COL216 INTELLIGENT CODING

Jahnabi Roy
2022CS11094

24th April, 2024

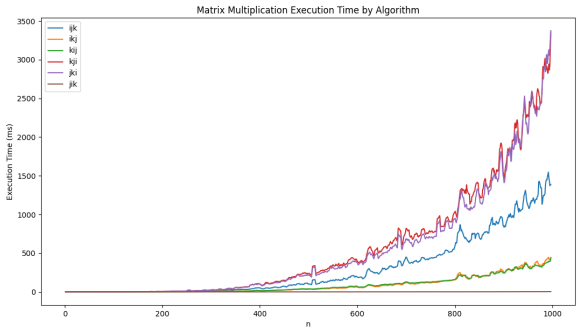
1 Introduction

This assignment focusses on finding the best coding practice for a particular task to be accomplished.

1.0.1 Matrix Multiplication

The code snippet runs in time complexity of O(n^3) since an additional matrix is used for storing the multiplication result. The combinations used were the following -

- i-j-k order of indices
- i-k-j order of indices
- k-i-j order of indices
- k-j-i order of indices
- j-k-i order of indices
- j-i-k order of indices



Here as seen from the graph, the best performing order turns out to be the j-i-k order of indices. The execution time is in the order of 10ms for this order of indices. This is an interesting observation to note that on running all the 6

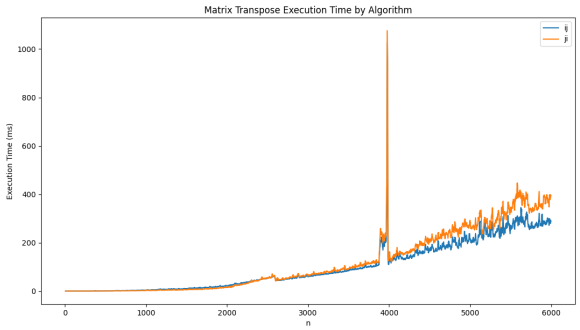
order combinations in a loop over n, the j-i-k turns out to be the least. But on running these for individual n values, these are the results obtained on running these individually -

n is : 256
Time taken for matrix multiplication (i-j-k order) with size 256: 8.75 ms
Time taken for matrix multiplication (i-k-j order) with size 256: 3.58 ms
Time taken for matrix multiplication (k-i-j order) with size 256: 2.72 ms
Time taken for matrix multiplication (k-j-i order) with size 256: 13.48 ms
Time taken for matrix multiplication (j-k-i order) with size 256: 13.49 ms
Time taken for matrix multiplication (j-i-k order) with size 256: 8.95 ms
n is : 2048
Time taken for matrix multiplication (i-j-k order) with size 2048: 21839.65 ms
Time taken for matrix multiplication (i-k-j order) with size 2048: 5388.47 ms
Time taken for matrix multiplication (k-i-j order) with size 2048: 6372.26 ms
Time taken for matrix multiplication (k-j-i order) with size 2048: 104370.66 ms
Time taken for matrix multiplication (j-k-i order) with size 2048: 105413.36 ms
Time taken for matrix multiplication (j-i-k order) with size 2048: 16695.02 ms

In the above cases, we observe that **k-i-j and i-k-j** order start beating the other loop orders. In the graph, we can also see (keeping aside the anomalous behavior of the j-i-k order) that k-i-j and i-k-j outperform the other strategies easily. This is also consistent with the principle of locality - spatial locality in general $\rightarrow C$ works in row major so iterating over k in the outer loop makes it easier accessing than in the innermost loop of the algorithm.

The spiking at regular intervals indicates the block size of the cache since this spike might be due to cache misses faced. Again spatial locality is being used here as a result leading to the spiking. As per observation, for every 250 values of n, approximately we get a spike. With this approximation, keeping in mind the 1D structure of the block in cache; we can estimate the cache size to be of the order of 10^2 with a factor obtained according to proper calculations.

1.0.2 Matrix Transpose - Additional Matrix

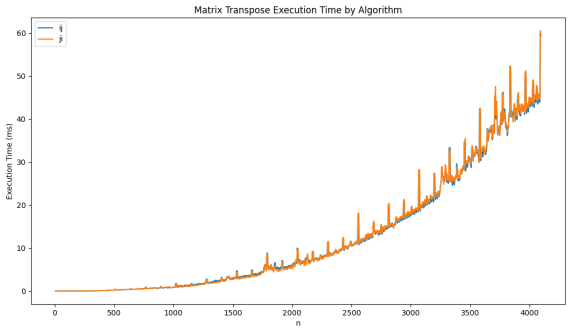


According to the graph, as n increases the i-j order surpasses the j-i order in performance as expected since C follows a row-major format. There are two interesting observations in this graph -

- There is a kink at nearly n = 2589.
- There is a sudden spike at around n - 4000.

Based on these observations, due to the kink in the graph, this could imply hitting a cache boundary leading to cache miss. A sudden spike could indicate hitting the cache capacity thus needing a high execution time. For this case, since an additional matrix is being used, a space complexity for the same would be around O(n^2). So the space in memory/ cache occupied would be of the order 2^{24} Bytes. So, this should be nearly 16MB. And considering the coefficients to play a role in the exact calculations, we can thus say that cache size is nearly 16MB which is realistically correct as well - 12 MB.

1.0.3 Matrix Transpose - In Place Matrix

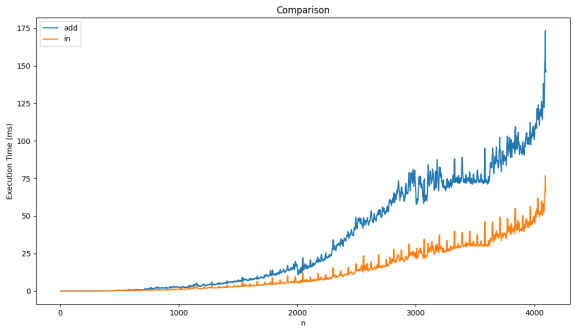


As expected, the i-j order is better performing than j-i for higher values of n. But in general it takes more time than using an additional matrix; the reason

being the additional swap function acting in every inner loop leading to a cumulative delay.

Here as observed as n increases the execution time also increases. But again as it nears the n = 4000 mark, it also starts plateauing and with a spike in n = 4000 it might start a slight downward trend as well. It is consistent with the calculations in the above section. Here, in place manipulation affected time performance but improves space performance slightly if not much.

1.0.4 Matrix Transpose - Compare Between In Place and Additional Matrix



This is the comparison of the two graphs as discussed above. The observations and inferences are same as listed above.