CS232 Lab 3 Report

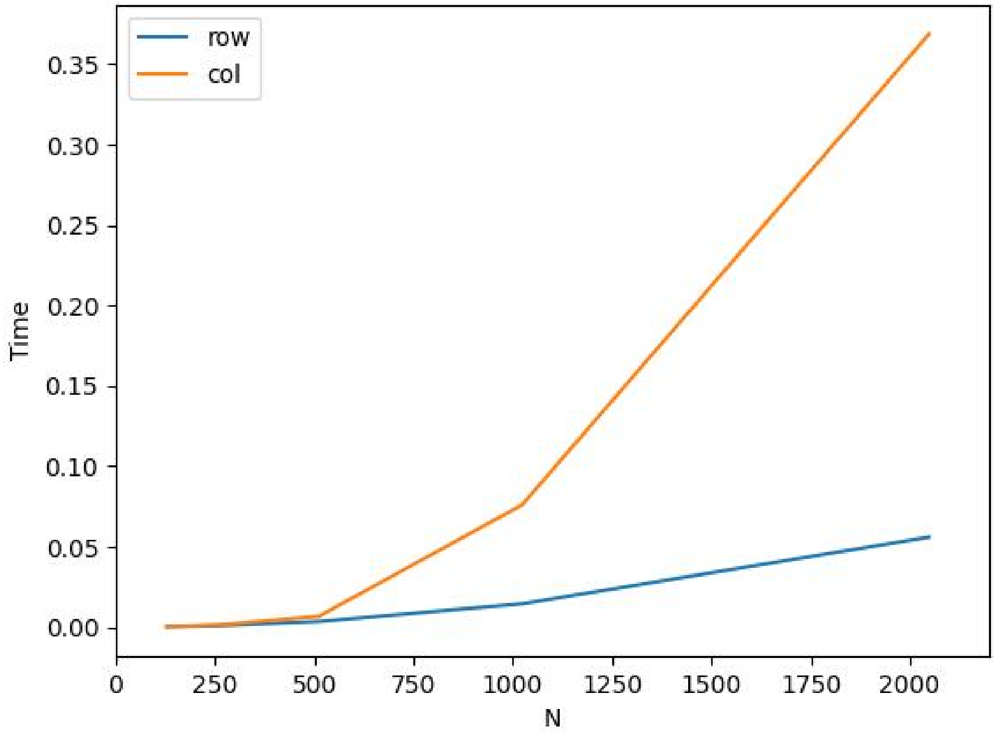
Part 3

Rijul Bhat (22B0971)

Contents

1 Measurements

Plot: Time v/s N



TSC = 1002.460571 MHz.

Accessing Row Wise

N	no. of cycles	time = no of cycles/TSC(s)
128	299141	0.00029840674900718866
256	980566	0.0009781591699131278
512	3593883	0.003585061701144952
1024	14606570	0.01457071771454241
2048	56135762	0.05599797500663994

Table 1: Calculating using Row Wise Access

Accessing Column Wise

N	**no. of cycles**	**time = no of cycles/TSC**(s)
128	328575	0.0003277685023284572
256	1383500	0.001380104155737413
512	6868916	0.006852056029643085
1024	76376583	0.0761891142748995
2048	369593316	0.36868613758176433

Table 2: Calculating using Column Wise Access

2 Approach and Implementation

- **Overview**
  The difference between the approaches in row and column execution is that accessing the indices is much easier in case of rows. In case of rows, we just need to iterate through them in the order in which they are stored. In case of column wise iteration, the data being stored in row major format each time we have to iterate we have to access the indices in increment intervals of `8n` bytes where `nxn` is the dimension of matrix and then come back to the first row every a column has been iterated which is inefficient and cumbersome.
  Also, another important observation was that r12 has been used in the testbench for calculating number of cycles so if we are using r12 in our program we must push and pop and in the stack to preserve its value for the testbench.

- **Linear Combination**
  Registers `r14` and `r15` are being used as the 2 loop counters `i` and `j`, the increment value i.e. `ni+j` (in case of row wise iteration) is stored in `rbx`. [a1], [a2] are stored in `r13`, `r12` and [b1] is stored in r10, incrementing those addresses by rbx to give access to the corresponding

```
push r12
mov r14, 0

counter_i:
cmp r14, r9
je exit_i
        mov r15, 0

        mov rbx, r14
        imul rbx, r9
        sal rbx, 3

        counter_j:
        cmp r15, r9
        je exit_j

        mov r13, rdi
        mov r12, rdx
        mov r10, r8

        add r13, rbx
        add r12, rbx
        add r10, rbx

        mov r13, [r13]
        mov r12, [r12]

        imul r13, rsi
        imul r12, rcx
        add r13, r12

        mov [r10], r13


        add rbx, 8

        add r15, 1
        jmp counter_j
exit_j:
add r14, 1
jmp counter_i
exit_i:
pop r12
```

```
push r12
mov r14, 0

counter_i:
cmp r14, r9
je exit_i
        mov r15, 0

        mov rbx, r14
        sal rbx, 3

        counter_j:

        cmp r15, r9
        je exit_j

        mov r13, rdi
        mov r12, rdx
        mov r10, r8

        add r13, rbx
        add r12, rbx
        add r10, rbx

        mov r13, [r13]
        mov r12, [r12]

        imul r13, rsi
        imul r12, rcx
        add r13, r12

        mov [r10], r13

        shl r9, 3
        add rbx, r9
        shr r9, 3
        add r15, 1
        jmp counter_j
exit_j:
add r14, 1
jmp counter_i
exit_i:
pop r12
```

index. Now value stored in r12 and r13 is multiplied by the scalars `rcx`, `rsi` and added then stored in [r10] i.e. address of index `i` and `j` for [b1].

- **Hadamard Product**
  The program is extremely similar to the previous part so only the program inside the for loop is differing, here the product of r12 and r13 is to be taken instead of linear combination and store it in r10.

```
        mov r13, rdi
        mov r12, rdx
        mov r10, r8

        add r13, rbx
        add r12, rbx
        add r10, rbx

        mov r13, [r13]
        mov r12, [r12]

        imul r13, r12

        mov [r10], r13

        add rbx, 8
```

```
        mov r13, rdi
        mov r12, rdx
        mov r10, r8

        add r13, rbx
        add r12, rbx
        add r10, rbx

        mov r13, [r13]
        mov r12, [r12]

        imul r13, r12

        mov [r10], r13
        shl r9, 3
        add rbx, r9
        shr r9, 3
```

- **Alternate Sum**
  Here I have maintained value of `i+j` in `rcx` and exploited the fact that the least significant bits of odd decimal number is 1 and even is 0, so we can just **and** the value in `rcx` by 1, and determine whether we have to multiply by -1 or not while taking the sum.

```
        mov r13, rdi
        add r13, rbx
        mov r13, [r13]

        mov rcx, r15
        add rcx, r14

        and rcx, 1

        cmp rcx, 0
        jne one

        add rax, r13

        jmp skip_one

        one:

        imul r13, -1
        add rax, r13

        skip_one:

        add rbx, 8
```

```
        mov r13, rdi
        add r13, rbx
        mov r13, [r13]

        mov rcx, r15
        add rcx, r14

        and rcx, 1

        cmp rcx, 0
        jne one

        add rax, r13

        jmp skip_one

        one:

        imul r13, -1
        add rax, r13

        skip_one:
        shl r9, 3
        add rbx, r9
        shr r9, 3
```

- **Time, TSC and Cycle Count**

```c
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>

static inline uint64_t rdtsc(void)
{
```

```c
uint32_t lo,hi;
__asm__ __volatile__ ("rdtsc": "=a" (lo), "=d" (hi));
return ((uint64_t)hi << 32) | lo;
}

int main(void)
{
uint64_t tsc1 = rdtsc();
sleep(1);
uint64_t tsc2 = rdtsc();
printf("%f\n", ((float) (tsc2 - tsc1))/(1000000));
return 0;
}
```

The above C code computes the clock frequency of our processor using the TSC register and returns the output in MHz. I got this value close to **1000 MHz** every time the given program was executed on my Macbook Air using Docker.

3 Observations and Conclusion

We conclude that the row-wise access performs significantly better than the column-wise access. Row-wise access patterns take advantage of spatial proximity and cache line prefetching, resulting in fewer cache misses and quicker data retrieval, making them ideal for cache memory behaviour. In contrast, because column-wise access skips between memory regions and only partially utilizes spatial locality, it frequently results in more cache misses and is slower.