Report - Assignment 1

Inderpreet Singh Chera
160101035

Problem Statement:

To understand the effect of Cache.

System Setup:

All tests were done on **Intel(R) Core(TM) i3-5005U CPU @ 2.00GHz** processor.

- **Clock Speed:**
  CPU max MHz: *1900.00*
  CPU min MHz: *500.00*
- **Memory Size:** *8076584 kB = 7887 MB*
- **Cache Sizes:**
  L1d cache: *32KB*
  L1i cache: *32KB*
  L2 cache: *256KB*
  L3 cache: *3072KB*
- **Theoretical Estimations for different values of 'N':**
  Assumptions: Multiplication requires 3 clock cycles and Addition requires 1 clock cycle. We are assuming both the matrices are of size NxN.
  Now for matrix multiplication, we have for each element N multiplications and N-1 additions. And we do this for all N^2 elements, hence theoretically we will be requiring $N^2(3N_{(\text{for multiplication})} + (N-1)_{(\text{for addition})}) = N^2(4N-1)$ clock cycles. Hence theoretical time required is **$(4N^3 - N^2)$/(Clock Speed).**
  I am taking maximum clock speed of my pc, hence 1900 MHz.
  - N=32:
    $(4*32^3 - 32^2)/(1.9 \times 10^9) = 68.449 \times 10^{-6}$ s.
  - N=64:
    $(4*64^3 - 64^2)/(1.9 \times 10^9) = 0.000549726$ s.
    SImilarly,
  - N=128: 0.00440643 s.
  - N=256: 0.035286 s.
  - N=512: 0.282426 s.
  - N=1024: 2.25996 s.
  - N=2048: 18.0819 s.
  - N=4096: 144.664 s.

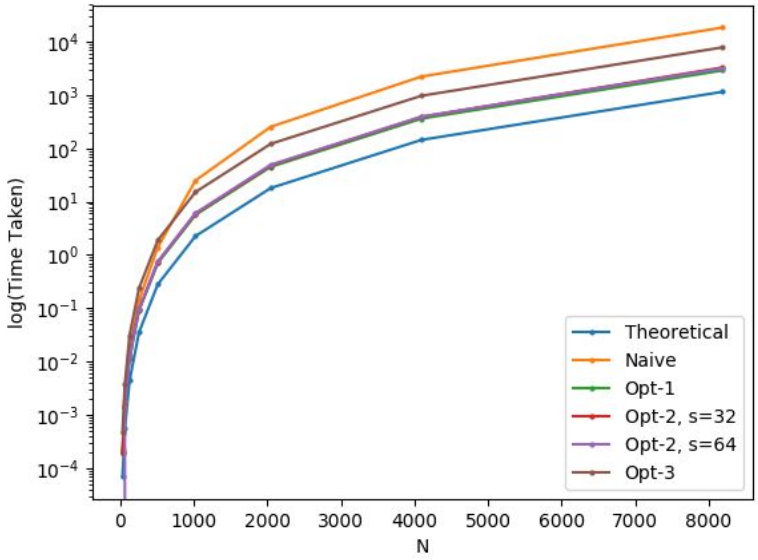
- N=8192: 1157.35 s.

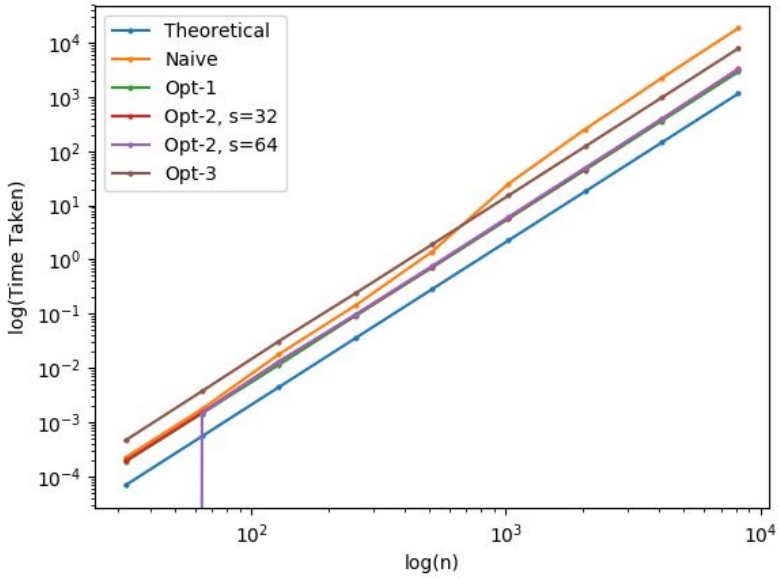
- **Theoretical estimation of 's' for optimization 2:**
  s^2 = O(Memory Size)
  Therefore **s = O($\sqrt{M}$)**
  Now, cache memory size is 32kB = 32 x 1024 Bytes.
  Assuming 1 integer takes 4 bytes,
  The max. number of integers that can fit in L1d cache = sqrt(32*1024/4) = **90.5**.
  Keeping s power of 2 for good division of matrices, s = 64 is used for testing.

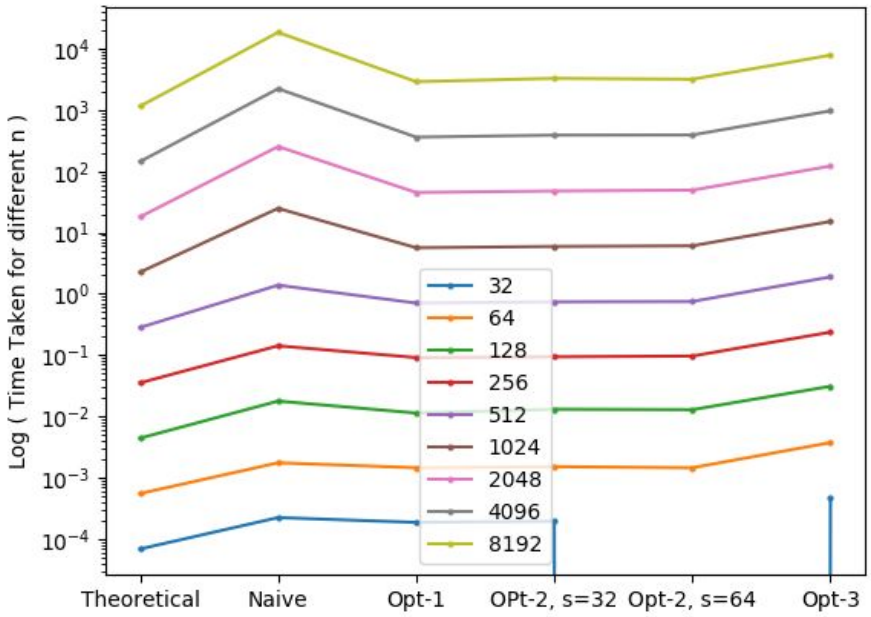
Values Observed:

- **During all the tests it was insured no other application was running.**
- I have taken 2 values of 's' for optimisation 2, **s=32** and **s=64**.
  When n=32 and s=64, the code won't run and hence will take 0 seconds.
- Different values of n that were used are all power of 2 starting from 32, 64 till 8192.

N	Theoretical (in sec)	Naive (in sec)	Opt-1 (in sec)	Opt-2 (s = 32) (in sec)	Opt-2 (s = 64) (in sec)	Opt-3 (in sec)
32	6.8446e-05	0.000223	0.000186	0.000194	0.00000	0.000461
64	0.00054973	0.001749	0.001451	0.001511	0.001451	0.003732
128	0.00440643	0.017780	0.011352	0.013084	0.012878	0.031031
256	0.035286	0.142042	0.091328	0.093948	0.096684	0.237299
512	0.282426	1.388570	0.708196	0.742125	0.751683	1.891817
1024	2.25996	24.969940	5.667843	5.950698	6.097923	15.196035
2048	18.0819	254.91187	45.265446	47.896635	49.376596	122.09030
4096	144.664	2233.6435	362.50003	391.20009	393.35201	973.81965
8192	1157.35	18600.950	2914.1899	3304.4068	3192.5985	7894.4289


log(Time Taken)
N
Theoretical
Naive
Opt-1
Opt-2, s=32
Opt-2, s=64
Opt-3


log(Time Taken)
log(n)
Theoretical
Naive
Opt-1
Opt-2, s=32
Opt-2, s=64
Opt-3

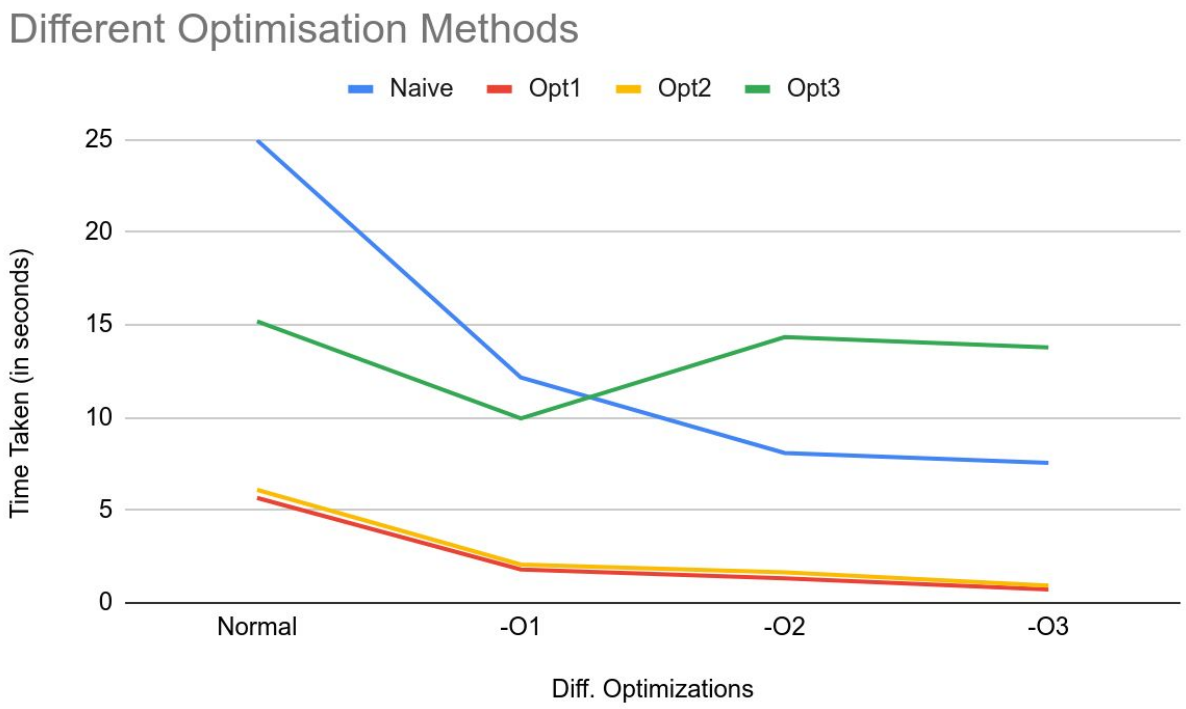


Optimisations:

Initially, I tried to optimise my code by removing repetitive multiplications from my code and tried to convert into additions as much as possible. I saw a significant decrease in time by doing these optimisations. For n=1024, the time required was reduced from 10 seconds to 6 seconds for optimisation 2. Similarly, the time required reduced a bit for optimisation 1 too when I changed some of the multiplications to additions.
For optimisation 3 to reduce the time required I declared all the 3 arrays globally.

For further optimisations, compiler's -O1, -O2, -O3 options were used. Values below are taken for n=1024 and s=64.

Diff. Optimizations	**Naive**	**Opt1**	**Opt2**	**Opt3**
Normal	24.96994	5.667843	6.097923	15.196035
-O1	12.169774	1.798506	2.070664	9.9605
-O2	8.093529	1.321937	1.641484	14.348536
-O3	7.550739	0.728919	0.931402	13.781909



Observations:

- Our optimisations performed worse than the theoretical expected values but better than the naive solutions.
- It was expected that optimisation 2 to be better than optimisation 1(because it has everything same as optimisation 1 plus the feature that the size of submatrix can now fit in the cache) but optimisation 2 always remained a bit more expensive than optimisation 1.
- Optimisation 3 was better than naive solution but performed badly than optimisations 1 and 2.
- Optimisation 3 performed a bit worse than naive solutions for small size of n probably because of heavy function calls which may increase a bit of time, but for large values of n optimisation 3 finally performs better than naive code.
- Between s=32 and s=64, it was observed that for small values of n, s=32 performed better than s=64 but for large values of n, we saw that s=64 significantly performed better than s=32.
- Compiler optimisations -O1, -O2, and -O3 all performed really well with naive, optimisation 1 and optimisation 2 codes. -O3 performed best among these 3 because of its -ftree-slp-vectorize flag. Because of this flag it does vector operations and hence speeding up the total process.
- Compiler optimisation -O1 performed best with optimisation 3 code.