Lab Report 1

This report talks about Matrix Multiplication and how it can be optimised by writing cache friendly code. We know that accessing data is the fastest at the registers followed by caches, followed by main memory and eventually followed by data storage.
In view of this, if developers write code which makes use of this fact, the algorithms become more computationally intensive and less time is spent on accessing memory and thereby increasing CPU's throughput and performance.

Simple Matrix Multiplication

```
void multiply(int** result, int **arr1, int**arr2, int size)
{
    for(int i=0; i< size; i++)
    {
        for(int j=0; j< size; j++)
        {
            for(int k=0; k<size; k++){
                result[i][j] += arr1[i][k] * arr2[k][j];
            }
        }
    }
}
```

We know that arrays are either stored row-wise or column-wise, but from the above calculation, we can see that one matrix is always going to fetch memory, since the previous memory fetch does not include the member to be accessed next. This causes a high number of cache misses. A solution to this issue in the simple matrix multiplication is Block Matrix Multiplication.

Block Matrix Multiplication :

The basic idea for cache misses is to optimise the algorithm so that data once fetched to cache is used again in the near time. To do so in matrix multiplication, we can divide the original matrix into blocks of size, let's say 'b'. Now, we can view these blocks as elements of a matrix and use them the same way to calculate simple matrix multiplication.

```
void blockMultiply(int**result, int**arr1, int**arr2, int size)
{
    int bsize = 10;
    for(int i=0; i<size; i+=bsize){
        for(int j=0; j<size; j+=bsize){
            for(int k=0; k<size; k+=bsize){
                //R[i][j] += A[i][k] * B[k][j]
                for(int a=i; a<i+bsize;a++){
                    for(int b=j;b<j+bsize;b++){
                        for(int c=0; c<bsize;c++){
                                result[a][b] += arr1[a][c+k]*arr2[c+k][b];
                        }
                    }
                }
            }
        }
    }

}
```

This way, our code becomes more computationally dense and uses cache in a much better way. The code below the commented line, is for the block matrix multiplication, whereas the code above the commented line depicts the iteration over blocks formed in the matrix.

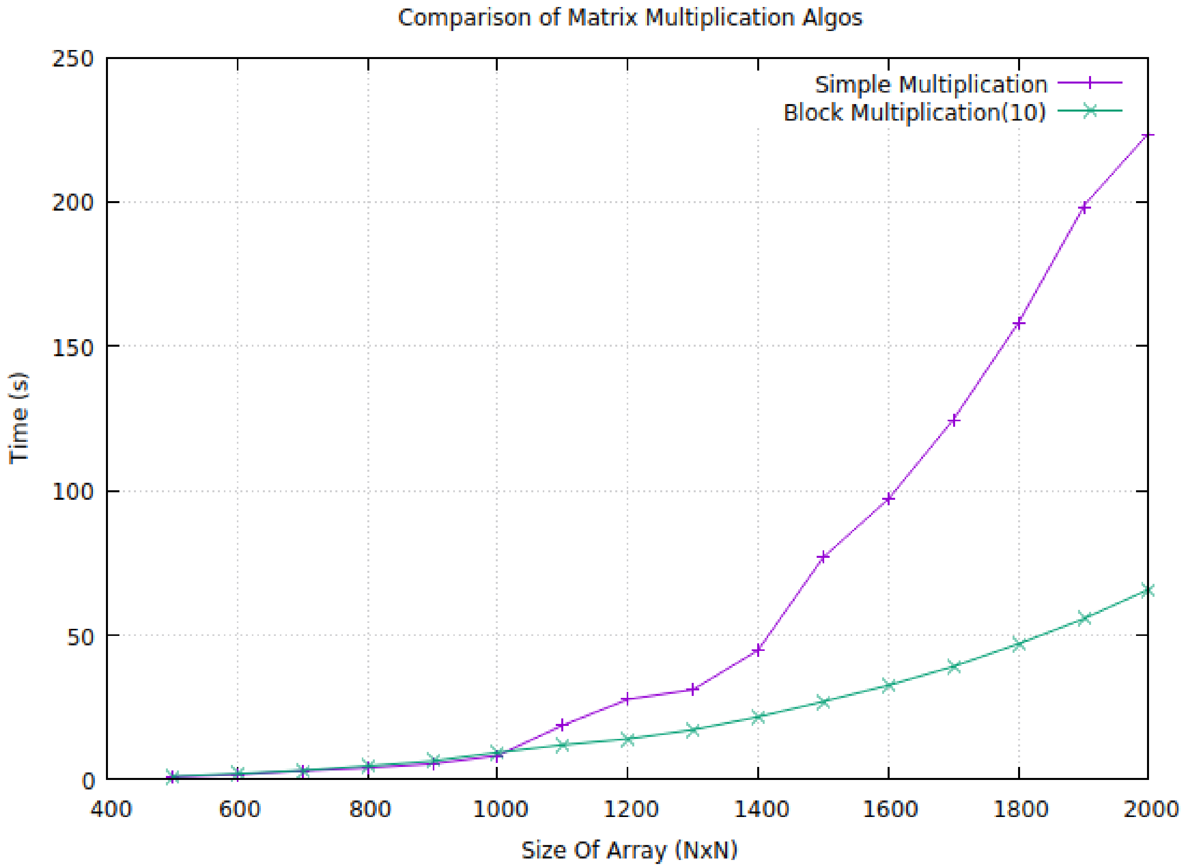
Observations & Results

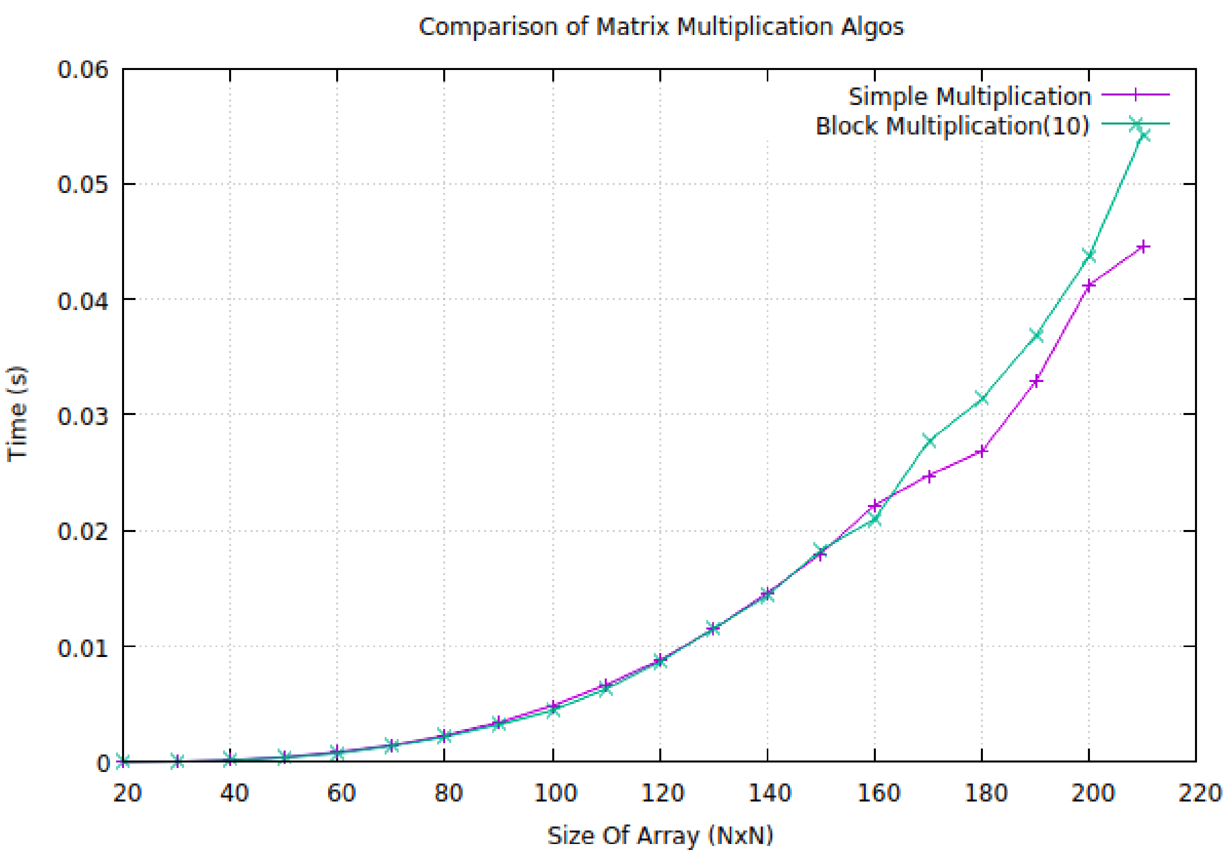
It is observed that for low values of N, there is not much difference in the time consumption typically till ~400x400 matrix size. This can be due to the fact that at small sizes, the cache is able to store much of the matrix, and thus in the block matrix multiplication, the code overhead becomes more than the simple matrix multiplication.

Now, if we increase the size of N, beyond 400 we find that the block matrix multiplication shows performance gain over the simple matrix multiplication. The curve for simple matrix multiplication sharply rises.

The observations were carried out with the following specifications
OS : Ubuntu-20.04
RAM : 8GB

The below graphs shows the curve for the time taken by Simple Matrix Multiplication in purple linen and Block Matrix Multiplication with block size 10, in green line.

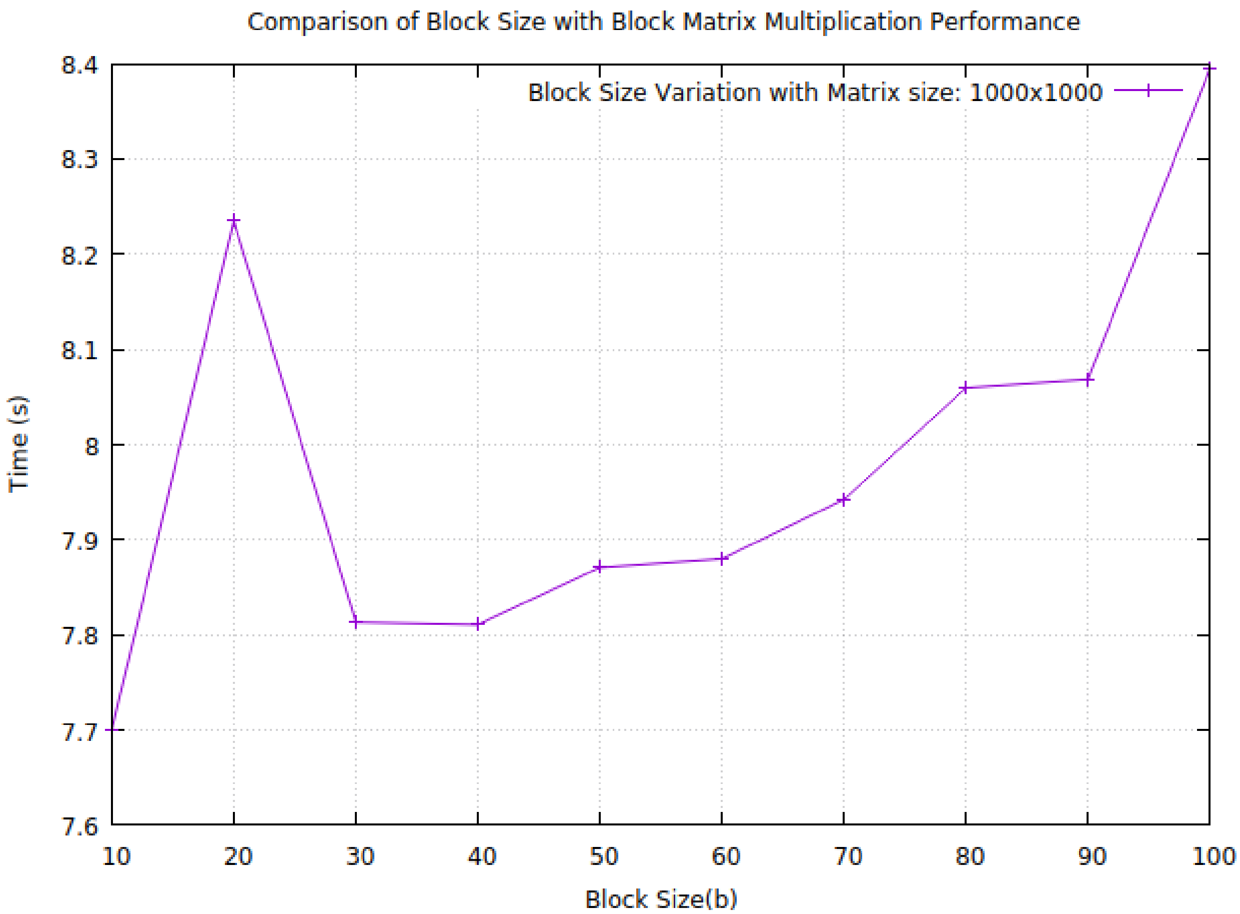




2021MCS2152

Hence, we see that as the size of N increases, we get a better performance from Block Matrix Multiplication algorithm, than the Simple Matrix Multiplication algorithm.

Now, for different block sizes, there can be slightly better or bad performance, due to fixed cache size in the hardware.



Data Observed

Matrix Size	SimpleMM Time(s)	BlockMM Time(s)
20	0.000015	0.000009
30	0.000098	0.000078
40	0.000197	0.000265
50	0.000437	0.000568
60	0.00086	0.001152
70	0.001472	0.001971
80	0.002312	0.003107

90	0.003428	0.004615
100	0.004901	0.006677
110	0.007075	0.009088
120	0.008787	0.012095
130	0.011694	0.015508
140	0.015806	0.019795
150	0.017415	0.025229
160	0.021713	0.030756
170	0.027261	0.037291
180	0.032798	0.044632
190	0.035279	0.053555
200	0.04434	0.062807
210	0.047847	0.073186
220	0.056028	0.084288
230	0.072266	0.096593
240	0.080003	0.110861
250	0.088771	0.126993
260	0.098913	0.144305
270	0.111752	0.161699
280	0.127571	0.17785
290	0.153234	0.199845
300	0.174068	0.224989
310	0.179947	0.245388
320	0.205642	0.269977
330	0.236774	0.302367
340	0.265875	0.331854
350	0.279528	0.359353
360	0.322448	0.390733
370	0.350888	0.428193
380	0.377722	0.462715
390	0.53921	0.484372
400	0.434396	0.511335

For Large Matrices.

Matrix Size	SimpleMM Time(s)	BlockMM Time(s)
500	0.9579	1.187306

600	1.741944	2.062523
700	3.010404	3.198004
800	4.018294	4.728603
900	5.518255	6.443341
1000	8.350665	9.414221
1100	18.811814	11.991683
1200	27.813933	14.075015
1300	31.168969	17.274848
1400	44.79207	21.703862
1500	76.901613	26.989357
1600	97.042783	32.648643
1700	124.59255	39.174858
1800	158.101966	46.945063
1900	198.160086	55.643703
2000	223.394378	65.859872

Block Size Variation Data
The matrix size was kept constant at 1000x1000.

Block Size	BlockMM Time(s)
10	7.699691
20	8.234739
30	7.813301
40	7.810884
50	7.870913
60	7.879908
70	7.942394
80	8.059627
90	8.068451
100	8.394962

References :

1. Lab Lecture 1 Notes.
2. Malith Jataweeras article on blocked matrix multiplication.
3. Wikipedia - https://en.wikipedia.org/wiki/Block_matrix