

SAHIL SHETYE
UIN: 673274841

ECE 466 Advance Computer Architecture
Project 2

1.

I am using Linux to run and compile my code. Compiler used in this project is gcc (version 4.8.2) compiler.

C function - rand () to automatically update values in each of the matrix elements. The size of the matrix is defined at the start of the program and is later on changed as per the requirements.

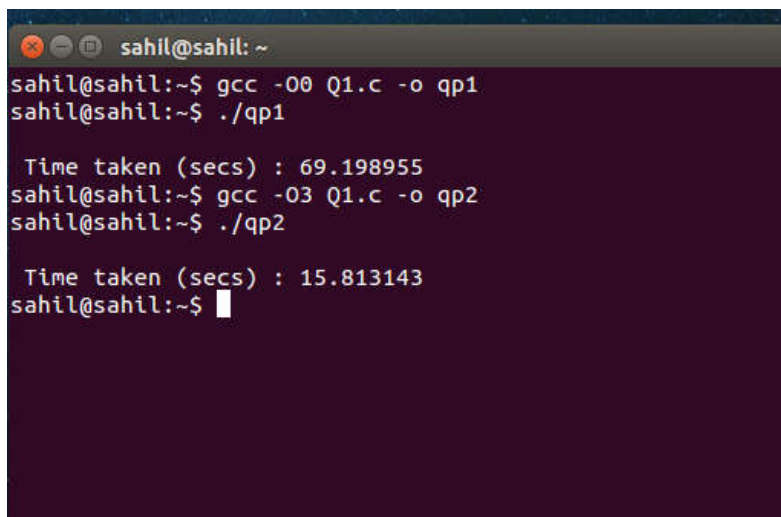
As per the instruction in the project 2 .pdf file, I have created the timing function timestamp which returns sends value w.r.t. to reference time. This time obtained in seconds is relative.

To compile the program, we write this command –

gcc compiles Q1.c by using no optimization and stores the compiled running program to q1

and to run the program –

./qp1



```
sahil@sahil: ~  
sahil@sahil:~$ gcc -O0 Q1.c -o qp1  
sahil@sahil:~$ ./qp1  
  
Time taken (secs) : 69.198955  
sahil@sahil:~$ gcc -O3 Q1.c -o qp2  
sahil@sahil:~$ ./qp2  
  
Time taken (secs) : 15.813143  
sahil@sahil:~$
```

Figure 1: Matrix Transpose with $N = 2048$ & -O3 optimization

Similarly

gcc -O3 Q1.c -o qp1

gcc compiles Q1.c by using standard optimization and stores the compiled running program to q1

and to run the program –

./qp1

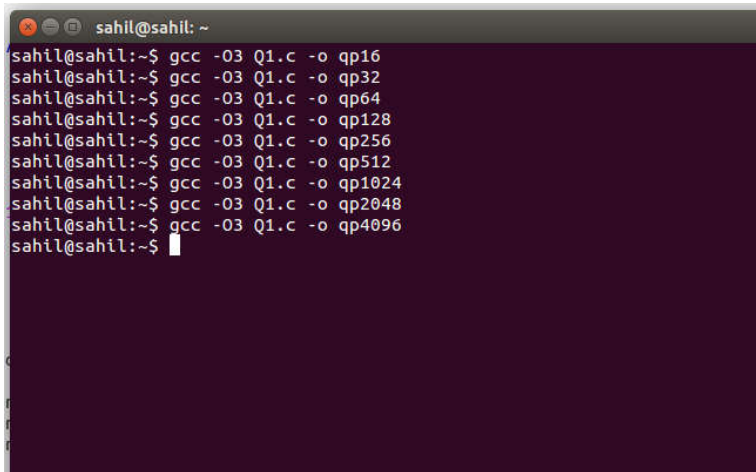
$$\text{Hence, the overall speedup} = \frac{\textit{Execeution time without optimization}}{\textit{Execeution time with optimization}} = \frac{69.19 \textit{ sec}}{15.81 \textit{ sec}}$$

$$= 4.37633 \text{ times}$$

Thus, as observed from the execution time, -O3 standard compiler optimization provides 4.3766 times speedup as compared to no compiler optimization.

2.

First we compile all the codes for array of different sizes varying from 16 to 4096. This is done by changing array size in each program and compiling them separately

A terminal window with a dark background and light text. The prompt is 'sahil@sahil: ~'. The user has entered a series of 'gcc' commands to compile programs for different array sizes. The commands are: 'gcc -O3 Q1.c -o qp16', 'gcc -O3 Q1.c -o qp32', 'gcc -O3 Q1.c -o qp64', 'gcc -O3 Q1.c -o qp128', 'gcc -O3 Q1.c -o qp256', 'gcc -O3 Q1.c -o qp512', 'gcc -O3 Q1.c -o qp1024', 'gcc -O3 Q1.c -o qp2048', and 'gcc -O3 Q1.c -o qp4096'. The cursor is at the end of the last command.

```
sahil@sahil: ~  
sahil@sahil:~$ gcc -O3 Q1.c -o qp16  
sahil@sahil:~$ gcc -O3 Q1.c -o qp32  
sahil@sahil:~$ gcc -O3 Q1.c -o qp64  
sahil@sahil:~$ gcc -O3 Q1.c -o qp128  
sahil@sahil:~$ gcc -O3 Q1.c -o qp256  
sahil@sahil:~$ gcc -O3 Q1.c -o qp512  
sahil@sahil:~$ gcc -O3 Q1.c -o qp1024  
sahil@sahil:~$ gcc -O3 Q1.c -o qp2048  
sahil@sahil:~$ gcc -O3 Q1.c -o qp4096  
sahil@sahil:~$
```

Figure 2: Compiling all the varying array sizes

Below is the result obtained by varying the size of the array from 16 to 4096 and its effect on execution time. The last column states the factor of execution time increase as compared with the previous execution time. It is clear that when the array size was increased from 512 to 1024, we observed the maximum factor increase in the execution time. In rest of the cases, the factor was very close to 4-5 times.

Array Size	Execution Time (in seconds)
16	0.029
32	0.033
64	0.035
128	0.038
256	0.080
512	0.48
1024	15.81
2048	110
4096	940

One can easily see that a steep rise in the execution time is seen at 1024 level where the value increased by 15.4 second. It is evident from the results that the first maximum increase in execution time was when the array size was 1024.

When the memory required for the execution of the program approaches the cache size, we see a sudden increase in the execution time. When the size of the program is lesser than the cache size, the factor is quite less in comparison. That is the reason why the factor remains relatively smaller in smaller array size. With the system that I am using, I am using 192 KB L1 cache and 4MB L2 cache. So 3 array of matrix will require 3×1024^2 space which is well beyond L1 cache hence first steep increase should be seen there since it also nears filling of L2 cache. if we increase the value of the input size, then it cannot store the value in cache and hence causes additional time to execute.

```
sahil@sahil: ~
sahil@sahil:~$ ./qp16
Time taken (secs) : 0.029
sahil@sahil:~$ ./qp32
Time taken (secs) : 0.033
sahil@sahil:~$ ./qp64
Time taken (secs) : 0.035
sahil@sahil:~$ ./qp128
Time taken (secs) : 0.038
sahil@sahil:~$ ./qp256
Time taken (secs) : 0.080
sahil@sahil:~$ ./qp512
Time taken (secs) : 0.48
sahil@sahil:~$ ./qp1024
Time taken (secs) : 15.81
sahil@sahil:~$ ./qp2048
Time taken (secs) : 110
sahil@sahil:~$ ./qp4096
Time taken (secs) : 940
sahil@sahil:~$
```

Figure 3: execution time from $N = 16$ to 4096

3. Implement the blocking method, set the blocking factor B to 8, and vary the array size as above. Compare the execution time of the two implementations with and without blocking. Based on the results, what's your suggestion on applying the blocking method?

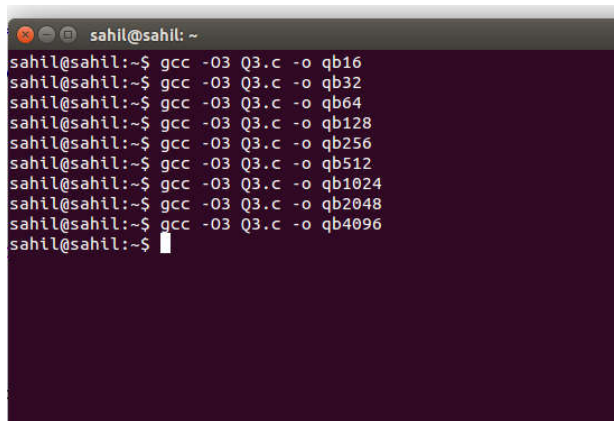
Array Size	Execution Time without Blocking (in seconds)	Execution Time with Blocking (in seconds)
16	0.029	0.045
32	0.033	0.046
64	0.035	0.040
128	0.038	0.030
256	0.080	0.040
512	0.48	0.22
1024	15.81	3
2048	110	29
4096	940	260

As observed from the table above, the execution time with blocking seems to be negligible but more than execution time of non-blocking until the value of matrix is 128. Later, as the value of the matrix increases, the execution time of the blocking execution decreases and drops lower than the execution time of the non-blocking execution method.

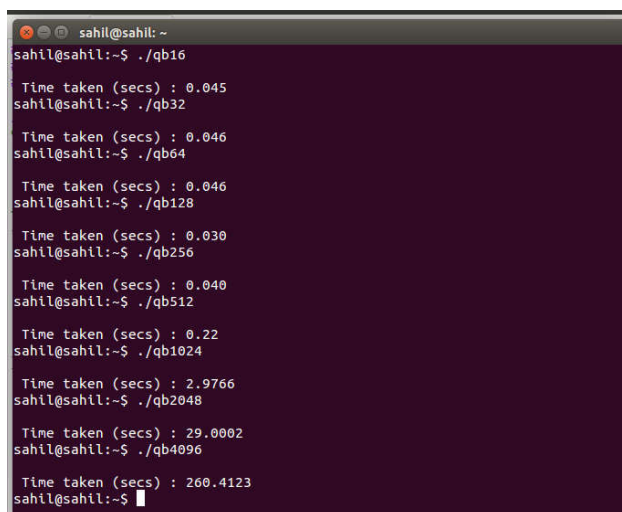
Thus, from the results we can conclude that the execution time with blocking decreases as compared to program execution time without blocking. Initially for small input size, the difference was extremely small and non-blocking execution time is less but as $N > 100$, the difference became more significant with blocking time becoming less than non-blocking time. Blocking method operates on blocks within the matrix and maximize accesses to data loaded into cache before they are replaced. The reason for initial large blocking execution time is for small array sizes, cache memory required is very small. Hence, performance enhancement due to blocking codes is not significant due to overhead. For larger array sizes,

blocking method increases the temporal locality and increases the performance of the program by reducing execution time. Thus better performance for large is obtained by blocking code.

Observation:



```
sahil@sahil: ~
sahil@sahil:~$ gcc -O3 Q3.c -o qb16
sahil@sahil:~$ gcc -O3 Q3.c -o qb32
sahil@sahil:~$ gcc -O3 Q3.c -o qb64
sahil@sahil:~$ gcc -O3 Q3.c -o qb128
sahil@sahil:~$ gcc -O3 Q3.c -o qb256
sahil@sahil:~$ gcc -O3 Q3.c -o qb512
sahil@sahil:~$ gcc -O3 Q3.c -o qb1024
sahil@sahil:~$ gcc -O3 Q3.c -o qb2048
sahil@sahil:~$ gcc -O3 Q3.c -o qb4096
sahil@sahil:~$
```



```
sahil@sahil: ~
sahil@sahil:~$ ./qb16
Time taken (secs) : 0.045
sahil@sahil:~$ ./qb32
Time taken (secs) : 0.046
sahil@sahil:~$ ./qb64
Time taken (secs) : 0.046
sahil@sahil:~$ ./qb128
Time taken (secs) : 0.030
sahil@sahil:~$ ./qb256
Time taken (secs) : 0.040
sahil@sahil:~$ ./qb512
Time taken (secs) : 0.22
sahil@sahil:~$ ./qb1024
Time taken (secs) : 2.9766
sahil@sahil:~$ ./qb2048
Time taken (secs) : 29.0002
sahil@sahil:~$ ./qb4096
Time taken (secs) : 260.4123
sahil@sahil:~$
```

Figure 4: Execution time with blocking and values of $N = 16$ to 4096

4.

Blocking factor	Execution Time (in seconds)
4	33
8	29
16	14
32	24
64	31
128	33
256	39
512	93

In this example, the input array size was kept constant at 2048 and the blocking factor was varied from $B = 4$ to 512 and the results were recorded. It was observed that for blocking factor of $B = 8$, we observed the most optimal solution with execution time of 29 seconds. After increasing the blocking factor, the execution time was increased. For various other values of the input, we can conclude that for the blocking factor of 16, Execution time is the least. And increases thereafter. Thus block size of 16 should give you optimum execution time.

Observation:

```
sahil@sahil: ~
sahil@sahil:~$ gcc -O3 Q4.c -o qc4
sahil@sahil:~$ gcc -O3 Q4.c -o qc8
sahil@sahil:~$ gcc -O3 Q4.c -o qc16
sahil@sahil:~$ gcc -O3 Q4.c -o qc32
sahil@sahil:~$ gcc -O3 Q4.c -o qc64
sahil@sahil:~$ gcc -O3 Q4.c -o qc128
sahil@sahil:~$ gcc -O3 Q4.c -o qc256
sahil@sahil:~$ gcc -O3 Q4.c -o qc512
sahil@sahil:~$
```

```
sahil@sahil: ~  
sahil@sahil:~$ ./qc4  
Time taken (secs) : 33.112  
sahil@sahil:~$ ./qc8  
Time taken (secs) : 29.002  
sahil@sahil:~$ ./qc16  
Time taken (secs) : 14.239  
sahil@sahil:~$ ./qc32  
Time taken (secs) : 23.899  
sahil@sahil:~$ ./qc64  
Time taken (secs) : 31.423  
sahil@sahil:~$ ./qc128  
Time taken (secs) : 33.433  
sahil@sahil:~$ ./qc256  
Time taken (secs) : 39.29  
sahil@sahil:~$ ./qc512  
Time taken (secs) : 92.978  
sahil@sahil:~$
```

Figure 5: Execution time for blocking factors from $B = 4$ to 512

5. What did you learn from this project?

This project gave me an insight of different types of compiler optimization techniques that can be used while executing larger programs. Most importantly, how compiler optimization plays a very important role in the programming. An optimization method can improve the performance by 4 times. Also, it is important to keep the running time of the algorithm into considerations when dealing with such larger input values. We often seem to neglect this fact, but when the input size is too large, it becomes increasingly difficult to keep the execution time under control even with optimization. Since even if code may be optimized if there are many nested loops, it'll create a problem in the computation time by introducing complexity.

Blocking techniques are a double-edged sword and they perform well only when many certain conditions are met. Cache size plays a very important role in deciding block size, maximum array size for optimum performance.

Source Code:

For Question 1 & 2

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
/* change dimension size as needed */
const int dimension = 2048;
struct timeval tv;
double timestamp()
{
    double t;
    gettimeofday(&tv, NULL);
    t = tv.tv_sec + (tv.tv_usec/1000000.0);
    return t;
}
int main(int argc, char *argv[])
{
    int i, j, k;
    double *A, *B, *C, start, end;
    A = (double*)malloc(dimension*dimension*sizeof(double));
    B = (double*)malloc(dimension*dimension*sizeof(double));
    C = (double*)malloc(dimension*dimension*sizeof(double));

    printf("\n Counting stated\n");
    start = timestamp();
    for(i = 0; i < dimension; i++)
        for(j = 0; j < dimension; j++)
            for(k = 0; k < dimension; k++)
                C[dimension*i+j] += 1.0;
    end = timestamp();
    printf("\n Time taken (secs) : %f\n", end-start);

    free(A);
    free(B);
    free(C);

    return 0;
}
```

For Question 3 & 4

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
```

```

/* change dimension size as needed */
const int dimension = 2048;
const int Bl= 8;
int main(int argc, char *argv[])
{

    int i, j, k, ii, jj, kk;
    double *A, *B, *C, start, end;

    A = (double*)malloc(dimension*dimension*sizeof(double));
    B = (double*)malloc(dimension*dimension*sizeof(double));
    C = (double*)malloc(dimension*dimension*sizeof(double));

    for(i = 0; i < dimension; i++)
        for(j = 0; j < dimension; j++)
        {
            A[dimension*i+j] = 1.0;
            B[dimension*i+j] = 1.0;
            C[dimension*i+j] = 0.0;
        }
    double timestamp()
    {
        double t;
        gettimeofday(&tv, NULL);
        t = tv.tv_sec + (tv.tv_usec/1000000.0);
        return t;
    }
    start = timestamp();

        for(i=0 ;i< dimension; i=i+Bl){

            for (j=0;j< dimension;j=j+Bl){

                for(k = 0; k < dimension; k=k+Bl){

                    for(ii =i ;ii < i+Bl; ii++){

                        for(jj = j; jj < j+Bl; jj++){
                            for(kk=k;kk<k+Bl;kk++){
                                C[dimension*ii+jj] += 1.0;
                            }
                        }
                    }
                }
            }
        }

    end = timestamp();

```

```
    printf("\n Time taken (secs):%f\n", end-start);  
    Free(A);  
    free(B);  
    free(C);  
    return 0;  
  
}
```