ACA Assignment 1

Part 1

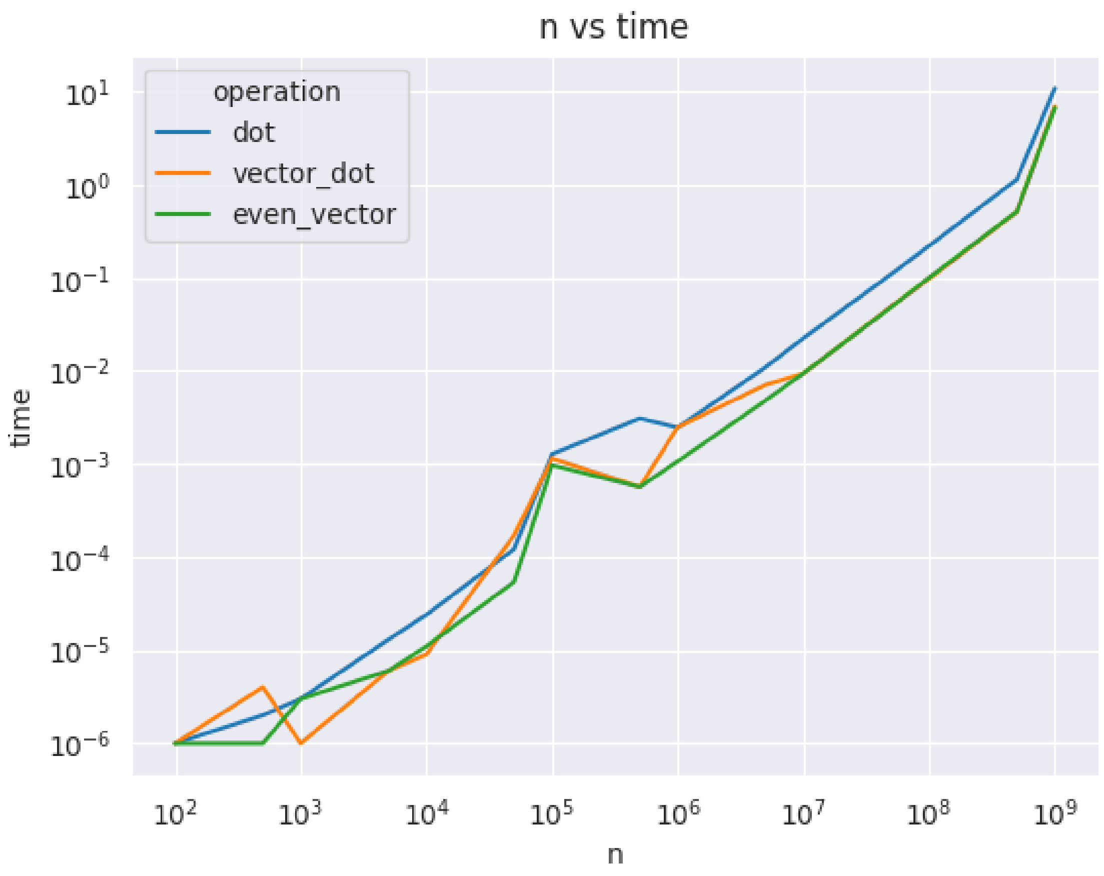
(a), (b) and (c)

The resepective programs for parts (a),(b) and (c) are in the files `dot.c`, `vector_dot.c` and `even_vector.c` respectively. The programs in part (a) and (b) perform the dot product of two vectors using scalar and vector operations respectively. The program in part (c) performs the dot product of two vectors using vector operations but only for even indices. AVX2 has been used for vector operations.

I have run these three programs and constructed the final `out.csv` using the following commands:

```
gcc dot.c -o dot
gcc -mavx2 vector_dot.c -o vector_dot
gcc -mavx2 even_vector.c -o even_vector

echo "n,dot,vector_dot,even_vector" > dot.csv

arg=16000
for i in {1..18}
do
    echo "Matrix Size: $arg"
    echo -n "$arg," >> dot.csv
    ./dot $arg >> dot.csv

    echo -n "," >> dot.csv
    ./vector_dot $arg >> dot.csv

    echo -n "," >> dot.csv
    ./even_vector $arg >> dot.csv

    echo "" >> dot.csv

    arg=$((arg*2))
done

rm dot
rm vector_dot
rm even_vector
```

This runs on all three programs on vector sizes ranging from 10^2 to 10^9 and stores the output in `out.csv`. The output is plotted in the following graph (time and n are in the log scale) -



These results clearly indicate how the vector operations improve the perfomance of the program. The program in part (c) is not much faster than the program in part (b) as in the vector operation, it has to load 4 consecutive elements and cannot store only even indices. This makes it store all values in the vector register and then perform the operation.

(d)

This program computes the memory bandwidth of a system. This uses a simple array to calculate both the read and write bandwidth. All the elements in the array are first read using a for loop and the write bandwidth is calculated by write to that same array. The array is of size 256 MB so it can't fit in either of the caches. The program is in the file `bandwidth.c`.

The program outputs the following:

```
Read Bandwidth: 1.82 GB/s
Write Bandwidth: 1.78 GB/s
```

(e)

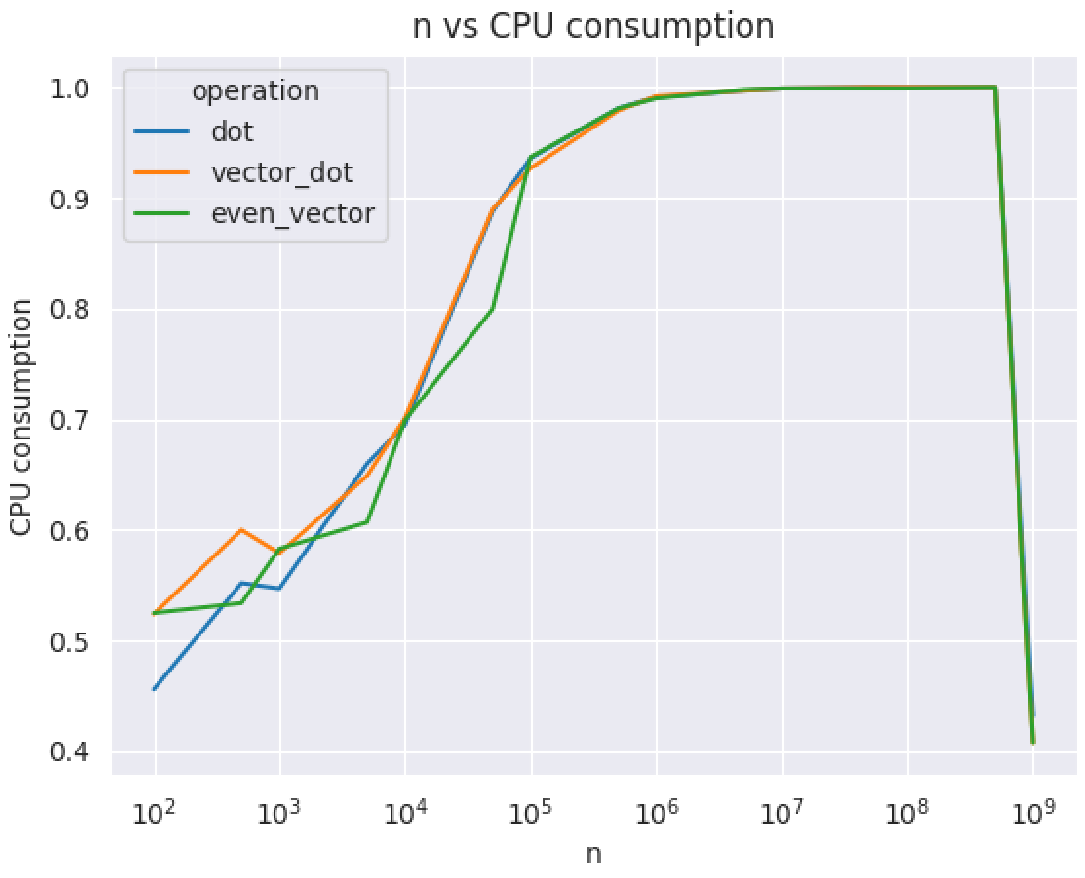
The program in part (e) calculates peak GFLOPS of the system. It uses basic floating point addition to calculate the GFLops of the system. The operation is executed 1e5 times to ensure correctness. The program is in the file `gflops.c`.

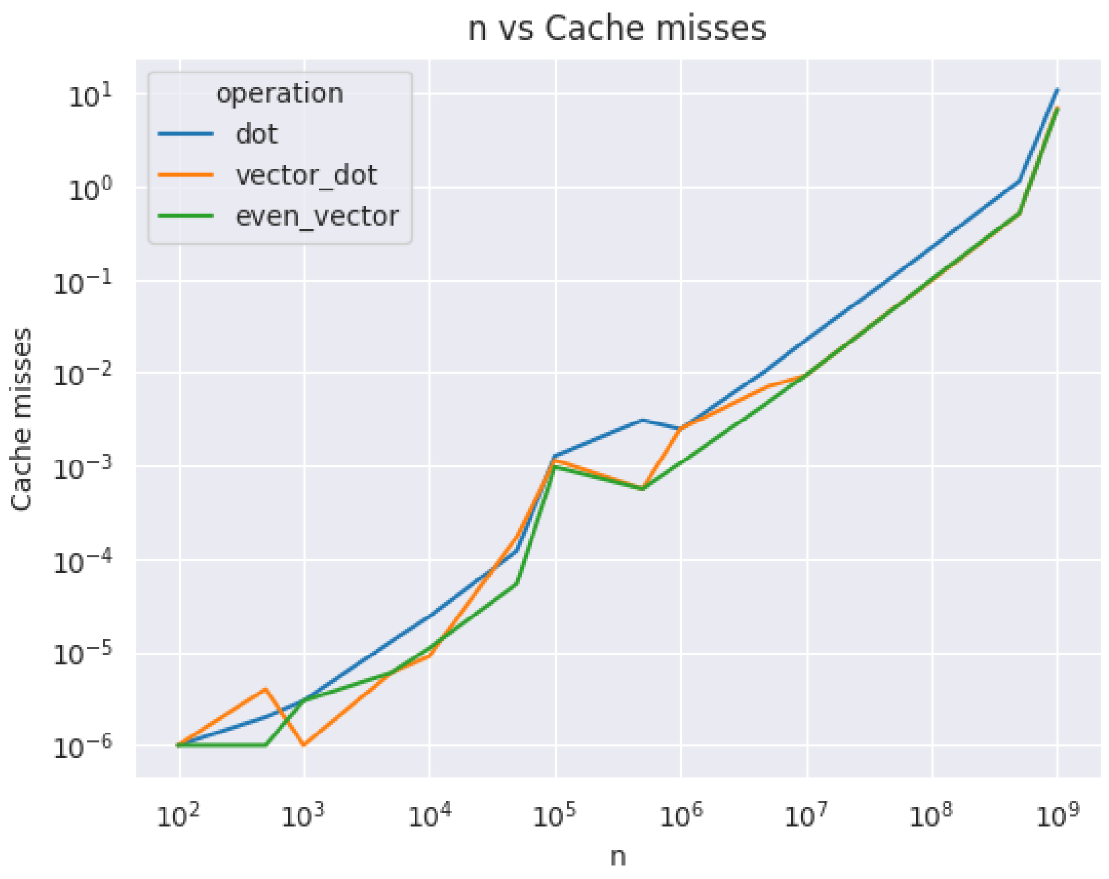
The program outputs the following for my system:

```
1.102124 GFLOPS
```

(f)

The programs in part (a),(b) and (c) are all memory bound for large n as they have to wait for fetching the array and have to read large portions of the array. This is clearly indicated by the results of these programs as the CPU consumption remains approximately same but the cache hits keep going up with increasing n. This shows that the CPU is waiting for memory to be fetched for all operations.


n vs CPU consumption
operation
dot
vector_dot
even_vector
CPU consumption
n
1.0
0.9
0.8
0.7
0.6
0.5
0.4
10^2
10^3
10^4
10^5
10^6
10^7
10^8
10^9



Part 2

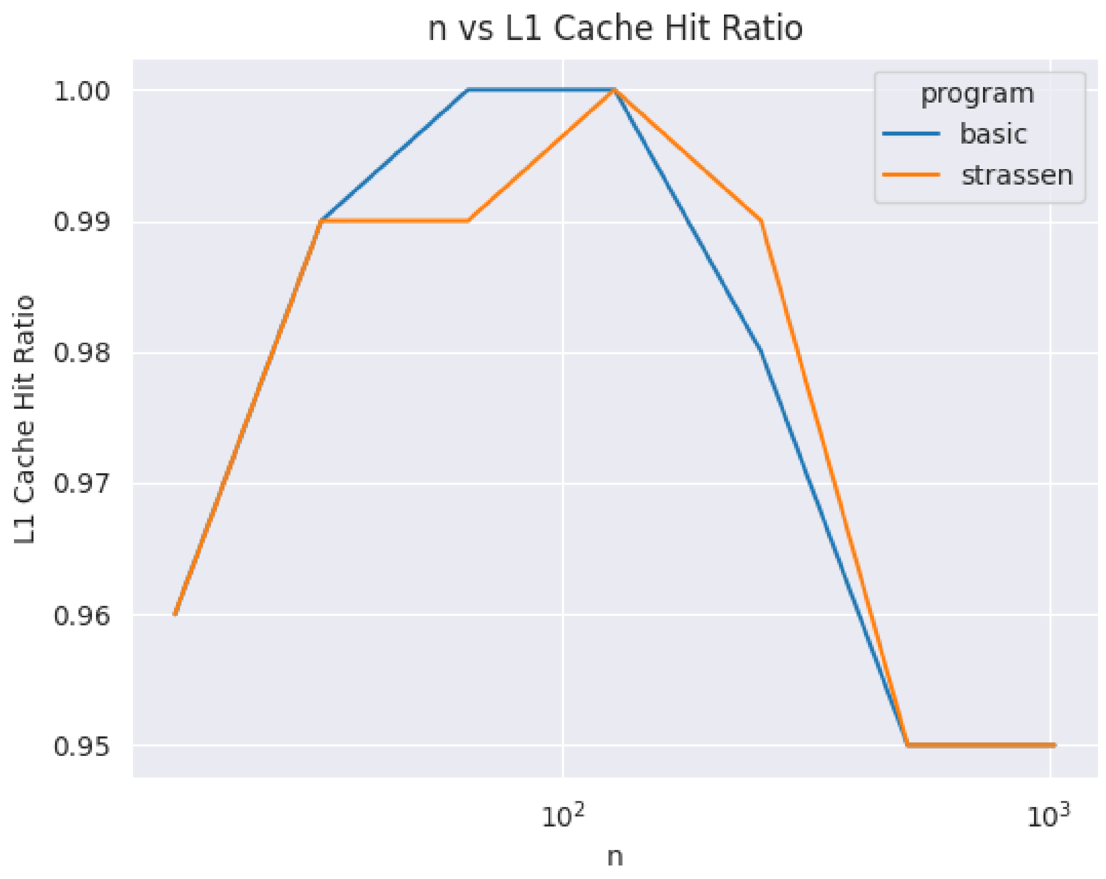
The programs for part (a), (b) and (c) are in `basic.c`, `strassen.c` and `tiled.c` respectively. All of them are different variations of matrix multiplication.

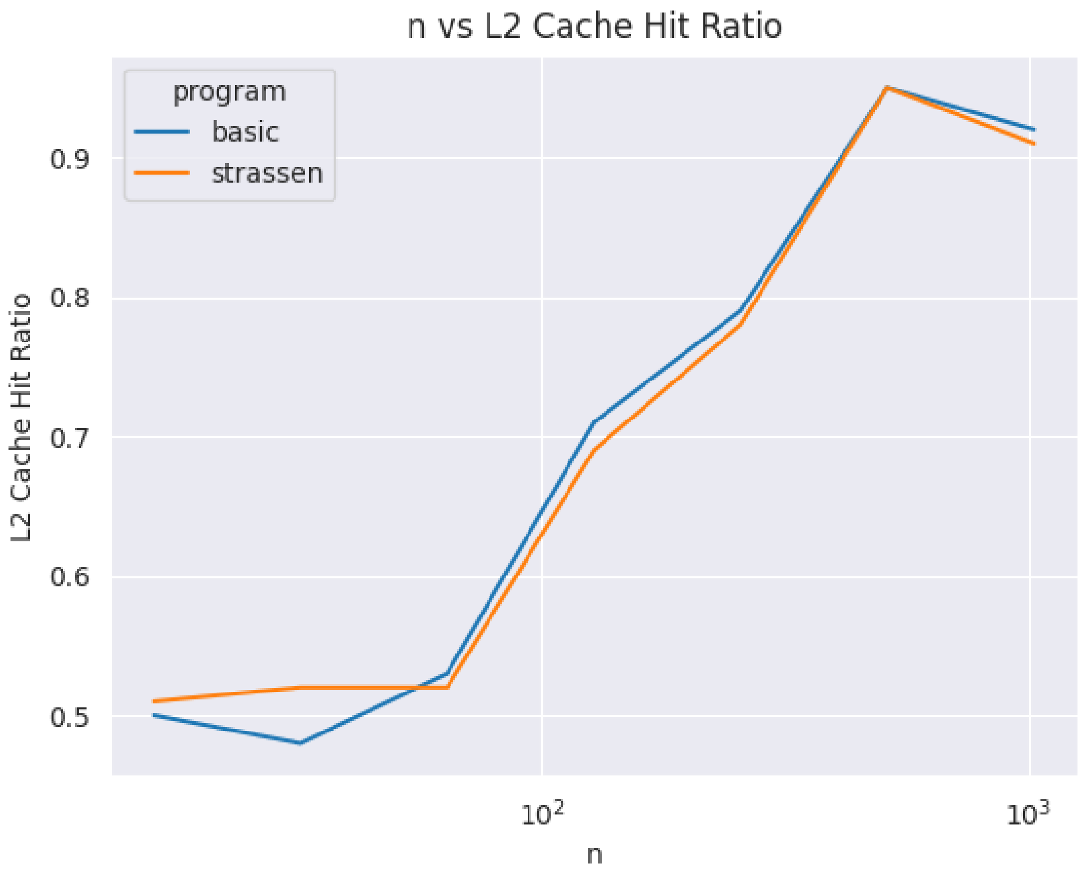
For profiling cache misses, I have used the perf profiler and the script `execute.sh` to run the program for different values of n. The script is:

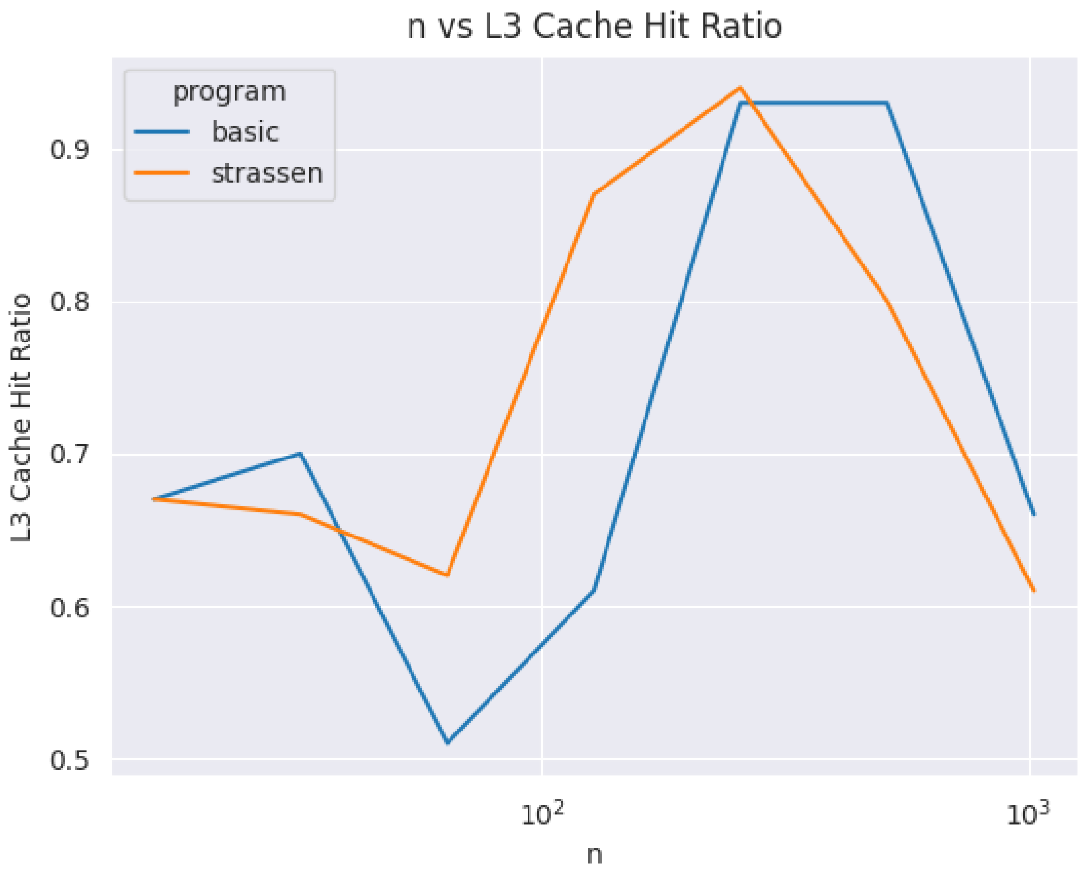
```
gcc --std=c11 -o basic -g basic.c
gcc --std=c11 -o strassen -g strassen.c

BASIC="./basic"
STRASSEN="./strassen"

echo "n,program,instructions,CPI,l1-cache-hit ratio,l2-cache-hit ratio,l3-cache-hit-ratio,
time" > out.csv

L1_EVENTS="L1-dcache-loads,L1-dcache-load-misses"
L2_EVENTS="l2_rqsts.all_demand_references,l2_rqsts.all_demand_miss,LLC-stores,LLC-store-mi
```

```
sses"
L3_EVENTS="LLC-loads,LLC-load-misses"
CPI="instructions,cycles"

> temp.txt

arg=16
for i in {4..10}
do
    echo -n "$arg,basic," >> out.csv

    perf stat -e "$L1_EVENTS" "$BASIC" "$arg" &>> temp.txt
    perf stat -e "$L2_EVENTS" "$BASIC" "$arg" &>> temp.txt
    perf stat -e "$L3_EVENTS" "$BASIC" "$arg" &>> temp.txt
    perf stat -e "$CPI" "$BASIC" "$arg" &>> temp.txt
    python3 process.py temp.txt >> out.csv

    > temp.txt

    echo -e -n "$arg,strassen," >> out.csv

    perf stat -e "$L1_EVENTS" "$STRASSEN" "$arg" &>> temp.txt
    perf stat -e "$L2_EVENTS" "$STRASSEN" "$arg" &>> temp.txt
    perf stat -e "$L3_EVENTS" "$STRASSEN" "$arg" &>> temp.txt
    perf stat -e "$CPI" "$STRASSEN" "$arg" &>> temp.txt
    python3 process.py temp.txt >> out.csv

    > temp.txt

    echo "Done with $arg"

    arg=$((arg*2))
done


rm temp.txt basic strassen
printf "Done writing to out.csv\\n"
```

The final output for these programs is stored in `out.csv`. The graphs which show the final results are given below:


n vs L1 Cache Hit Ratio
L1 Cache Hit Ratio
1.00
0.99
0.98
0.97
0.96
0.95
10^2
10^3
n
program
basic
strassen


n vs L2 Cache Hit Ratio
program
basic
strassen
L2 Cache Hit Ratio
0.5
0.6
0.7
0.8
0.9
10^2
10^3
n



This shows us the performance of the strassen algorithm keeps getting better for bigger n. This is because of the time complexity of the algorithm and the strassen cache algorith is also more cache friendly. The basic algorithm is not cache friendly and hence the performance is not as good as the strassen algorithm.

For part (c), I have used a script that runs the program for different tile sizes and stores the output in `out.csv`. The dimension of the matrix used for the profiling is 512x512 The script is:

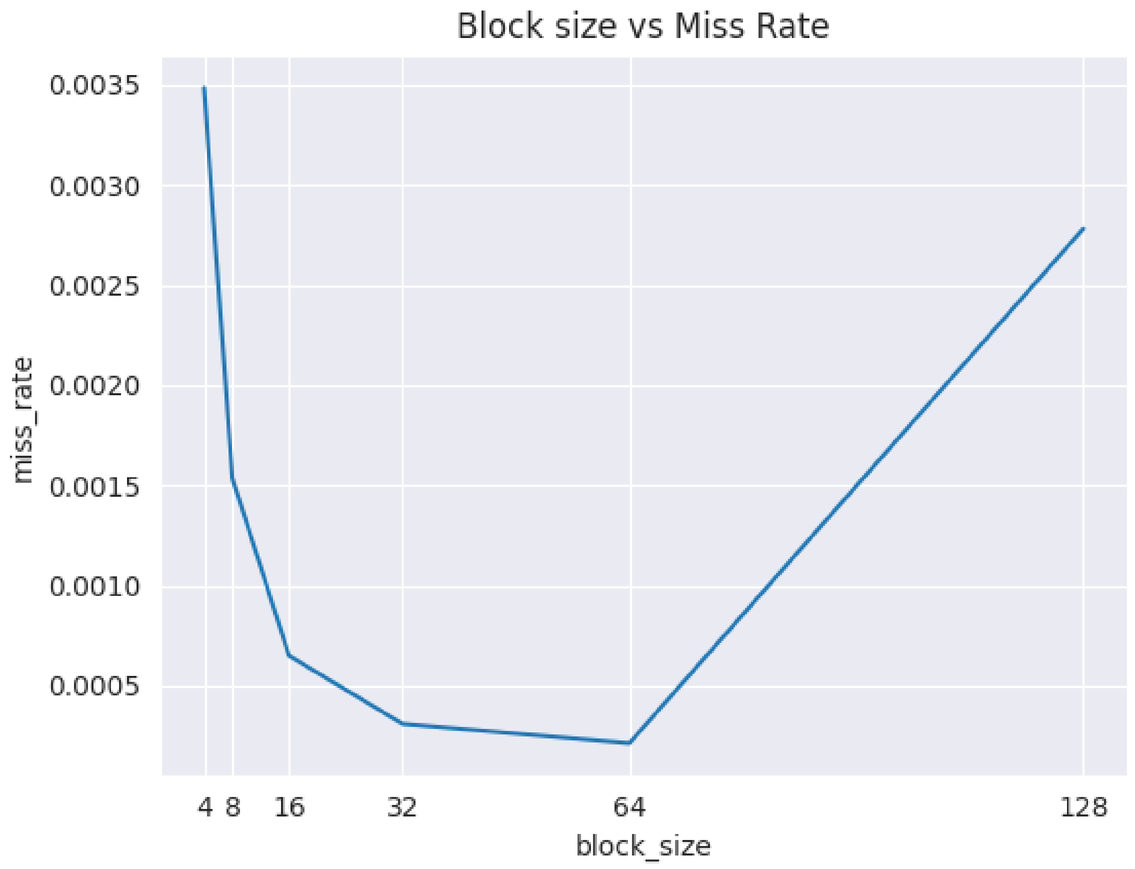
```
SOURCE_FILE="/home/adyanshk/Personal/courses/ACA/A1/2/c/tiled.c"

gcc -g ./tiled.c -o tiled

BLOCK_SIZES=(4 8 16 32 64 128)

L1_SIZE=$((32*1024))  # 32 KB
L2_SIZE=$((1024*1024))  # 1 MB
LINE_SIZE=64  # 64 bytes
```

```
L1_LINES=$((L1_SIZE / LINE_SIZE))
L2_LINES=$((L2_SIZE / LINE_SIZE))

echo "block_size,miss_rate" > out.csv

for BLOCK_SIZE in "${BLOCK_SIZES[@]}"
do
  echo "Running with block size: $BLOCK_SIZE..."

  valgrind --tool=callgrind --cache-sim=yes --D1=$L1_SIZE,$L1_LINES,$LINE_SIZE --L2=$L2_SI
ZE,$L2_LINES,$LINE_SIZE ./tiled "$BLOCK_SIZE" > output 2>&1

  python3 process.py output $BLOCK_SIZE >> out.csv
done

rm tiled callgrind.out.* output
```

The output of the program is:

```
time,instructions,block_size,miss_rate
149,8171097874,4,0.0034850679802710333
125,7754951826,8,0.0015366366039358096
106,7582593893,16,0.0006499460055023385
105,7504225477,32,0.00031003563783085764
101,7466854133,64,0.00021350014986870768
122,7448604224,128,0.0027805491598219215
```

This clearly shows that 64 is the best tile size for the program. The program is in `tiled.c` and the graph is:



On comparing with the matrix multiplication program in part (b), we see that the miss rate is much lower for the tiled program. This is because the tiled program is more cache friendly and hence has a lower miss rate. Even the instruction count has been reduced by a factor of 10. This is because the tiled program has to perform less operations than the strassen program. This shows us the importance of cache friendly programming and how much of a drastic change it can have on the performance.

> NOTE: The times given in the output are not accurate as the program was run using valgrind which is a cache simulator, hence not being accurate for running on actual hardware. The times are given just for reference.

Part 3

The given program has been implemented in `3.c`. I have used a script to run through different values of l1 and l2 sizes to find the optimal cache configuration using fully associative caches. I used callgrind to profile the program and the script is:

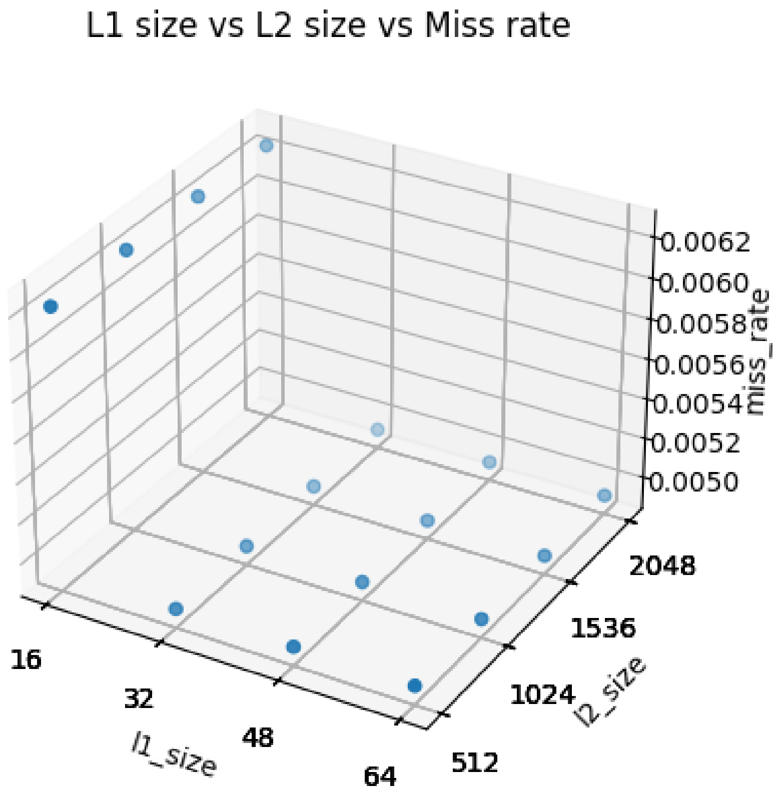
```
L1_sizes=(16 32 48 64)
L2_sizes=(512 1024 1536 2048)
LINE_SIZE=64

echo "time,l1_size,l2_size,miss_rate" > out.csv

gcc -g -o 3 3.c

for l1_size in "${L1_sizes[@]}"; do
    for l2_size in "${L2_sizes[@]}"; do
        echo "l1_size: $l1_size, l2_size: $l2_size"

        start=`date +%s%N`
        valgrind --cache-sim=yes --tool=callgrind --D1=$((l1_size*1024)),$((l1_size*1024/
$LINE_SIZE)),$LINE_SIZE --LL=$((l2_size*1024)),$((l2_size*1024/$LINE_SIZE)),$LINE_SIZE ./3
> output 2>&1
        end=`date +%s%N`
        runtime=$(((end-start) / 1000000000))

        python3 process.py output $l1_size $l2_size $runtime >> out.csv

    done
done

python3 graph.py

rm callgrind.out.* 3 output
```

The program gives these values as output:

```
time,l1_size,l2_size,miss_rate
51,16,512,0.006232737769443012
65,16,1024,0.0062326716474593695
78,16,1536,0.006232621443731048
98,16,2048,0.006232572464483905
73,32,512,0.004934026092866166
99,32,1024,0.0049339611953637015
114,32,1536,0.00493391099163538
138,32,2048,0.004933862012388238
63,48,512,0.004933945277108381
82,48,1024,0.004933880379605916
101,48,1536,0.004933830175877595
107,48,2048,0.004933781196630452
```

```
53,64,512,0.004933898746823595
85,64,1024,0.00493383384932113
92,64,1536,0.004933783645592809
165,64,2048,0.0049337346663456665
```

The best cache configuration is 64 bytes for L1 and 512 bytes for L2. This is because it takes the minimum time to execute alnog with a very less miss rate. We see that the miss rates are majorly equal for all configurations beyond 16. This is because majority of the misses occur because of the L1_cache. The 3D graph between l1 size, l2 size and miss rate is:


L1 size vs L2 size vs Miss rate
miss_rate
l2_size
l1_size