Thinking of a Title

Himanshu Gangwal

October 22, 2023

Abstract

Forgive me, the pdf won't look as the beautiful framework of latex allows, mainly due to tha fact that I am incredibly lazy also due to the fact that I falling behind on every deadline ever possible !!

Contents

1 The Memory Structure

In order to accommodate the matrices of say size n, we allocate $n * n * 8$ contiguous bytes on the stack and then in order to do the given operations we basically employ two different ways to traverse the arrays completely, one is we traverse each element in a row first and then proceed to the next row, and for the second one, we traverse each element in a column first and then move onto the next column. The problem here, I think, revolves around the idea of caching and localities formed in the memory.

Further we shall see and analyze the results of the two traversal schemes by ourselves. First lets state the results of our tests.

2 The Results

The details of testing can be found in the **The Testing Utils** section, ahead. The given are the results of one such run of the makefile, with
$sizes = [128, 256, 512, 1024, 2048]$

```
$ make
Now Testing . . .
row.x avg cycles for Z=128 is 254612
row.x avg cycles for Z=256 is 968169
row.x avg cycles for Z=512 is 3372916
row.x avg cycles for Z=1024 is 14409590
row.x avg cycles for Z=2048 is 62858072
col.x avg cycles for Z=128 is 986881
col.x avg cycles for Z=256 is 7163033
col.x avg cycles for Z=512 is 41652441
col.x avg cycles for Z=1024 is 170360078
col.x avg cycles for Z=2048 is 701300491
Close the window to continue . . .
Plot saved as CvZ.png . . .
```

Before moving onto the results, the TSC of the machine that we're using for testing can be found by using the command

```
$ sudo turbostat --interval 1
```

(see here)

```
Core    CPU    Avg_MHz Bzy_MHz TSC_MHz
 -       -       1.07    2361    3197
```

(see here)

So we have the $TSC\_MHz = 3197$, which is quite close to the base frequency of my machine AMD Ryzen 7 5800H that is 3.2 GHz.

Z	row.x(cycles)	column.x(cycles)	time(us)	time(us)
128	254612	986881	79.5	308.4
256	968169	7163033	302.5	2238.4
512	3372916	41652441	1054	13016.3
1024	14409590	170360078	4503	53237.5
2048	62858072	701300491	19643	219156.4

here row.x and column.x are the two variants. Correspondingly we have the graph,

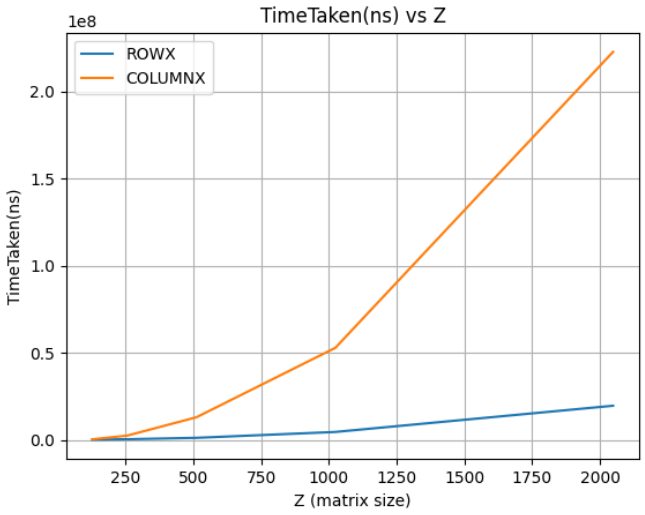

Figure 1: TimeTaken vs Size(Z)

3 Explaining the Results

Now continuing from the memory structure, the difference we see here, in accessing the same memory and doing the same operations is basically due to the order that we are accessing memory in. In the case when we are accessing the memory byte-by-byte in addressing order, for row.x variant, the cache misses would be way less as compared to column.x , as in the case of row.x the next of the current byte resides withing a specified range, and that is too consecutively, so following the pattern there's an opportunity to reduce the access cost by caching as, spatial locality will lead to chunks of data being brought closer into the cache which is then accessed again and again. This is not the case with the column.x variant, as the next element is less likely to be present in the same cached data as the jumps are huge, basically the address that is about to be fetched is at a considerable offset from the current location to be present in the cache, hence, cache misses are more frequent, and thus we take a toll compared to the cache-friendly variant row.x.

Processing over contiguous data, gives an huge advantage as it induces spatial locality in accesses, and as the first few of the elements will be fetched, a great portion of the array or matrix or even local variables stored contiguously in the memory get cached and thus, improving the access time for further instruction.

(read this)
(and this)

4 The Testing Utils

4.1 Makefile

This is the all in one make file that I wrote for testing, just run make and it compiles, tests, plots and shows the results to the user. Alongside this there are three files, one bash scripts to test and generate the results and two python scripts to generate random matrices and other to plot the results. Alongside this, the argument of the bash file in the rule test represents the number of runs to compute the average and can be change accordingly, initially set to 1 and can be varied as per need.

```
all: asmbl test plot clean
asmbl:
        @nasm -felf64 io.asm -o io.o
        @nasm -felf64 matrix-testbench.asm -o matrix-testbench.o
        @nasm -felf64 my_LINCOMB_COL.asm -o my_LINCOMB_COL.o
        @nasm -felf64 my_SUM_COL.asm -o my_SUM_COL.o
        @nasm -felf64 my_PROD_COL.asm -o my_PROD_COL.o
        @gcc -no-pie -g -w io.o my_PROD_COL.o my_SUM_COL.o my_LINCOMB_COL.o
                                matrix-testbench.o -o col.x 2>/dev/null
        @nasm -felf64 my_LINCOMB_ROW.asm -o my_LINCOMB_ROW.o
        @nasm -felf64 my_SUM_ROW.asm -o my_SUM_ROW.o
        @nasm -felf64 my_PROD_ROW.asm -o my_PROD_ROW.o
        @gcc -no-pie -g -w io.o my_PROD_ROW.o my_SUM_ROW.o my_LINCOMB_ROW.o
                                matrix-testbench.o -o row.x 2>/dev/null
test:
        @echo "Now Testing . . ."
        @bash tester.sh 1
plot:
        @echo "Close the window to continue . . ."
        @python3 plotter.py
        @echo "Plot saved as CvZ.png . . ."
clean:
        @rm results.txt
        @rm input.txt
        @rm *.o
        @rm *.x
```

4.2 maketrix.py

This file is used to generate the pseudo-random input matrices and scalers, it takes an argument that is the size of the square matrix.

```
import random
import sys
from random import randint
Z=int(sys.argv[1])
print(Z)
for i in range(5):
        y=random.randint(1,1000)
        print(y)
for j in range(5):
        for i in range(Z**2):
                y=random.randint(1,10000)
                print(y)
```

Sample usage (As you can see I do not have space for Z greater than 1 :)
usage :

```
   $ python3 maketrix.py 1
1
994
783
356
812
597
11
9188
9025
9643
1319
```

4.3 tester.sh

This is the file which tests the programs for 5 different values of Z (the matrix size). It calls maketrix.py to generate the matrices and stores the averaged out value for each size Z in a file names *results.txt*, which is then used by plotter.py. (n_avg is the number of iterations for averaging at each size)

```
################################################
#!/bin/bash
n_avg=$1
sizes[0]=128
sizes[1]=256
sizes[2]=512
sizes[3]=1024
sizes[4]=2048
names[0]=row.x
names[1]=col.x
rm results.txt 2>/dev/null
for k in ${names[@]}
do
        for i in ${sizes[@]}
        do
                j=$n_avg
                sum=0
                while [ $j -gt 0 ]
                do
                        python3 maketrix.py $i > input.txt
                        ./$k < input.txt > temp.txt
                        x=`sed -n '2p' temp.txt`
                        rm temp.txt
                        let "sum=x+sum"
                        let "j=j-1"
                done
        echo $((sum / n_avg)) >> results.txt
        echo $k "avg cycles for Z=$i is" $((sum / n_avg))
        done
done
################################################
```

usage :

```
$ bash tester.sh <n_avg>
```

4.4 plotter.py

This is the last step, this script reads the results.txt file generated by the tester and creates a plot for the same, which then gets saved as *CvZ.png*

```
################################################
import matplotlib.pyplot as plt

def stripper(lines):
    y = []
    for line in lines :
        y.append(float(line.strip()))
    return y

with open("results.txt", "r") as file:
    lines = file.readlines()

rowx = lines[:5]
colx = lines[5:10]

rowx = [i/3.2 for i in stripper(rowx)]
colx = [i/3.2 for i in stripper(colx)]

sizes = [128, 256, 512, 1024, 2048]
plt.plot(sizes, rowx, label="ROWX")
plt.plot(sizes, colx, label="COLUMNX")
plt.xlabel('Z (matrix size)')
plt.ylabel('TimeTaken(ns)')
plt.legend()
plt.title('TimeTaken(ns) vs Z')
plt.grid(True)
plt.savefig("TvZ.png")
plt.show()
################################################
```

4.5 sampleOutput

```
$ make
Now Testing . . .
row.x avg cycles for Z=128 is 260380
row.x avg cycles for Z=256 is 900019
row.x avg cycles for Z=512 is 3619046
row.x avg cycles for Z=1024 is 14406812
row.x avg cycles for Z=2048 is 63443363
col.x avg cycles for Z=128 is 949635
col.x avg cycles for Z=256 is 7207996
col.x avg cycles for Z=512 is 41006489
col.x avg cycles for Z=1024 is 172541379
col.x avg cycles for Z=2048 is 716986758
Close the window to continue . . .
Plot saved as TvZ.png . . .
```

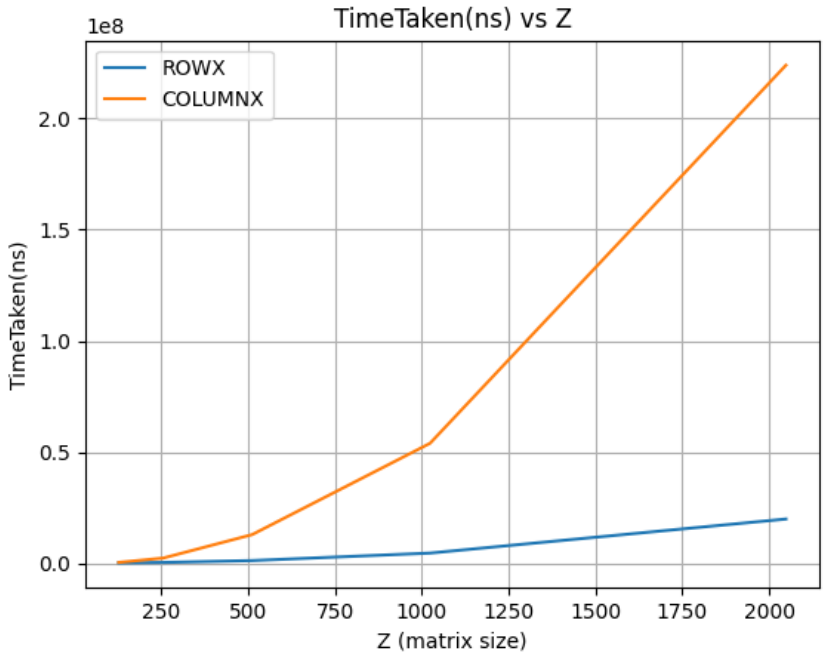

Figure 2: TvZ.png