Performance Analysis of Matrix Multiplication and Transpose

Abhinav Rajesh Shripad(2022CS11596)

April 24, 2024

1 Introduction

In this report, we graph the time taken for matrix multiplication and transpose operation, and also calculate the **size of L1 cache** of a system based on the data received in case of transpose.

2 Methodology

We implemented matrix multiplication and transpose algorithms in C and compiled the code with the -O3 optimization flag using **gcc**. We measured the implementation time and graphed it. All the data can be found in the submitted zip file.

3 Matrix Multiplication

For a standard brute matrix multiplication algorithm , the time complexity if $O(n^3)$ where n is the size of the matrix. It consists of 3 independent loops. Thus there can be total 6 different ways to arrange these loops. Let the matrices be A_{n*n}, B_{n*n} and C_{n*n} and $C = AB$. For calculating the entry $C[i][j]$ we consider the following 6 ways :-
order1:-i over rows of A, j over columns of B, k over common dimension
order2:-i over rows of A, k over common dimension, j over columns of B
order3:-j over columns of B, i over rows of A, k over common dimension
order4:-j over columns of B, k over common dimension, i over rows of A
order5:-k over common dimension, j over columns of B, i over rows of A
order6:-k over common dimension, i over rows of A, j over columns of B

Claim:-We claim that **(time1,time3)** , **(time2,time6)** and **(time4,time5)** will be approximately equal.

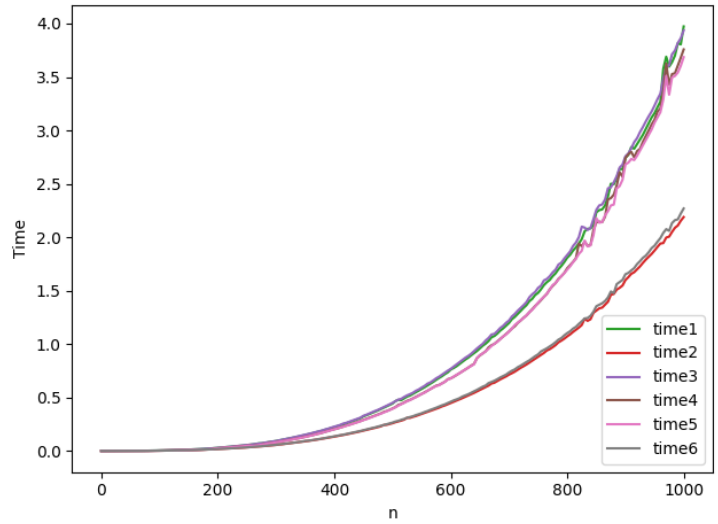

Figure 1: Time vs n

The reason for this is simple, the **"inner most"** loop is same for each of this pair. Explicitly following the code for time1 and time3 following is the code, we can see that inner most loop ie k is same.The reason for this consistency arises from the nature of the matrix multiplication algorithm, where the innermost loop is responsible for accessing the elements of the matrices and performing the multiplication operation. Regardless of the order in which the outer loops iterate, the innermost loop executes the same number of operations.

```
for (int i = 0; i < n; i++) {
for (int j = 0; j < n; j++) {
for (int k = 0; k < n; k++) {
    C[i][j] += A[i][k] * B[k][j];
}}}
```

```
for (int j = 0; j < n; j++) {
for (int i = 0; i < n; i++) {
for (int k = 0; k < n; k++) {
    C[i][j] += A[i][k] * B[k][j];
}}}
```

The proportionality constant for this $O(n^3)$ complexity algorithm, as calculated from the data is :-

Table 1: Proportionality Constants for Matrix Multiplication for different Loop Orderings

	time1	time2	time3	time4	time5	time6
constant	$2.14*10^{-6}$	$1.26*10^{-6}$	$2.17*10^{-6}$	$1.99*10^{-6}$	$1.98*10^{-6}$	$1.29*10^{-6}$

4 Matrix Transpose

We implemented two versions of matrix transpose: in-place transpose and transpose using an additional array. Similarly, we varied the matrix size n and recorded the execution time for each version. Since time complexity for this is only $O(n^2)$ we can sample a larger set of points.

5 Results

Following is the graph of the time taken for each individual methods.

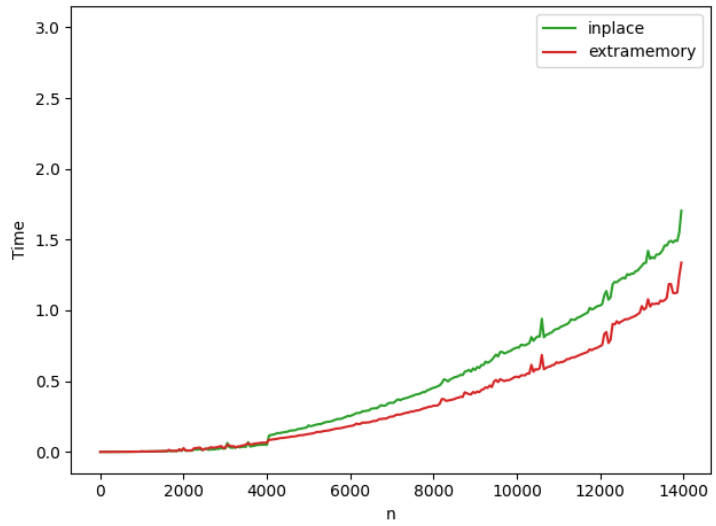

Figure 2: Time vs n

We calculate the time complexity for each method with the time constants also, We assume each individual instruction in C takes equal time. Thus for in-place it is $O(3n^2/2)$ and for extra-memory $O(n^2)$

```
for (int i = 0; i < n; i++) {
for (int j = 0; j < n; j++) {
    res[j][i] = mat[i][j];


}}
```

```
for (int i = 0; i < n; i++) {
for (int j = i + 1; j < n; j++) {
    double temp = mat[i][j];
    mat[i][j] = mat[j][i];
    mat[j][i] = temp;
}}
```

6 Observation

Observe that there is a slight kink or jump in the graph at some value near 4000, let us assume that the kink occurs at $n = 4096$, we will justify this assumption soon enough, stay with us for a moment. So to justify the kink, we find the propotionality constant for the curve's before and after $n = 4096$. It is as follows:- See that the constants are approximately in the ratio 3/2 as calculated

Table 2: Proportionality Constants for Matrix Transpose for different methods

	inplace	**extramemory**
constant	$3.28*10^{-9}$	$2.58*10^{-9}$
constant	$7.32*10^{-9}$	$5.31*10^{-9}$

above, although our main focus is that the huge jump in the proportionality constant. We use it to predict the size of the L1 cache of the system on which these tests are done.

7 Calculation

For a size n matrix, after each loop, there is a total of 2n number of data in the cache, "double" takes 8 bytes in C, thus total memory is 16n bytes, which is our cache size. Now we assume that the cache memory size is a power of 2, so it was a reasonable assumption to assume that the jump in the graph occurs at $n = 4096$, thus the total cache size is 65536 bytes. Which is exactly the l1 data size on the machine (Macbook pro 2022 model), screenshot attached below. To get this data on laptop, enter "sysctl -a | grep cache" on the terminal.

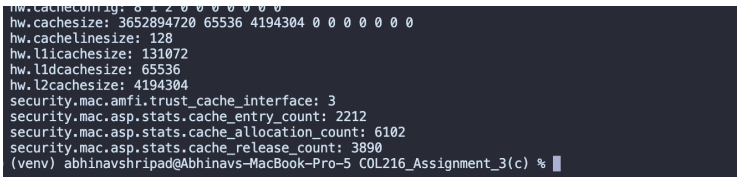

Figure 3: hw.l1dcachesize is the l1 data cache

8 Conclusion

The data utilized for all computations has been meticulously curated and is readily accessible alongside the submission. Additionally, accompanying Python scripts have been thoughtfully crafted to facilitate the retrieval and manipulation of the precise datasets employed herein. While these scripts are versatile, they may necessitate tailored adjustments to suit specific analytical requirements and preferences.