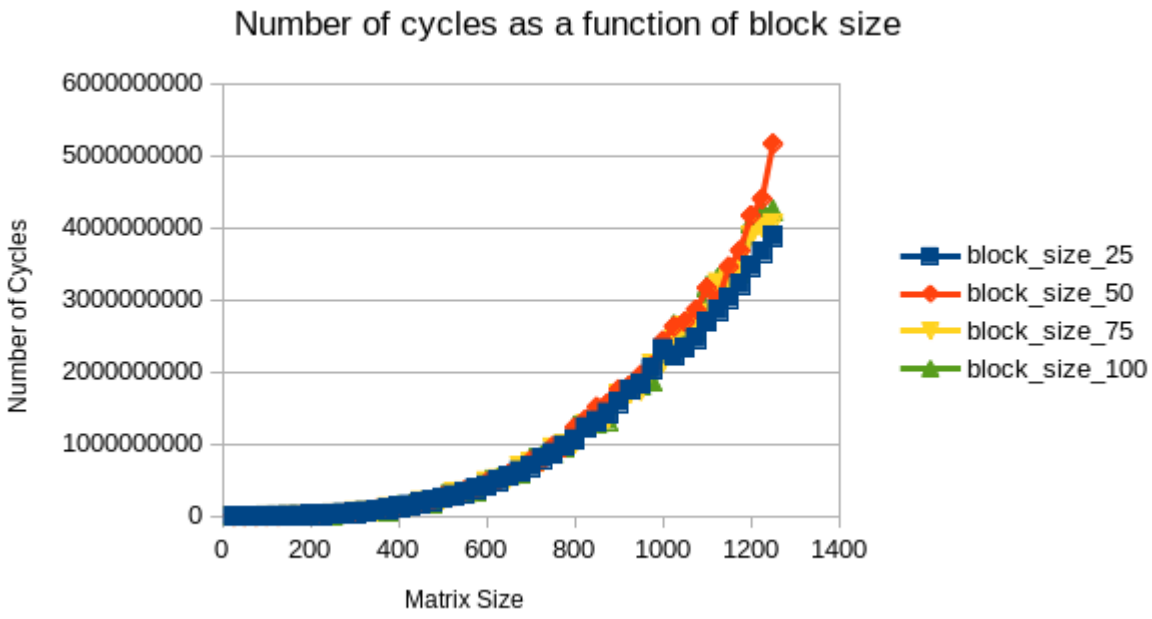
Part 3.

1. I observed that if the matrix size increases, the effect is blocking is still significantly better than the non-blocking matrix. However, for small matrix sizes, we don't observe a lot of misses in case of blocking but as the size keeps increasing, we tend to get misses in the blocking matrix multiplication as well.
   The large spike we see in the second graph is due to the fact that the block size is 25, DELTA is 20 and the array size is 3000, so it the blocks don't fit perfectly in the whole array. I did this to test this effect since for the first graph, the block size, number of iterations and the DELTA are all in multiples of 25 and hence show a perfect behavior.





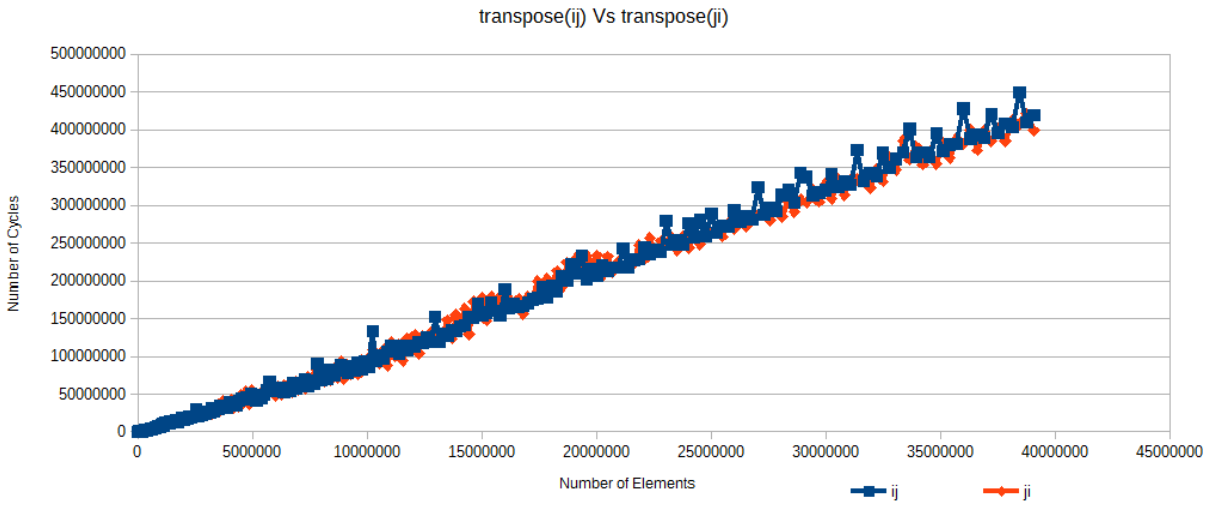
2. For the various block sizes of 25,50,75 and 100, keeping BASE=0, ITERS = 50, DELTA = 25, I get the following result:



As we can see from the graph obtained, the optimal block size in my case is 25 since the number of cycles taken to multiply the array of fixed size 1250 is the least for block size = 25.

Part 4:

1. Implemented test_transpose.c as described.
2. ij and ji loop permutations for a wide range of matrix sizes---



BASE = 0
ITERS = 250
DELTA = 25
As per the graph when the array size becomes greater than the cache size, the number of miss increases thus spatial locality and temporal locality does not help improve the performance.

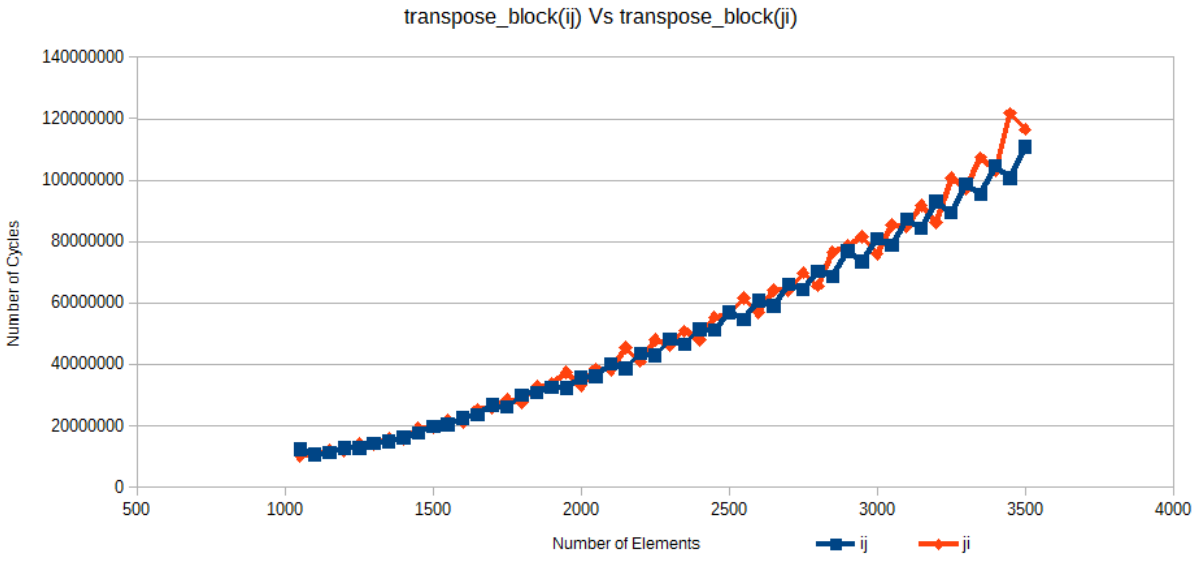
1. The new code for blocking is under the name test_transpose_block.c

```
for (i = 0; i < length; i += blocksize) {
for (j = 0; j < length; j += blocksize) {
// transpose the block beginning at [i,j]
for (k = i; k < i + blocksize; ++k) {
for (l = j; l < j + blocksize; ++l) {
dest[l + k*length] = data[k + l*length];
```

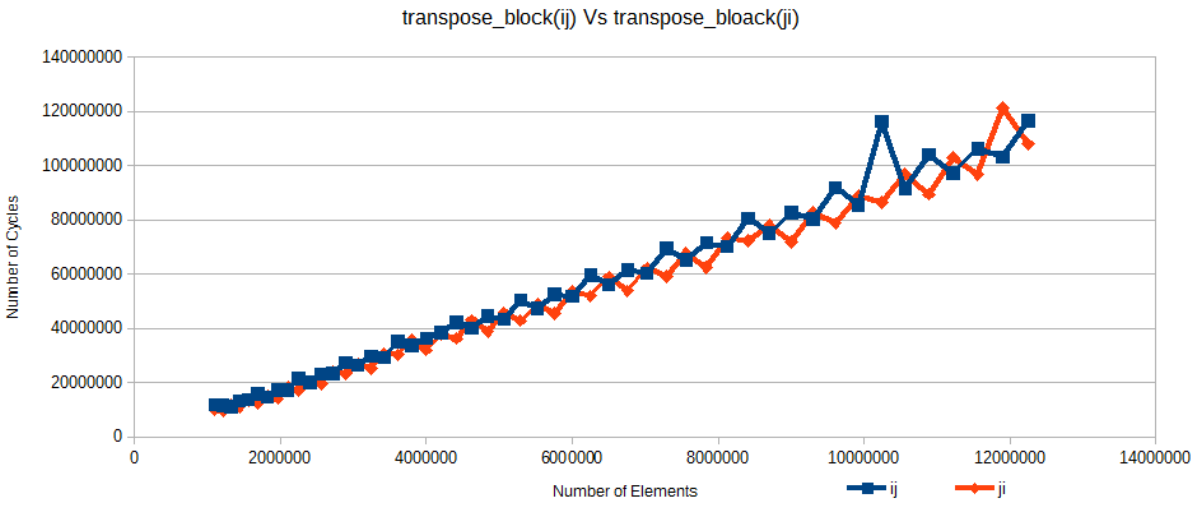
Blocking is done using four loops. Two loops are used to traverse through the blocks and other two loops are used to traverse through the array. Blocking should help because the block gets loaded into the cache which does not increase than the cache size. Blocking does not help in improving the performance for both ij and ji.

Different block sizes---

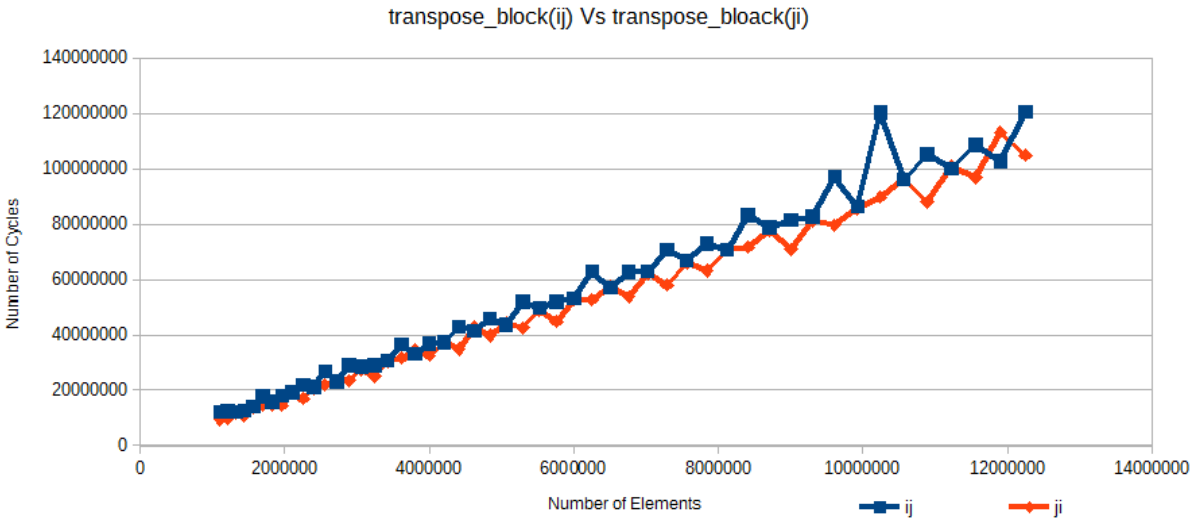
Block = 25



Block = 50



Block = 75



Part 5:

1. 4 days
2. No.
3. No.
4. No.