ISS Assignment1

By Aniruddha Bala
Roll No: 06-02-01-10-51-18-1-15655

September 28, 2018

1 Introduction to Scalable Systems - Assignment 1

1.1 Objective

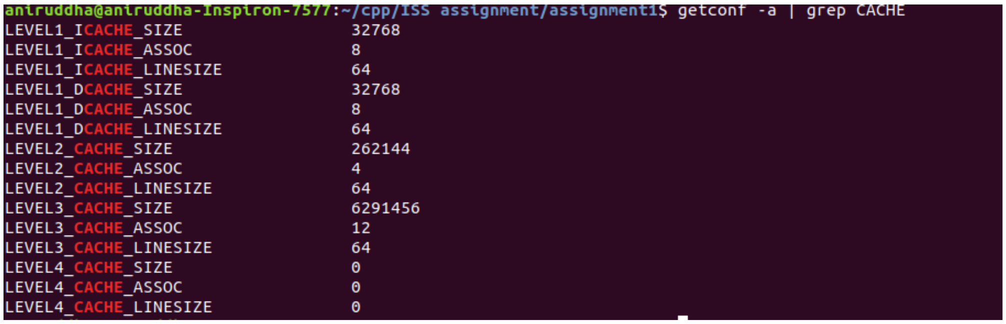
To optimize the naive algorithm for 2-D matrix multiplication using techniques such as loop interchange, blocking, vectorization etc and record the time taken for multiplying a 1024x1024 double matrices using different techniques.

1.2 Study of system hardware

```
aniruddha@aniruddha-Inspiron-7577:~/cpp/ISS assignment/assignment1$ lscpu
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              8
On-line CPU(s) list: 0-7
Thread(s) per core:  2
Core(s) per socket:  4
Socket(s):           1
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               158
Model name:          Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
Stepping:            9
CPU MHz:             800.082
CPU max MHz:         3800.0000
CPU min MHz:         800.0000
BogoMIPS:            5616.00
Virtualization:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            6144K
NUMA node0 CPU(s):   0-7
```

The system used for running the code has following specifications:
CPU: Intel Core i7 7700-hq processor
Clock speed: 2.8 GHz
Main memory: 8 GB
L1 data cache: 32 KB
L1 information cache: 32 KB
L2 cache: 256 KB
L3 cache: 6 MB

A deeper look into the cache configuration.



```
aniruddha@aniruddha-Inspiron-7577:~/cpp/ISS assignment/assignment1$ getconf -a | grep CACHE
LEVEL1_ICACHE_SIZE                 32768
LEVEL1_ICACHE_ASSOC                8
LEVEL1_ICACHE_LINESIZE             64
LEVEL1_DCACHE_SIZE                 32768
LEVEL1_DCACHE_ASSOC                8
LEVEL1_DCACHE_LINESIZE             64
LEVEL2_CACHE_SIZE                  262144
LEVEL2_CACHE_ASSOC                 4
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  6291456
LEVEL3_CACHE_ASSOC                 12
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC                 0
LEVEL4_CACHE_LINESIZE              0
```

Cache configuration

We can see from the above output the specifications of each cache block in detail. Let's consider the L3 cache since it is the one that interacts with the main memory. The L3 cache is total 6 MB in size. It has a line size of 64 bytes and is 12 way set associative. Therefore we can calculate the number of sets from this.

$$\text{Number of sets} = \frac{\text{Total cache size in (B)}}{\text{(Number of lines in a set)*block size in (B)}} = \frac{6 * 2^{20}}{12 * 2^{6}} = 8192$$

1.3 Experiments

Please refer file Matmul.cpp or appendix section for the code. In this file I have implemented 6 different versions of matrix multiplication.

- Version 1: The function matmul_v1 takes two input matrices a and b , an output matrix c and the size of the square matrix as an input. The function implements the matrix multiplication code without any cache level optimization

- Version 2: The function matmul_v2 takes two input matrices a and b , an output matrix c and the size of the square matrix as an input. The function implements the matrix multiplication code exploiting the temporal locality of reference by replacing c[i][j] with a temp variable that can be stored in register file for faster access.

- Version 3: The function matmul_v3 takes two input matrices a and b , an output matrix c and the size of the square matrix as an input. The function implements the matrix multiplication code using loop interchange. Here the loops j and k have been interchanged.

- Version 4: The function matmul_v4 takes two input matrices a and b , an output matrix c and the size of the square matrix as an input. The function implements the matrix multiplication code using loop interchange and loop unrolling done for each of the i, j and k loops.

- Version 5: The function matmul_v5 takes two input matrices a and b , an output matrix c, the size of the square matrix and the block size as an input. The function implements the matrix multiplication code using blocking with loop interchange.

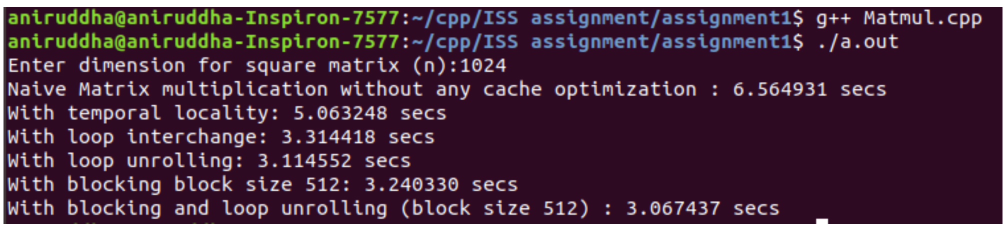
- Version 6: The function matmul_v6 takes two input matrices a and b , an output matrix c, the size of the square matrix and the block size as an input. The function implements the matrix multiplication code using blocking with loop interchange and loop unrolling.

* Please note that the functions for version 4 and 6 assume even n value. This assumption has been made in case of loop unrolling for simplicity as adding extra instructions to compute the value for the remaining elements unnecessarily adds to the computational cost. For n odd functions 4 and 6 raise an assertion error.

This experiment has been done for n=1024 by initializing the input 2-D arrays a and b with random numbers of type double. On running the code with various compilation schemes for an input size of 1024x1024 we get the following outputs. The outputs denote the time taken by each of the above 6 implementations.

1.4 Observation

1.4.1 i) With normal compilation

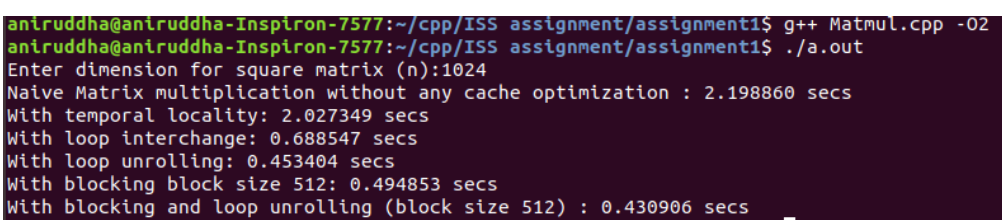


```
aniruddha@aniruddha-Inspiron-7577:~/cpp/ISS assignment/assignment1$ g++ Matmul.cpp
aniruddha@aniruddha-Inspiron-7577:~/cpp/ISS assignment/assignment1$ ./a.out
Enter dimension for square matrix (n):1024
Naive Matrix multiplication without any cache optimization : 6.564931 secs
With temporal locality: 5.063248 secs
With loop interchange: 3.314418 secs
With loop unrolling: 3.114552 secs
With blocking block size 512: 3.240330 secs
With blocking and loop unrolling (block size 512) : 3.067437 secs
```

Outputs for normal compilation

From the above outputs we can see the gradual improvements in run time as we introduce cache level optimization. Even just with temporal locality of reference we can see the marginal improvement in performance of the code. The reason being introduction of variable temp to store the result of the sum. This reduces the memory accesses to c[i][j] every time as the sum value now gets stored in a register file which can be accessed faster. Loop interchange results in a significant improvement of performance. This is because now the arrays a and b are getting accessed row-wise which results in increase of overall hit rate. Unrolling the i,j, k loops further improve the run time. As the loops have been unrolled the for loops now run for half the number of iterations as before thus the loop related costs are reduced though the number of loop instructions per loop has increased. Blocking has been implemented in two parts one without loop unrolling and one with loop unrolling. Block size of 512 is found to work the best (I have shown the experiment results for this below). Blocking with unrolling works the best among all the 6 methods. Blocking helps to utilize the cache to the fullest and loop unrolling further helps by reducing the loop control and test instructions.

1.4.2 ii) With optimization level 2

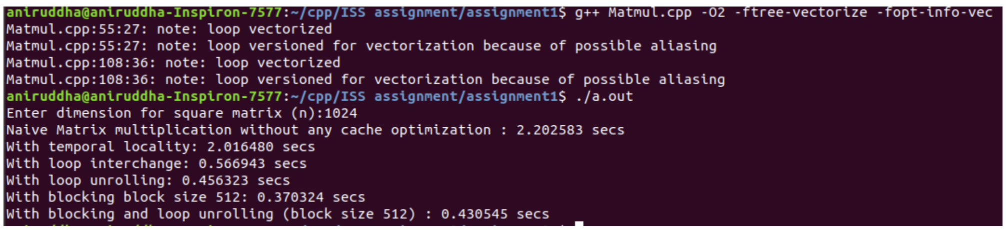


```
aniruddha@aniruddha-Inspiron-7577:~/cpp/ISS assignment/assignment1$ g++ Matmul.cpp -O2
aniruddha@aniruddha-Inspiron-7577:~/cpp/ISS assignment/assignment1$ ./a.out
Enter dimension for square matrix (n):1024
Naive Matrix multiplication without any cache optimization : 2.198860 secs
With temporal locality: 2.027349 secs
With loop interchange: 0.688547 secs
With loop unrolling: 0.453404 secs
With blocking block size 512: 0.494853 secs
With blocking and loop unrolling (block size 512) : 0.430906 secs
```

Outputs for compilation with optimization level 2 (O2)

Using just the -O2 flag while compilation reduces the run times for each of the 6 methods drastically. But the trend is similar to the previous case i.e. in this also blocking with loop unrolling works the best.

1.4.3 iii) With vectorization

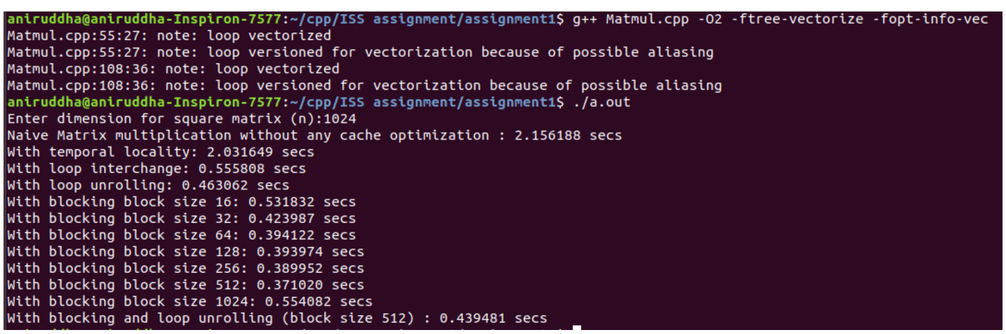

```
aniruddha@aniruddha-Inspiron-7577:~/cpp/ISS assignment/assignment1$ g++ Matmul.cpp -O2 -ftree-vectorize -fopt-info-vec
Matmul.cpp:55:27: note: loop vectorized
Matmul.cpp:55:27: note: loop versioned for vectorization because of possible aliasing
Matmul.cpp:108:36: note: loop vectorized
Matmul.cpp:108:36: note: loop versioned for vectorization because of possible aliasing
aniruddha@aniruddha-Inspiron-7577:~/cpp/ISS assignment/assignment1$ ./a.out
Enter dimension for square matrix (n):1024
Naive Matrix multiplication without any cache optimization : 2.202583 secs
With temporal locality: 2.016480 secs
With loop interchange: 0.566943 secs
With loop unrolling: 0.456323 secs
With blocking block size 512: 0.370324 secs
With blocking and loop unrolling (block size 512) : 0.430545 secs
```

Outputs for compilation with vectorization

When vectorization is used as an option while compilation the inner loop for the functions with loop interchange (version 3) and block multiplication (version 5) is vectorized. Here block multiplication outperforms block multiplication with loop unrolling because inner loop of the prior gets vectorized whereas the inner loop in case of block multiplication with loop unrolling is not vectorizable as loop counter gets incremented by 2 in each iteration and array access is not sequential over the iterations.

1.4.4 Choice of block size

The blocking code was run for different block sizes starting from 16 in powers of 2 till 1024. The minimum time was observed for block size = 512. The below output shows the runtimes for different block sizes.


```
aniruddha@aniruddha-Inspiron-7577:~/cpp/ISS assignment/assignment1$ g++ Matmul.cpp -O2 -ftree-vectorize -fopt-info-vec
Matmul.cpp:55:27: note: loop vectorized
Matmul.cpp:55:27: note: loop versioned for vectorization because of possible aliasing
Matmul.cpp:108:36: note: loop vectorized
Matmul.cpp:108:36: note: loop versioned for vectorization because of possible aliasing
aniruddha@aniruddha-Inspiron-7577:~/cpp/ISS assignment/assignment1$ ./a.out
Enter dimension for square matrix (n):1024
Naive Matrix multiplication without any cache optimization : 2.156188 secs
With temporal locality: 2.031649 secs
With loop interchange: 0.555808 secs
With loop unrolling: 0.463062 secs
With blocking block size 16: 0.531832 secs
With blocking block size 32: 0.423987 secs
With blocking block size 64: 0.394122 secs
With blocking block size 128: 0.393974 secs
With blocking block size 256: 0.389952 secs
With blocking block size 512: 0.371020 secs
With blocking block size 1024: 0.554082 secs
With blocking and loop unrolling (block size 512) : 0.439481 secs
```

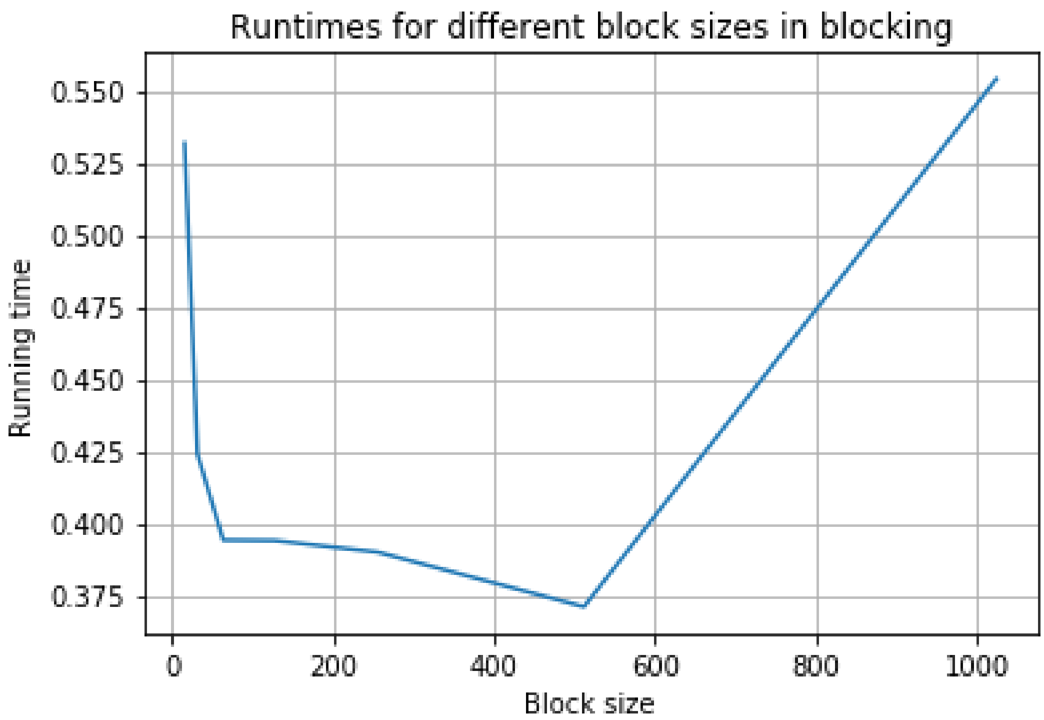
Running times for different block sizes

We expect such an output because for block size of 512 the two arrays a and b when accessed in blocks will have total block size of $2 * (512 * 512 * 8)$ bytes which is 4 MB and the L3 cache size is 6MB and therefore they fit in cache. Any size lesser than this will be under utilizing the cache. And any size greater than this such as 1024 will occupy $2 * (1024 * 1024 * 8)$ bytes i.e. 16 MB which is way more than the cache size and hence there will be lots of conflicts.

```
In [1]: import matplotlib.pyplot as plt
        %matplotlib inline
        plt.plot([2**i for i in range(4,11) ],[0.531832, 0.423987, 0.394122, 0.393974,  0.3899
```

```
plt.grid()
plt.xlabel('Block size')
plt.ylabel('Running time')
plt.title('Runtimes for different block sizes in blocking')
```

Out[1]: Text(0.5,1,'Runtimes for different block sizes in blocking')



At last as a sanity check I will printout the results obtained by different methods to check if they are consistent for a smaller 4x4 matrix.

```
aniruddha@aniruddha-Inspiron-7577:~/cpp/ISS assignment/assignment1$ g++ Matmul.cpp -O2 -ftree-vectorize -fopt-info-vec
Matmul.cpp:55:27: note: loop vectorized
Matmul.cpp:55:27: note: loop versioned for vectorization because of possible aliasing
Matmul.cpp:108:36: note: loop vectorized
Matmul.cpp:108:36: note: loop versioned for vectorization because of possible aliasing
aniruddha@aniruddha-Inspiron-7577:~/cpp/ISS assignment/assignment1$ ./a.out
Enter dimension for square matrix (n):4
93.830000 8.860000 27.770000 69.150000
77.930000 83.350000 53.860000 4.920000
66.490000 14.210000 23.620000 0.270000
86.900000 0.590000 77.630000 39.260000
5.400000 34.260000 91.720000 57.360000
52.110000 53.680000 25.670000 64.290000
57.820000 15.300000 28.620000 51.230000
40.670000 31.350000 39.290000 98.020000
Naive Matrix multiplication without any cache optimization : 0.000002 secs
5386.368500 6282.954100 12345.204700 14152.438300
8078.472100 8122.409800 11022.114100 13070.142500
2476.218400 3410.590700 7149.846200 5963.945300
6585.275700 5427.405200 11749.909300 12847.765200
With temporal locality: 0.000005 secs
5386.368500 6282.954100 12345.204700 14152.438300
8078.472100 8122.409800 11022.114100 13070.142500
2476.218400 3410.590700 7149.846200 5963.945300
6585.275700 5427.405200 11749.909300 12847.765200
With loop interchange: 0.000004 secs
5386.368500 6282.954100 12345.204700 14152.438300
8078.472100 8122.409800 11022.114100 13070.142500
2476.218400 3410.590700 7149.846200 5963.945300
6585.275700 5427.405200 11749.909300 12847.765200
With loop unrolling: 0.000003 secs
5386.368500 6282.954100 12345.204700 14152.438300
8078.472100 8122.409800 11022.114100 13070.142500
2476.218400 3410.590700 7149.846200 5963.945300
6585.275700 5427.405200 11749.909300 12847.765200
With blocking block size 2: 0.000004 secs
5386.368500 6282.954100 12345.204700 14152.438300
8078.472100 8122.409800 11022.114100 13070.142500
2476.218400 3410.590700 7149.846200 5963.945300
6585.275700 5427.405200 11749.909300 12847.765200
With blocking and loop unrolling (block size 2) : 0.000004 secs
5386.368500 6282.954100 12345.204700 14152.438300
8078.472100 8122.409800 11022.114100 13070.142500
2476.218400 3410.590700 7149.846200 5963.945300
6585.275700 5427.405200 11749.909300 12847.765200
```

Matrix multiplication outputs for different methods for 4x4 matrix

In the output first 4 rows are of matrix A and the next 4 of matrix B. Thus we can see from the results that the matrix product obtained with all 6 methods are consistent and match with the naive matrix multiplication algorithm which was verified for correctness.

1.5 Appendix

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<string.h>
#include<algorithm>
#include<assert.h>
using namespace std;

#define MAXN 2049
#define MAXV 1000
#define OPT_BLCK_SZ 512

double A[MAXN][MAXN],B[MAXN][MAXN], C[MAXN][MAXN];

struct timespec start, finish;

//without any cache optimization
```

```
void matmul_v1(double a[][MAXN], double b[][MAXN], double c[][MAXN], int n){
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &start);
    for(int i=0; i<n; i++){
        for(int j=0; j<n; j++){
            for(int k=0; k<n; k++){
                c[i][j] += a[i][k]*b[k][j];
            }
        }
    }
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &finish);
    double run_time = (finish.tv_sec - start.tv_sec) + (finish.tv_nsec - start.
        ↪ tv_nsec)/1e9;
    printf("Naive Matrix multiplication without any cache optimization : %lf secs\
        ↪ n",run_time);
}


// exploiting temporal locality
void matmul_v2(double a[][MAXN], double b[][MAXN], double c[][MAXN], int n){
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &start);
    for(int i=0; i<n; i++){
        for(int j=0; j<n; j++){
            double sum = 0;
            for(int k=0; k<n; k++){
                sum += a[i][k]*b[k][j];
            }
            c[i][j] = sum;
        }
    }
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &finish);
    double run_time = (finish.tv_sec - start.tv_sec) + (finish.tv_nsec - start.
        ↪ tv_nsec)/1e9;
    printf("With temporal locality: %lf secs\n",run_time);
}

// with loop interchange
void matmul_v3(double a[][MAXN], double b[][MAXN], double c[][MAXN], int n){
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &start);
    for(int i=0; i<n; i++){
        for(int k=0; k<n; k++){
            double x = a[i][k];
            for(int j=0; j<n; j++){
                c[i][j] += x*b[k][j];
            }
        }
    }
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &finish);
```

```
    double run_time = (finish.tv_sec - start.tv_sec) + (finish.tv_nsec - start.
        ↪ tv_nsec)/1e9;
    printf("With loop interchange: %lf secs\n",run_time);
}

//with loop unrolling
void matmul_v4(double a[][MAXN], double b[][MAXN], double c[][MAXN], int n){

    assert(n%2==0);
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &start);
    for(int i=0; i<n; i+=2){
        for(int k=0; k<n; k+=2){
            double x = a[i][k];
            double y = a[i][k+1];
            double u = a[i+1][k];
            double v = a[i+1][k+1];
            for(int j=0; j<n; j+=2){
                c[i][j] += x*b[k][j];
                c[i][j+1] += x*b[k][j+1];
                c[i][j] += y*b[k+1][j];
                c[i][j+1] += y*b[k+1][j+1];

                c[i+1][j] += u*b[k][j];
                c[i+1][j+1] += u*b[k][j+1];
                c[i+1][j] += v*b[k+1][j];
                c[i+1][j+1] += v*b[k+1][j+1];

            }

        }
    }
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &finish);
    double run_time = (finish.tv_sec - start.tv_sec) + (finish.tv_nsec - start.
        ↪ tv_nsec)/1e9;
    printf("With loop unrolling: %lf secs\n",run_time);
}


//using block multiplication
void matmul_v5(double a[][MAXN], double b[][MAXN], double c[][MAXN], int n, int
    ↪ blck_sz){

    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &start);
    for(int bk=0; bk<n; bk+=blck_sz){
        for(int bj=0; bj<n; bj+=blck_sz){
            int to_k = min(bk+blck_sz,n);
            int to_j = min(bj+blck_sz,n);
            for(int i=0; i<n; i++){
```

```
                for(int k=bk; k<to_k; k++){
                    double x = a[i][k];
                    for(int j=bj; j<to_j; j++)
                        c[i][j]+=x*b[k][j];
                }
            }
        }
    }
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &finish);
    double run_time = (finish.tv_sec - start.tv_sec) + (finish.tv_nsec - start.
        ↪ tv_nsec)/1e9;
    printf("With blocking block size %d: %lf secs\n",blck_sz,run_time);
}

//using block multiplication with loop unrolling
void matmul_v6(double a[][MAXN], double b[][MAXN], double c[][MAXN], int n, int
    ↪ blck_sz){

    assert(n%2==0);
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &start);
    for(int bk=0; bk<n; bk+=blck_sz){
        for(int bj=0; bj<n; bj+=blck_sz){
            int to_k = min(bk+blck_sz,n);
            int to_j = min(bj+blck_sz,n);
            for(int i=0; i<n; i+=2){
                for(int k=bk; k<to_k; k+=2){
                    double x = a[i][k];
                    double y = a[i][k+1];
                    double u = a[i+1][k];
                    double v = a[i+1][k+1];
                    for(int j=bj; j<to_j; j+=2)
                    {
                        c[i][j] += x*b[k][j];
                        c[i][j+1] += x*b[k][j+1];
                        c[i][j] += y*b[k+1][j];
                        c[i][j+1] += y*b[k+1][j+1];

                        c[i+1][j] += u*b[k][j];
                        c[i+1][j+1] += u*b[k][j+1];
                        c[i+1][j] += v*b[k+1][j];
                        c[i+1][j+1] += v*b[k+1][j+1];
                    }

                }
            }
        }
    }
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &finish);
```

```
    double run_time = (finish.tv_sec - start.tv_sec) + (finish.tv_nsec - start.tv_nsec)/1e9;
    printf("With blocking and loop unrolling (block size %d) : %lf secs\n",blck_sz,run_time);
}
void read_mat(double a[][MAXN], int n){
    for(int i=0; i<n; i++){
        for(int j=0; j<n; j++){
            int x = scanf("%lf", &a[i][j]);
        }
    }
}

void rand_mat(double a[][MAXN], int n){
    for(int i=0; i<n; i++){
        for(int j=0; j<n; j++){
            a[i][j] = (rand()%10000)/100.0;
        }
    }
}
void print_mat(double a[][MAXN], int n){
    for(int i=0; i<n; i++){
        for(int j=0; j<n; j++){
            printf("%lf ", a[i][j]);
        }
        printf("\n");
    }
}

int main(){
    int n;
    printf("Enter dimension for square matrix (n):");
    int x = scanf("%d",&n);
    rand_mat(A,n);
    rand_mat(B,n);
    memset(C, 0.0, sizeof(C));
    matmul_v1(A,B,C,n);
    memset(C, 0.0, sizeof(C));
    matmul_v2(A,B,C,n);
    memset(C, 0.0, sizeof(C));
    matmul_v3(A,B,C,n);
    memset(C, 0.0, sizeof(C));
    matmul_v4(A,B,C,n);
    memset(C, 0.0, sizeof(C));
    matmul_v5(A,B,C,n,OPT_BLCK_SZ);
    memset(C, 0.0, sizeof(C));
    matmul_v6(A,B,C,n,OPT_BLCK_SZ);
}
```