CS205 - C/C++ Program Design

Fitria Zusni Farida (12112351)

Part I – Analysis

The problem asks to implement general matrix multiplication in C or C++ with high efficiency. The definition of GEMM in a BLAS library is as follow:

$$C \leftarrow \alpha AB + \beta C$$

Where α and β are scalar, A, B, and C are nxn double matrices.

The matrices are 2-dimentional matrix which are mapped into 1-dimentional array of double. The problem also asks to compare the implementation with cblas_dgemm() in OpenBLAS library. Since there might be several unknown implementations which one has high efficiency, some implementations were experimented and compared to find which implementation may be able to achieve. There are many libraries which are supported in my CPU, such as:

```
   CPU family:          6
   Model:               186
   Thread(s) per core:  2
   Core(s) per socket:  8
   Socket(s):           1
   Stepping:            2
   BogoMIPS:            6374.42
   Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse ss
                        e2 ss ht syscall nx pdpe1gb rdtscp lm constant_tsc rep_good nopl xtopology tsc_reliable nonstop
                        _tsc cpuid pni pclmulqdq vmx ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline
                        _timer aes xsave avx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch invpcid_single ssbd ibrs
                        ibpb stibp ibrs_enhanced tpr_shadow vnmi ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2 smep bmi
                        2 erms invpcid rdseed adx smap clflushopt clwb sha_ni xsaveopt xsavec xgetbv1 xsaves avx_vnni u
                        mip waitpkg gfni vaes vpclmulqdq rdpid movdiri movdir64b fsrm serialize flush_l1d arch_capabili
                        ties
Virtualization features:
 Virtualization:        VT-x
 Hypervisor vendor:     Microsoft
 Virtualization type:   full
```

But here only several are being implemented for some reasons.

The elements of matrix A, B, and C are randomly generated. The time execution is measured using ‘chrono’ standard library and put right before and after the GEMM operation function. To guarantee that all implementations are operated correctly and have the same result, the matrix A, B, and C are firstly assigned fixed double numbers.

```
const int n = 3;  // Matrix size
double alpha = 1.0;
double beta = 1.0;
int size = n * n;

double *A = new double[n*n]{1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0};
double *B = new double[n*n]{1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0};
double *C = new double[n*n]{1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0};
```

If all the implementation resulting correct matrix C that is as follow, it is assumed that the operation is correct.

31	38	45
70	86	102
109	134	159

Or mapped into 1D array becomes {31,38,45,70,86,102,109,134,159}.

In this project, there are several implementations that is experimented and compared according to the execution times. The implementations are as follow:

1) Using OpenBLAS Library
   Provides standard routines for performing linear algebra operations, such as matrix multiplication, vector operations, and matrix factorization. It is claimed for highly efficient implementation of these routines for various architecture including CPUs and GPUs, to obtain faster computation performance. It optimizes the use of hardware specific features, such as vector instructions and parallelization techniques, to maximize computational efficiency.
2) Naïve multiplication
   Basic and straightforward method of multiplication. It might be inefficient especially for large matrices because it uses iteration over the element of the matrices. It will be also compared here that the time execution of this method is far compared with OpenBLAS's.
3) Using Eigen Library
   This library has wide range of linear algebra functionality. It easy and user-friendly to be used.
4) Using naïve & OpenMP Library
   The naïve method may be not comparable with OpenBLAS, but additional implementation which combine with another method might be able to give more precise data to be analyzed.
5) Using m-block matrix multiplication
   Technique that optimizes the computation of matrix multiplication by dividing the matrices into smaller blocks and performing the multiplication on these blocks. The idea is to reduce memory access and improve data locality. By processing smaller blocks, the algorithm can fit the blocks into cache memory, minimizing the need to access data from the main memory. It will also observe how the block size m impact the performance.

6) OpenMP and m-block matrix multiplication
   Combining with other implementation will be also used to get higher performance.

Part II – Code

1. Using OpenBLAS library

main.cpp > ...

```cpp
#include <chrono>
#include <cblas.h>
#include <iostream>
using namespace std::chrono;

int main() {
    int n ;
    std::cin >> n ;
    double alpha = 1.0 ;
    double beta = 1.0  ;

    double * A = new double[n*n];
    double * B = new double[n*n];
    double * C = new double[n*n];
    // Fill A, B, and C with random values

    for (int i = 0; i < n * n; ++i) {
        A[i] = (double)rand() / RAND_MAX;
        B[i] = (double)rand() / RAND_MAX;
        C[i] = (double)rand() / RAND_MAX;
    }

    auto start_time = high_resolution_clock::now();

    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, n, n, n, alpha, A, n, B, n, beta, C, n);

    auto end_time = high_resolution_clock::now();

    auto total_time = duration_cast<nanoseconds>(end_time-start_time);

    std::cout << "n = " << n << std::endl;
    std::cout<< "Time needed for OpenBLAS : " << total_time.count() << "ns" << std::endl;

    for(int i=0 ; i<n*n ; i++){
        std::cout<<C[i] << " " ;
    }

    delete[] A;
    delete[] B;
```

2. Naïve multiplication

```cpp
#include <iostream>
#include <random>
#include <chrono>

void matrixOperation(double* A, double* B, double* C, const int n, const double alpha, const double beta)
{

    for (int i = 0; i < n; ++i)
    {
        for (int j = 0; j < n; ++j)
        {
            // Calculate the index for element (i, j)
            const int index = i * n + j;

            // Perform the computation
            C[index] = alpha * A[index] * B[index] + beta * C[index];
        }
    }
}
```

~ outer loop iterates over rows, inner loop iterates over columns.

```cpp
int main()
{
    const int n=3;
    const double alpha = 1.0;
    const double beta = 1.0;

    const int size =  n*n ;
    double* A = new double[size];
    double* B = new double[size];
    double* C = new double[size];

        for (int i = 0; i < n * n; ++i) {
            A[i] = (double)rand() / RAND_MAX;
            B[i] = (double)rand() / RAND_MAX;
            C[i] = (double)rand() / RAND_MAX;
        }
    auto start = std::chrono::high_resolution_clock::now();

    matrixOperation(A, B, C, n, alpha, beta);

    auto end = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end - start).count();

    std::cout << "Execution time: " << duration << " ns" << std::endl;

    for(int i=0; i<size ; i++){
        std::cout << C[i] << " " ;
    }

    // Deallocate memory
    delete[] A;
    delete[] B;
    delete[] C;

    return 0;
}
```

3. Using Eigen Library

Project5 > eig.cpp > main()

```cpp
#include <iostream>
#include <Eigen/Dense>
#include <chrono>
using namespace std::chrono;

int main() {
    int n ;
    std::cin >> n ;
    double alpha = 1.0;
    double beta = 1.0;

    Eigen::MatrixXd A(n, n);
    Eigen::MatrixXd B(n, n);
    Eigen::MatrixXd C(n, n);

    A.setRandom();
    B.setRandom();
    C.setRandom();

    auto start = high_resolution_clock::now();

    C = alpha * A * B + beta * C;

    std::cout << "n = " << n << std::endl;
    auto finish = high_resolution_clock::now();

    auto total_time = duration_cast<nanoseconds>(finish-start);

    std::cout<<"Time needed using eigen library: " << total_time.count() << "ns" << std::endl ;

    std::cout << C << std::endl;


    return 0;
}
```

Many basic functions are provided by the library by “#include <Eigen/Dense> such as: setRandom().

4. Using m-block matrix multiplication

```cpp
#include <iostream>
#include <algorithm>
#include <chrono>
using namespace std::chrono;

void blockMatrixMultiplication(double* A, double* B, double* C, int n, double alpha, double beta) {
    // Define block size -> change as we want
    int blockSize = 64;
    // Perform block matrix multiplication
    for (int i = 0; i < n; i += blockSize) {
        for (int j = 0; j < n; j += blockSize) {
            for (int k = 0; k < n; k += blockSize) {
                // Compute block matrix multiplication within the block
                int blockRowSize = std::min(blockSize, n - i);
                int blockColSize = std::min(blockSize, n - j);
                int blockKSize = std::min(blockSize, n - k);
                for (int ii = 0; ii < blockRowSize; ii++) {
                    for (int jj = 0; jj < blockColSize; jj++) {
                        double sum = 0.0;
                        for (int kk = 0; kk < blockKSize; kk++) {
                            sum += A[(i + ii) * n + (k + kk)] * B[(k + kk) * n + (j + jj)];
                        }
                        C[(i + ii) * n + (j + jj)] = alpha * sum + beta * C[(i + ii) * n + (j + jj)];
                    }
                }
            }
        }
    }
}
```

~ blockSize determines the size of blocks used for the block matrix multiplication.

~ outermost loop: iterates over rows(i) of matrix C, second loop iterates over columns(j) of matrix C, and the innermost loop iterates over the columns(k) or matrix A and the rows(k) of matrix B.

~ Within the innermost loop, block matrix multiplication is performed within each block. The size is determined by blockRowSize, blockColSIze, and blockSize which ensure that the block size do not exceeds the matrix dimensions.

~ Within the block, another pair of nested for loops iterates over the elements within the block. ‘ii’ represents the row index within the block, ‘jj’ represents the column index within the block.

~ Another for loop iterates over the column(k) within the block, calculating the partial sum of the products of corresponding elements of the matrices A and B.

~ Then finally multiplied by alpha and beta,

```cpp
int main(){

    int n ;
    std::cin>>n;
    std::cout<<"n = " << n << std::endl;
    double alpha = 1.0;
    double beta = 1.0;


    double* A = new double[n*n];
    double* B = new double[n*n];
    double* C = new double[n*n];


    for (int i = 0; i < n * n; ++i) {
            A[i] = (double)rand() / RAND_MAX;
            B[i] = (double)rand() / RAND_MAX;
            C[i] = (double)rand() / RAND_MAX;
    }


auto start = std::chrono::high_resolution_clock::now();

blockMatrixMultiplication(A,B,C,n,alpha,beta);

auto end = std::chrono::high_resolution_clock::now();

std::cout << "The time needed by 64 block matrix multi without library : " << std::chrono::duration_cast<std::chrono::nanoseconds>(en

//for(int i=0 ; i<n*n ; i++){
  //  std::cout<<C[i] << " " ;
//}

delete[] A;
delete[] B;
delete[] C;
```

5. Using naïve & OpenMP Library

```cpp
#include <iostream>
#include <omp.h>
#include <chrono>
using namespace std::chrono;

void gemm_omp(double alpha, double beta, double* A, double* B, double* C, int n) {
    #pragma omp parallel for
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            double temp = 0.0;

            for (int k = 0; k < n; ++k) {
                temp += A[i * n + k] * B[k * n + j];
            }

            C[i * n + j] = alpha * temp + beta * C[i * n + j];
        }
    }
}
```

In this implementation, “#pragma omp parallel for” is put outside the outermost for loop indicating the use of OpenMP parallelization for the outermost loop in the ‘gemm_omp’. When its snippet is put, the loop is parallelized and multiple threads are used to execute the loop iterations concurrently. The iteration of the oop are divided among the available threads, and each thread executes a subset of iterations independently. It allows speedup and better utilization of multicore CPUs.

```cpp
int main() {

    int n=3;
    std::cin >> n ;
    std::cout<< "n : " << n << std::endl;

    double alpha = 1.0 ;
    double beta = 1.0;

    double * A = new double[n*n];
    double * B = new double[n*n];
    double * C = new double[n*n];

        for (int i = 0; i < n*n; i++) {
            A[i] = rand() / double(RAND_MAX);
            B[i] = rand() / double(RAND_MAX);
            C[i] = rand() / double(RAND_MAX);
        }

    auto start = high_resolution_clock::now();
    gemm_omp(alpha,beta, A, B, C, n);
    auto finish = high_resolution_clock::now();

    std::cout << "time needed using OpenMP : " << duration_cast<nanoseconds>(finish-start).count()  << " ns" << std::endl;

    /*
    for(int i=0; i<n*n; i++){
        std::cout << C[i] << " " ;
    }
    std::cout << std::endl;
    */
   //deallocate memory
    delete[] A;
    delete[] B;
    delete[] C;
```

Part III – Result and Verification

1. OpenBLAS

   Element matrix C after operation performed:

```
root@FitriaZu:/mnt/d/CS205/Project5# g++ -o m
ain main.cpp -lopenblas
root@FitriaZu:/mnt/d/CS205/Project5# ./main
n = 3
Time needed for OpenBLAS : 3254803ns
31 38 45 70 86 102 109 134 159 root@FitriaZu:
/mnt/d/CS205/Project5#
```

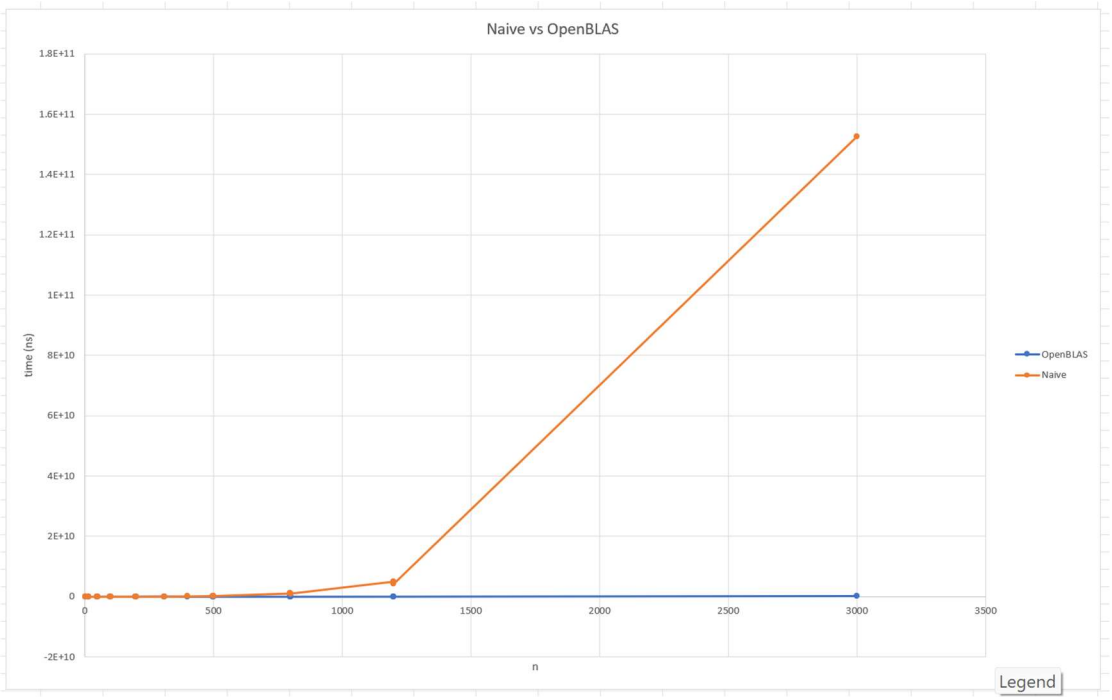
```
500
n = 500
Time needed for OpenBLAS : 11119343ns
root@FitriaZu:/mnt/d/CS205/Project5# g++ -o main main.cpp -lopenblas
root@FitriaZu:/mnt/d/CS205/Project5# ./main
798
n = 798
Time needed for OpenBLAS : 15895358ns
root@FitriaZu:/mnt/d/CS205/Project5# ./main
1200
n = 1200
Time needed for OpenBLAS : 29737408ns
root@FitriaZu:/mnt/d/CS205/Project5# g++ -o main main.cpp -lopenblas
root@FitriaZu:/mnt/d/CS205/Project5# ./main
3000
n = 3000
Time needed for OpenBLAS : 199566228ns
root@FitriaZu:/mnt/d/CS205/Project5# g++ -o main main.cpp -lopenblas
root@FitriaZu:/mnt/d/CS205/Project5# ./main
5000
n = 5000
Time needed for OpenBLAS : 926701720ns
root@FitriaZu:/mnt/d/CS205/Project5# g++ -o main main.cpp -lopenblas
root@FitriaZu:/mnt/d/CS205/Project5# ./main
10000
n = 10000
Time needed for OpenBLAS : 7936752845ns
root@FitriaZu:/mnt/d/CS205/Project5# g++ -o main main.cpp -lopenblas
root@FitriaZu:/mnt/d/CS205/Project5# ./main
15000
n = 15000
Time needed for OpenBLAS : 30196995774ns
root@FitriaZu:/mnt/d/CS205/Project5# g++ -o main main.cpp -lopenblas
root@FitriaZu:/mnt/d/CS205/Project5# ./main
20000
Killed
root@FitriaZu:/mnt/d/CS205/Project5#
```

2. Naïve multiplication

   Element matrix C after operation performed:

```
root@FitriaZu:/mnt/d/CS205/pro5# g++ main.cpp
root@FitriaZu:/mnt/d/CS205/pro5# ./a.out
time for gemm using manual for loop is 654 ns
n = 3
Elements of the matrix : 31 38 45 70 86 102 109 134 159
root@FitriaZu:/mnt/d/CS205/pro5#
```

   Comparison with OpenBLAS:



It's shown that the naïve approach has larger amount of time to execute the same operation with the same matrix size. So, it's not efficient enough especially for greater matrix size.

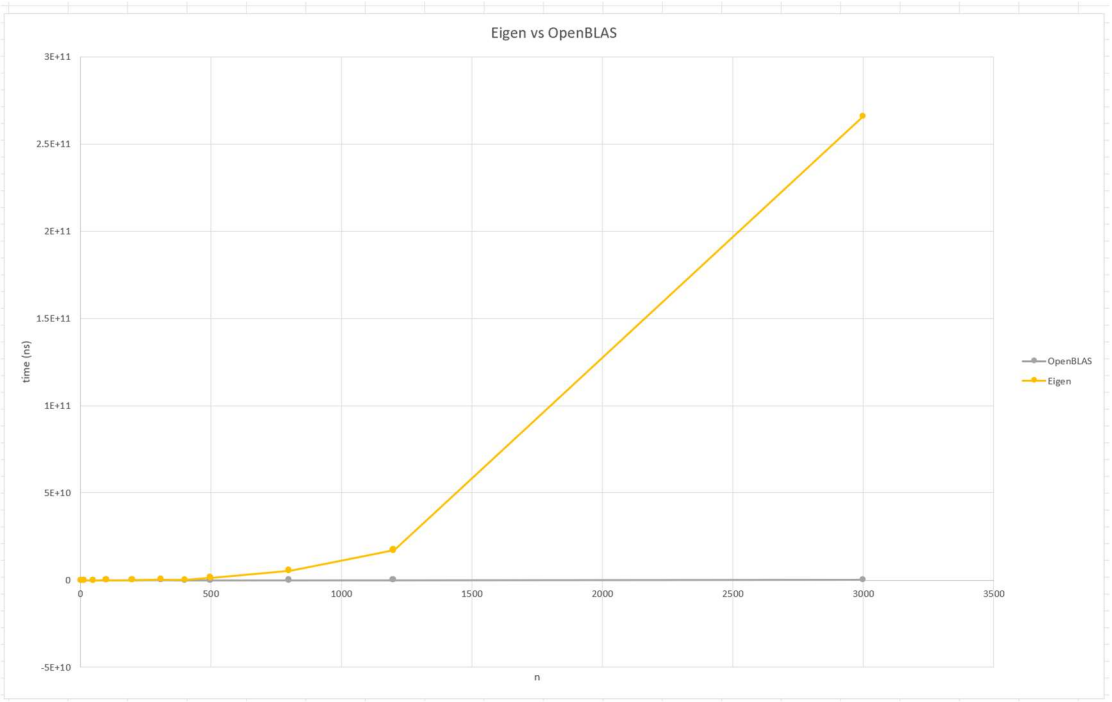
3. Using Eigen Library

Element matrix C after operation performed:

```
root@FitriaZu:/mnt/d/CS205/Project5# g++ -o eig eig.cpp -I/usr/inclu
de/eigen3
root@FitriaZu:/mnt/d/CS205/Project5# ./eig
Time needed using eigen library: 1337106ns
 31  38  45
 70  86 102
109 134 159
root@FitriaZu:/mnt/d/CS205/Project5#
```

Comparison with OpenBLAS:

Although Eigen library is straightforward and easy to use, unfortunately the time execution is still much slower than OpenBLAS especially when the matrix size become larger.



4. Using naïve + OpenMP

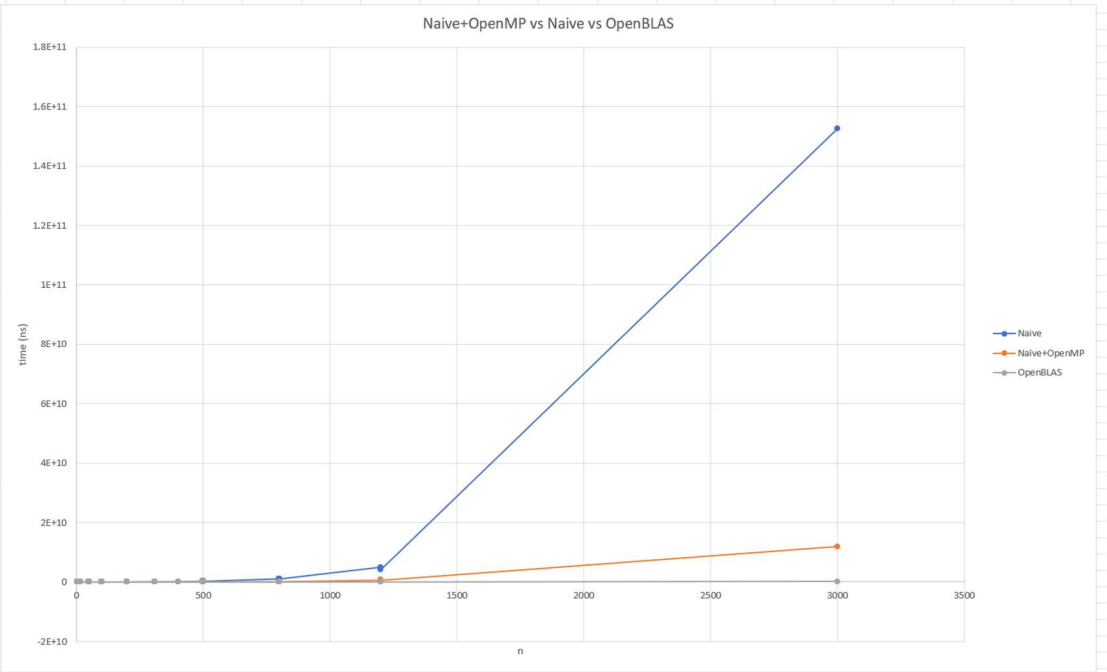
Element matrix C after operation performed:

```
root@FitriaZu:/mnt/d/CS205/Project5# g++ -fopenmp omp.cpp -o omp
root@FitriaZu:/mnt/d/CS205/Project5# ./omp
time needed using OpenMP : 13845362 ns
31 38 45 70 86 102 109 134 159
root@FitriaZu:/mnt/d/CS205/Project5#
```

```
time needed using OpenMP : 147887076 ns
root@FitriaZu:/mnt/d/CS205/Project5# g++ -fopenmp omp.cpp -o omp
^[[Aroot@FitriaZu:/mnt/d/CS205/Project5#./omp
1198
n : 1198
time needed using OpenMP : 629049947 ns
root@FitriaZu:/mnt/d/CS205/Project5# g++ -fopenmp omp.cpp -o omp
^[[Aroot@FitriaZu:/mnt/d/CS205/Project5#./omp
1199
n : 1199
time needed using OpenMP : 710535950 ns
root@FitriaZu:/mnt/d/CS205/Project5# g++ -fopenmp omp.cpp -o omp
root@FitriaZu:/mnt/d/CS205/Project5# ./omp
1200
n : 1200
time needed using OpenMP : 592446625 ns
root@FitriaZu:/mnt/d/CS205/Project5# g++ -fopenmp omp.cpp -o omp
^[[Aroot@FitriaZu:/mnt/d/CS205/Project5#./omp
3000
n : 3000
time needed using OpenMP : 11865288823 ns
root@FitriaZu:/mnt/d/CS205/Project5# g++ -fopenmp omp.cpp -o omp
^[[Aroot@FitriaZu:/mnt/d/CS205/Project5#./omp
5000
n : 5000
time needed using OpenMP : 70419151584 ns
root@FitriaZu:/mnt/d/CS205/Project5# g++ -fopenmp omp.cpp -o omp
root@FitriaZu:/mnt/d/CS205/Project5# ./omp
10000
n : 10000
time needed using OpenMP : 1317288017571 ns
```

Table execution time required (in ns):

	A	B	C	D	E
	n	OpenBLAS (ns)	naive (ns)	eigen (ns)	OpenMP+naïve
2	3	46566	568	1337106	13845362
3	4	42402	757	93924	20937085
4	5	328902	5894	4846964	14668716
5	13	87067	9286	5266069	17422024
6	14	108354	9366	1384842	18000322
7	15	2901055	10989	7034864	19328702
8	48	96556	324864	3328596	13547680
9	49	820080	386733	3484822	17433425
10	50	1966834	370489	4526832	16367870
11	98	63369494	2950750	15923630	14107267
12	99	47995997	3027492	17433153	13433542
13	100	37813151	3011646	16850411	12020380
14	198	33641567	22731243	90562023	19452132
15	199	101599356	17916734	115562314	15765979
16	200	45786516	15995536	133722777	19489125
17	308	73182350	54161299	435375570	22569665
18	309	68253005	54177241	424875784	27747917
19	400	5205500	118945380	374773224	58494531
20	498	13311160	239041250	1502613458	54908489
21	499	22236320	248137316	1426943150	52683540
22	500	11119343	247427279	1420982095	50985309
23	798	15895358	1087016135	5493245072	173808029
24	799	15130182	1096605105	5690366542	185076412
25	800	10973845	1048763516	5406208115	147887076
26	1198	27728019	4981783139	17273603827	629049947
27	1199	34190986	4712171439	17785184150	710535950
28	1200	29737408	4255870058	16934693645	592446625
29	3000	199566228	1.52667E+11	2.65977E+11	11865288823
30	5000	926701720		6.8024E+11	70419151584
31	10000	7936752845			1.31729E+12
32	15000	30196995774			



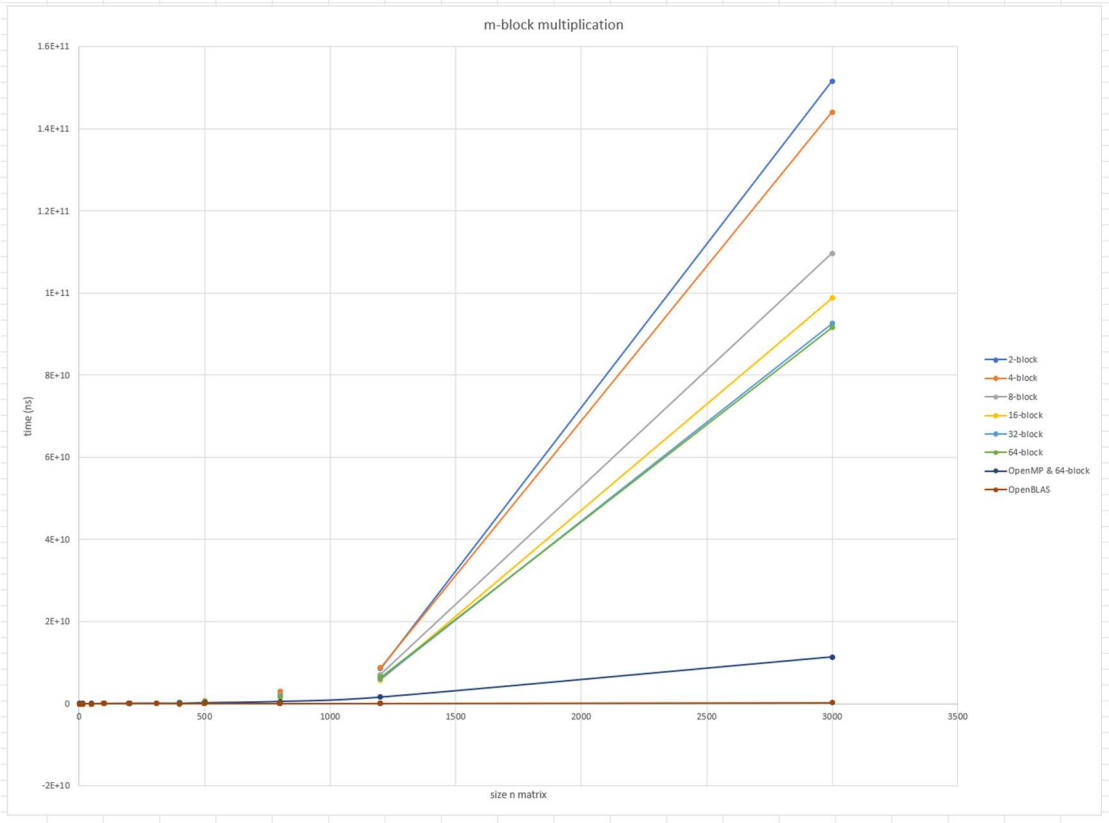
After implementing OpenMP to naïve approach, it can be seen that time required is much reduced compare with without implementing OpenMP, although still cannot perform more efficient than OpenBLAS.

5. m-block matrix multiplication

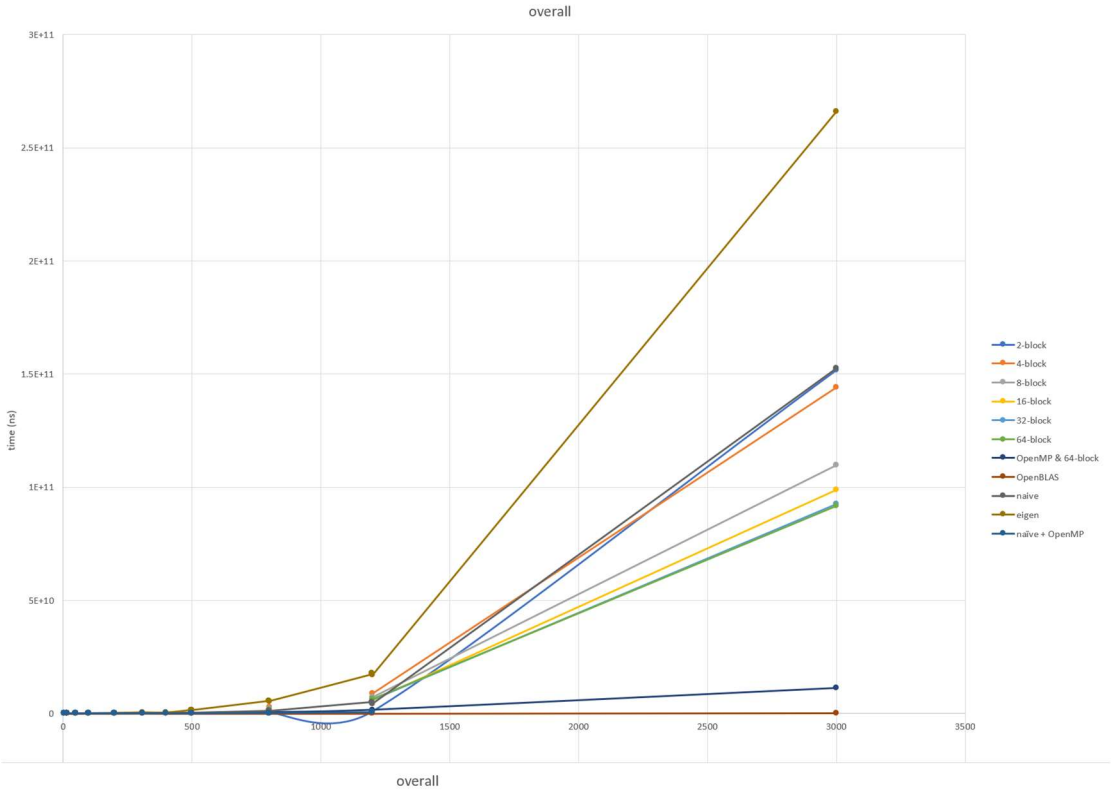
   Element matrix C after operation performed:

```
root@FitriaZu:/mnt/d/CS205/Project5# g++ block.cpp
root@FitriaZu:/mnt/d/CS205/Project5# ./a.out
n = 3
The time needed by block matrix multi without library : 1360 ns
31 38 45 70 86 102 109 134 159 root@FitriaZu:/mnt/d/CS205/Project5#
```

n (matrix size is n*n)	2-block	4-block	8-block	32-block	64-block	64-block + OpenMP
3	1360	713	559	1230	620	39142019
15	26873	15759	11717	11665	13667	32560921
50	740056	792006	424020	427786	398215	27227096
200	61922972	36352374	38623615	61398319	36937102	96136473
400	126871053	368441678	255502168	272831537	251091796	121503848
500	211963647	688153987	475520709	499895268	430559537	247939479
800	491391082	2937953666	1898556485	1859170413	1557129502	576877211
1200	1061437717	8736377973	5760683233	6089684945	6471816222	1623985904
3000	1.51632E+11	1.44071E+11	98747359550	92586401781	91665304620	11388654504



According to the graph, 64-block tend to have less execution time compared with other lower m-block. Additionally, when OpenMP is implemented into it, the gradient of the graph goes much smaller, although OpenBLAS still has lower time execution.



Part IV – Difficulties and Solutions

1. To verify that implementation library operates correctly, fixed number for element and matrix sized were generated and checked if the outputs are the same.
2. When testing each implementation, the codes were run in separate file in order to be easier to observe the result and avoid other unexpected errors and the submitted CPP file is including all the implementation.
3. The implementation may not be able to be faster than OpenBLAS library, but at least the data can show the comparison each of the implementation.
4. The scalar alpha and beta is set and fixed as 1.0 just to make the observation easier.