CS 231 - DLDCA Lab

Lab Assignment 3

Part 3

Report

Aditya Singh - 22B1844

Programming Task:

In this lab we had to perform matrix operations like, linear combination of matrices, Hadarmard product of two matrices and calculating the alternation summation of a matrix, i.e. $\sum_{i=0}^{n}\sum_{j=0}^{n}(-1)^{i+j}a_{ij}$.
We had to implement these operations in two different ways, once by traversing along the rows and also by traversing along the columns. We had to observe the difference in the number of clock cycles taken by the program in both these approaches.
Finally we had to write code for the allocation of memory for the matrices in the memory heap.

Approach:

In all the programs I used two registers r11 and r12 for indicating the matrix indices i and j respectively. In each one the code for the row and column traversal was almost the same except at a few places the value of i was replaced with j and vice versa.
For the Linear Combination we had to implement two for loops. To implement a nested for loop I created two functions, outer_loop and inner_loop. I carefully put the loop exit conditions and the i++ and j++ statements for the correct functioning of the loops. Inside the inner loop it was fairly simple to perform the linear combination by applying arithmetic operations on the matrix.
I tried to **optimise** the code by using the **minimum number of registers possible** and **avoiding unnecessary function calls.**
For the Hadarmard Product the code was almost similar to the above code only the linear combination was replaced with multiplication.
In the alternate summation, I implemented $(-1)^{i+j}$ using the register r15. First I gave it the value 1 because for i = 0, j = 0, $(-1)^{i+j} = 1$. Then **I multiplied r15 by -1 whenever i++ and j++ were performed** because value of $(-1)^{i+j}$ changes only when i or j change and an increase by one in them is equivalent to multiplying by -1. **This greatly simplified the process of calculating $(-1)^{i+j}$ and helped in reducing the number of clock cycles.**
In the beginning I had initialised r14 as 0 and used to maintain the alternate summation. In the end I moved the value of r14 to rax as the return value of the function.
For the allocation of the memory for the matrices I used **syscalls** and the rules of memory allocation in NASM. We have to pass the amount of memory to be allocated in **rdi**. The address where memory has to be allocated is given in **rax**. Then syscall is used.

Observations:

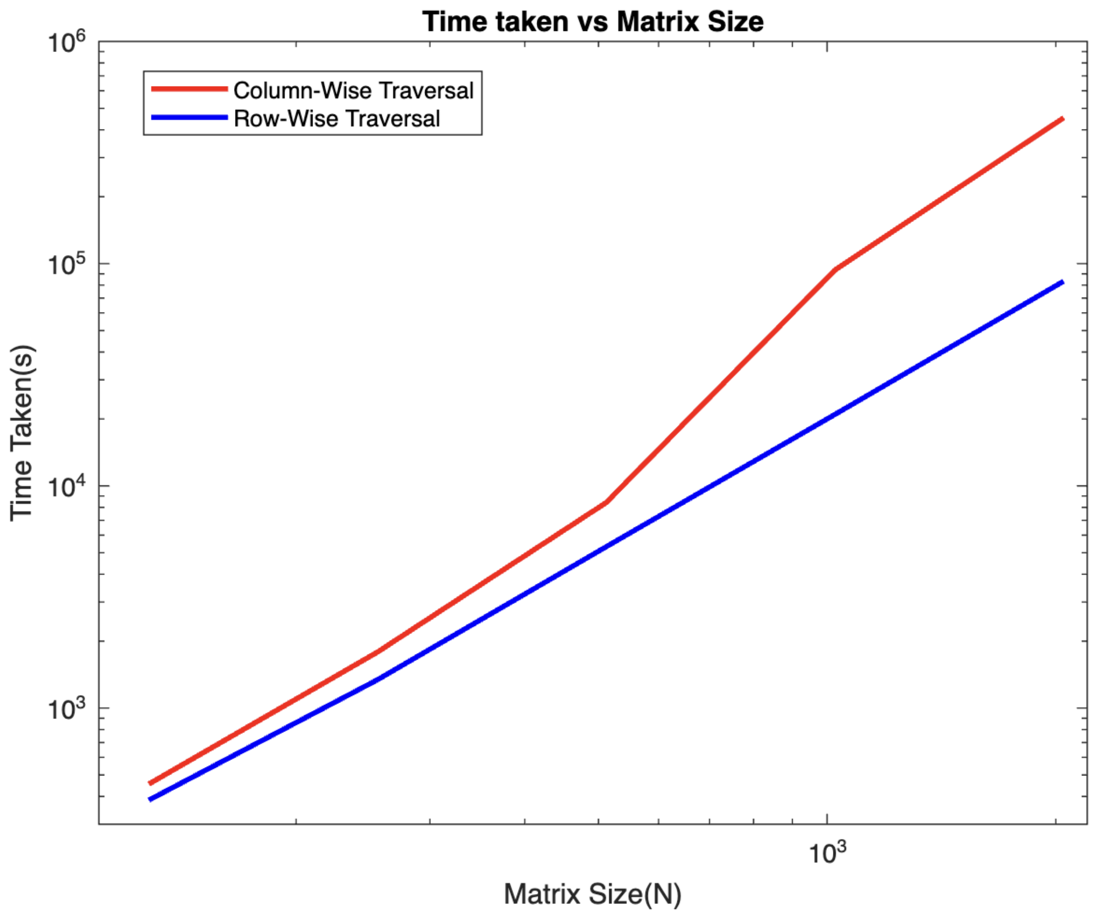
On running both the versions(row and column traversal) of the code we observe that for **larger size** of the matrices there is a **significant difference between the number of clock cycles**.
The row traversal version performs much better. This can be explained using the concept of **Caching**.

Explanation:

Cache is a way of optimising the memory access time. Caching is based on the principal of locality. According to **spatial locality** when a certain part of the memory is accessed, then there is a high chance that its neighbouring memory locations will also be accessed. In practise, this helps in **huge**

reduction in the overall memory access time.
Now we come to our problem. In our code we have stored memory in the following way. The members of a row are **contiguously placed** in memory. Whereas, for every column the next element is N locations ahead. So, when we traverse along a row caching helps because a group of next elements are brought in the cache which **reduces the number of memory accesses**. However, for matrices with large N, for column traversal if the number of elements brought in the cache is **less than N**, then it is not helpful, because the next required element is not in the cache and thus **we have to access the memory again.**
Notice that for **low N**, it may happen that **caching helps even for column traversal**, because the next element required may be brought in the cache because of less number of elements in the row. This is confirmed by looking at the number of clock cycles for small N for column and row traversal. **According to my code, for values of N from 1 to 64, the number of clock cycles for the column and row traversal are almost equal.**

We draw a plot of Time Taken vs Matrix Size for both types of traversals:



The data used for plotting the above is given in the following table:

TSC = 1005.430359 MHz

Matrix Size	Number of Cycles(Row Traversal)	Number of Cycles(Column Traversal)
128	387316	457525
256	1348795	1796687
512	5351204	8457458
1024	21074200	94107379
2048	83786612	455878446

We can then get the value of $\textbf{Time Taken} = \dfrac{\textbf{Number of Clock Cycles}}{\textbf{TSC}}$