CS205 C/ C++ Programming – Project 4

Name: 凌硕（Ling Shuo）
SID：11912409

Part 1 – Analysis

The project is a library for normal matrix multiplication and improved version in programming language C. To finish the work given by the requirements, I modified something important things based on project 3 and wrote new code to speed up the multiplications along matrices.

1 – Struct for matrices (modified version)

Consider the struct of a matrix, I should put the numbers of rows and columns in it. They are trivial since they're definitely integers. But an important change is that I no longer use the `int` type but the `size_t` type which can ensure the nonnegativity and store larger number of rows and columns. And I used a pointer to describe the address of the elements of a matrix rather than a double one. This helps to achieve the continuity of memory, which speeds up reading and writing data. And that makes sense because if `a` is a matrix, and `p` is the pointer, then we have `a[i][j]` = `p[i * a.columns + j]`. Above all, the struct contains `size_t rows`, `size_t columns` and `float * elements`.
Please see the code in Part 2. And this part of the code is written in file ***head.h***.

2 – Creating a matrix（modified version）

In project 3, I wanted the user to be free to enter the elements of the matrix they want to create. But when the size of the matrix is large, it is obviously not possible for the user to manually enter every elements. Thus modified this function which initializes a matrix based on the number of `rows` and `columns` input and assigns every elements of the matrix initial value of 0s. The purpose of initializing the element to 0 is for the convenience of computation. And I created a new function for users to set the values of the matrices as they want: `void setValue(Matrix *a, float *t, size_t len_of_t)` where `float *t` is the one-dimensional array waited for assignment and `size_t len_of_t` is its length.
Remark 1 : It is worth mentioning that if the arguments input are illegal, I will set the `rows` and `columns` of the matrix to 0 and set the pointer `elements` to NULL.
Please see the code in Part 2. And this part of the code is written in file ***source.c***. The function declaration is `Matrix * createMatrix(size_t r, size_t c)`.

3 – Deleting a matrix (modified version)

To achieve deleting a matrix completely, I used the double pointer of a matrix as the argument.

That is, we can achieve the function by inputting the address of the pointer of the matrix.

Remark 2 : When the input argument of a function is a pointer (or double pointer), it is necessary to check the pointer before using it.

Please see the code in Part 2. And this part of the code is written in file ***source.c***. The function declaration is `void deleteMatrix(Matrix ** a)`.

4 – Printing a matrix (same as the function in project 3)

The function in this section is exactly the same as this in project 3.

Remark 3 : In a function, if we do not want to modify the matrices pointed by the pointers which are given by the input arguments, it is better to use constant input arguments, for example, `const struct Matrix * a`. The same is true for other functions if necessary.

Please see the code in Part 2. And this part of the code is written in file ***source.c***. The function declaration is `void printMatrix(const Matrix * a)`.

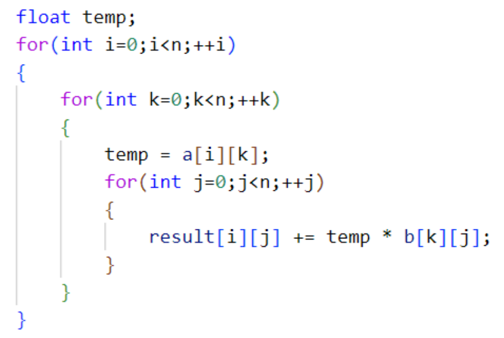
5 – Multiplying two matrices (normal version)

Assume the matrices are `a` and `b`, the result matrix is `result`. Then we have $result[i][j] = \sum_{k=1}^{n} a[i][k] \times b[k][j]$.

In this function, we use the normal method to achieve the function. That is, through the three-layers loop, we can get each element of the resulting matrix.

One important thing is that I modified this function to become a new one whose output is bool type (we should include a head file in C to use bool type), and this change is helpful for checking the correctness of the input pointers and allowing the user to check whether the multiplication is finished (if the output is true, then the multiplication is finished).

Remark 4 : In this three-layers loop, `i` and `j` means the element in i-th row and j-th column of the resulting matrix, and `k` is used for summing the several products. In fact, above method is called ijk loops method. This is not a wise choice because of the large jump in reading memory caused by such a loop order. But we still wrote it here as the most basic one. And we can rely on swapping the order of the loops to speed up matrix multiplication on the principle that we want to read memory with as few jumps as possible. A more sensible option is ikj loops method, it is not listed in our code file because it is not easy to optimize for our later methods which have greater influence, but I still present the code here as an easy example:



```
float temp;
for(int i=0;i<n;++i)
{
    for(int k=0;k<n;++k)
    {
        temp = a[i][k];
        for(int j=0;j<n;++j)
        {
            result[i][j] += temp * b[k][j];
        }
    }
}
```

Please see the code in Part 2. And this part of the code is written in file ***source.c***. The function declaration is `bool matmul_plain(const Matrix * a, const Matrix * b, Matrix * result)`.

6 – Multiplying two matrices (improved version)

Here we used SIMD and OpenMP to speed up our normal method of multiplication of matrices. Before doing some modifications to our normal function of multiplication, we need a function which can transpose matrices, and the approach to it is trivial. The function declaration is `Matrix *transpose(const Matrix *mat)`.

The reason why we need to transpose a matrix is to read data in memory continuously with function `_mm256_loadu_ps`. In fact, we greatly speed up the inner product operator by using registers since they can store eight data in each one and compute the products of eight pairs of data between two registers at the same time. Also, we use OpenMP to spend up our loops since our computers have several cores and we can use them in parallel.

Remark 5 : We want each core of the computer to be used as fully as possible for computa -tion, so we want each core to be assigned to the heavier tasks with little times, rather than being assigned to the easier tasks many times, which will waste a lot of time on assigning tasks. So we chose to split the outermost loop and let it compute in parallel.

Please see the code in Part 2. And this part of the code is written in file ***source.c***. The function declaration is `bool matmul_improved(const Matrix * a, const Matrix * b, Matrix * result)`.

7 – Creating matrix randomly

We can create a matrix which has the size you like and has random elements within each interval you like by this function.

Please see the code in Part 2. And this part of the code is written in file ***source.c***. The function declaration is `Matrix *randMat(size_t rows, size_t cols, float min, float max)`.

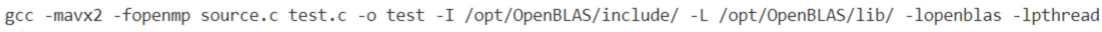
8 – Testing

We ran multiple rounds of tests on the correctness and speed of multiple methods. For details about the test results, please see Part 3.

And I used OpenBLAS to check my correctness and compared the speed of my method and OpenBLAS (please see Part 3).

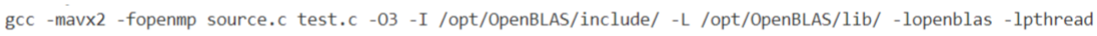
Please see the code in Part 2. And this part of the code is written in file ***test.c***.

Remark 7 : Since we used SIMD and OpenMP for acceleration, we need to add '-fopenmp -mavx2' at compile time. Here I record my method of compiling with the using of OpenBLAS:



```
gcc -mavx2 -fopenmp source.c test.c -o test -I /opt/OpenBLAS/include/ -L /opt/OpenBLAS/lib/ -lopenblas -lpthread
```

And using '-O3' instead of '-o' to compile can speed up the code :



```
gcc -mavx2 -fopenmp source.c test.c -O3 -I /opt/OpenBLAS/include/ -L /opt/OpenBLAS/lib/ -lopenblas -lpthread
```

Part 2 – Code

In this part, I put some important code.

1 – struct Matrix in head.h

```c
typedef struct Matrix
{
    size_t rows;
    size_t columns;
    float * elements;
}Matrix;
```

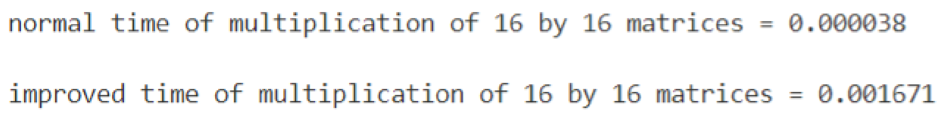
2 – code for partial functions in source.c

```c
Matrix * createMatrix(size_t r, size_t c)
{
    Matrix * a = (Matrix *)malloc(sizeof(Matrix));
    if(r<1||c<1)
    {
        a->rows=0;
        a->columns=0;
        a->elements=NULL;
    }
    else
    {
        a->rows=r;
        a->columns=c;
        // a->elements = (float *)malloc(r*c*sizeof(float));
        a->elements = (aligned_alloc(256, r*c*sizeof(float)));
        memset(a->elements, 0.0f, r*c* sizeof(float));
        //printf(“Created successfully!\n\n”);
    }
    return a;
}

void deleteMatrix(Matrix ** a)
{
    if(a==NULL)
    {
        return;
    }
    if(*a==NULL)
    {
        return;
    }
    if((*a)->elements==NULL)
    {
        return;
    }
    free((*a)->elements);
    (*a)->elements = NULL;
    free(*a);
    (*a) = NULL;
    // printf(“Deleted successfully!\n”);
    // printf(“\n”);
}


bool matmul_plain(const Matrix * a, const Matrix * b, Matrix * result)
{
    if(a==NULL||b==NULL||result==NULL)
    {
        return false;
    }
    if(a->elements==NULL||b->elements==NULL)
    {
        return false;
    }
    else if(a->columns == b->rows && a->rows == result->rows && b->columns == result->columns &&
a->rows!=0 && b->rows!=0)
    {
        for(size_t i = 0; i < result->rows; i++)
        {
            for(size_t j = 0; j < result->columns; j++)
            {
                for(size_t k = 0; k < a->columns; k++)
                {
                    result->elements[i*result->columns+j] = result->elements[i*result->columns+j] +
a->elements[i*a->columns+k] * b->elements[k*b->columns+j];
                }
            }
        }
        return true;
    }
    else
    {
        printf(“The dimensions of matrices are not consistent!\n”);
        return false;
    }
}


bool matmul_improved(const Matrix * a, const Matrix * b,Matrix * result)
{
    if(a==NULL||b==NULL||result==NULL)
    {
        return false;
    }
    if(a->elements==NULL||b->elements==NULL)
    {
        return false;
    }
    if(a->columns == b->rows && a->rows == result->rows && b->columns == result->columns &&
a->rows!=0 && b->rows!=0)
    {
        Matrix *b_tran = transpose(b);

        size_t mat_size = (a->rows) * (b->columns);
        if (result->rows * result->columns != mat_size)
        {
            free(result->elements);
            result->elements = (aligned_alloc(256, mat_size * sizeof(float)));
        }
        result->rows = a->rows;
        result->columns = b->columns;

        #pragma omp parallel for
        for (size_t i = 0; i < a->rows; i++)
        {
            for (size_t j = 0; j < b->columns; j++)
            {
                size_t k;

                for (k = 0; k < (a->columns / 8) * 8; k += 8)
                {
                    float sum[8] __attribute__((aligned(256))) = {0};
                    __m256 aa = _mm256_loadu_ps(a->elements + (i * (a->columns) + k));
                    __m256 bb = _mm256_loadu_ps(b_tran->elements + (j * (b_tran->columns) + k));
                    __m256 cc = _mm256_mul_ps(aa, bb);

                    _mm256_storeu_ps(sum, cc);
                    result->elements[i * (result->columns) + j] += (sum[0] + sum[1] + sum[2] + sum[3]
+ sum[4] + sum[5] + sum[6] + sum[7]);
```

```
                }

                for(k = (a->columns / 8) * 8; k < a->columns; k++)
                {
                    result->elements[i * (result->columns) + j] += a->elements[i*(a->columns)+k] *
b->elements[k*(b->columns)+j];
                }

            }
        }

        deleteMatrix(&b_tran);
        return true;
    }
    else
    {
        printf(“The dimensions of matrices are not consistent!\n”);
        return false;
    }
}


Matrix *randMat(size_t rows, size_t cols, float min, float max)
{
    Matrix *result = createMatrix(rows, cols);
    for (size_t i = 0; i < rows * cols; i++)
    {
        result->elements[i] = min + 1.0 * (rand() % RAND_MAX) / RAND_MAX * (max – min);
    }
    return result;
}


Matrix *transpose(const Matrix *mat)
{
    if (mat == NULL)
    {
        return NULL;
    }
    if (mat->elements == NULL)
    {
        return NULL;
    }
    Matrix *result = createMatrix(mat->columns, mat->rows);
    for (size_t i = 0; i < mat->rows; i++)
    {
        for (size_t j = 0; j < mat->columns; j++)
        {
            result->elements[j * mat->rows + i] = mat->elements[i * (mat->columns) + j];
        }
    }
    return result;
}
```

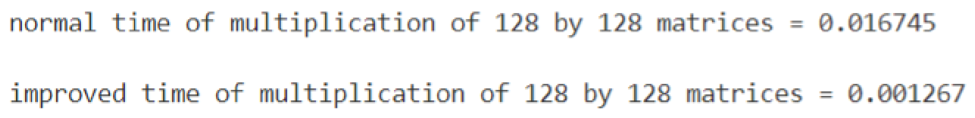
Part 3 – Result & Verification

To show that our method really speed things up, we created matrices of several sizes (16×16、128×128、$1k \times 1k$、$2k \times 2k$、$4k \times 4k$、$8k \times 8k$) and recorded the time of finishing multiplications of different size of matrices by different methods. The result is given here:



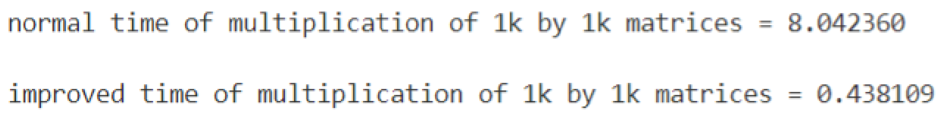
```
normal time of multiplication of 16 by 16 matrices = 0.000038

improved time of multiplication of 16 by 16 matrices = 0.001671
```

(multiplication of 16×16 matrices)



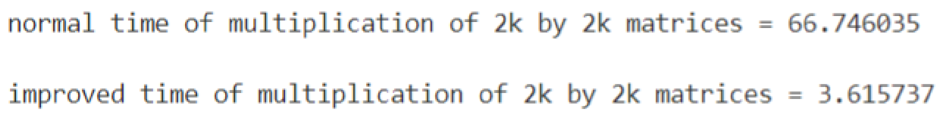
```
normal time of multiplication of 128 by 128 matrices = 0.016745

improved time of multiplication of 128 by 128 matrices = 0.001267
```

(multiplication of 128×128 matrices)



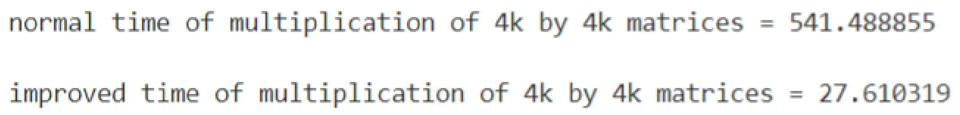
```
normal time of multiplication of 1k by 1k matrices = 8.042360

improved time of multiplication of 1k by 1k matrices = 0.438109
```

(multiplication of $1k \times 1k$ matrices)



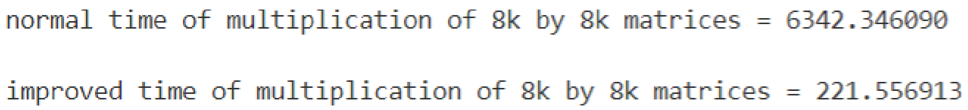
```
normal time of multiplication of 2k by 2k matrices = 66.746035

improved time of multiplication of 2k by 2k matrices = 3.615737
```

(multiplication of $2k \times 2k$ matrices)



```
normal time of multiplication of 4k by 4k matrices = 541.488855

improved time of multiplication of 4k by 4k matrices = 27.610319
```

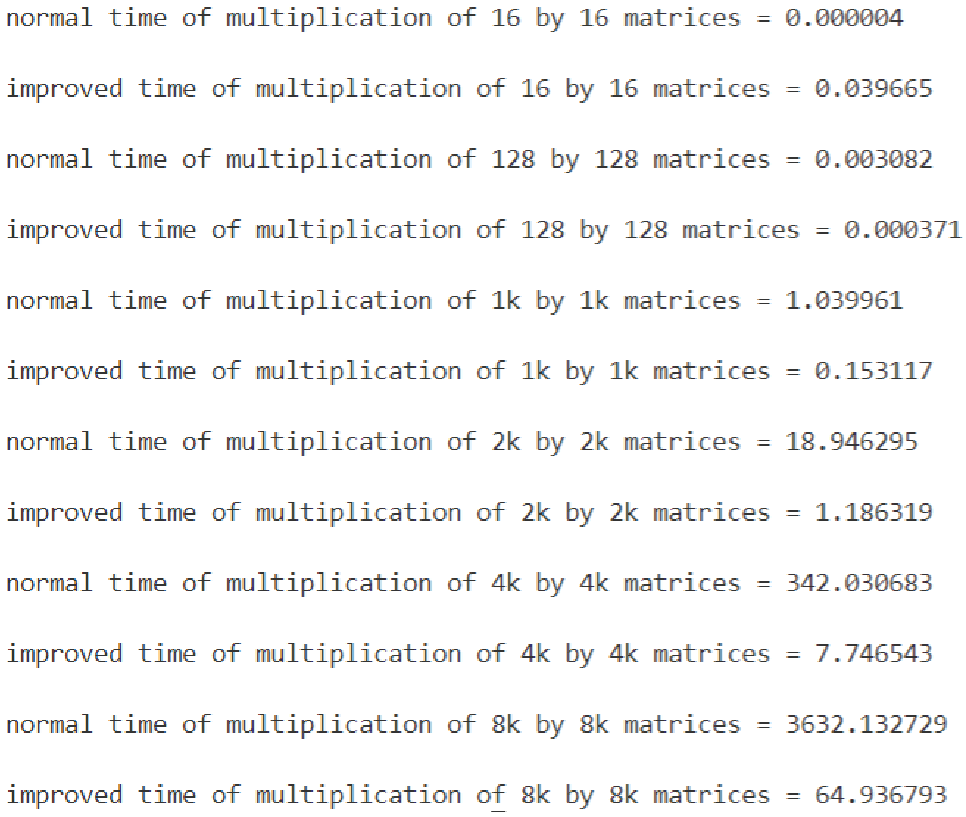
(multiplication of $4k \times 4k$ matrices)



```
normal time of multiplication of 8k by 8k matrices = 6342.346090

improved time of multiplication of 8k by 8k matrices = 221.556913
```

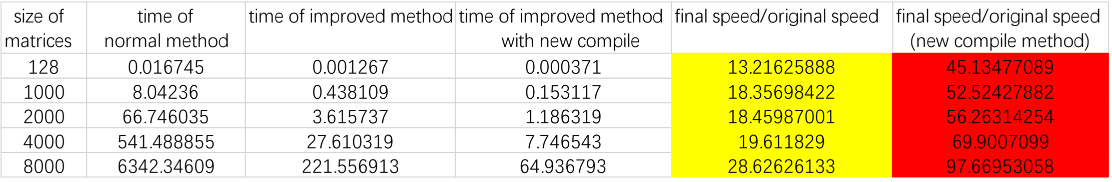
(multiplication of $8k \times 8k$ matrices)

And I changed the way of compiling (use '-O3'), here is the result:



```
normal time of multiplication of 16 by 16 matrices = 0.000004

improved time of multiplication of 16 by 16 matrices = 0.039665

normal time of multiplication of 128 by 128 matrices = 0.003082

improved time of multiplication of 128 by 128 matrices = 0.000371

normal time of multiplication of 1k by 1k matrices = 1.039961

improved time of multiplication of 1k by 1k matrices = 0.153117

normal time of multiplication of 2k by 2k matrices = 18.946295

improved time of multiplication of 2k by 2k matrices = 1.186319

normal time of multiplication of 4k by 4k matrices = 342.030683

improved time of multiplication of 4k by 4k matrices = 7.746543

normal time of multiplication of 8k by 8k matrices = 3632.132729

improved time of multiplication of 8k by 8k matrices = 64.936793
```

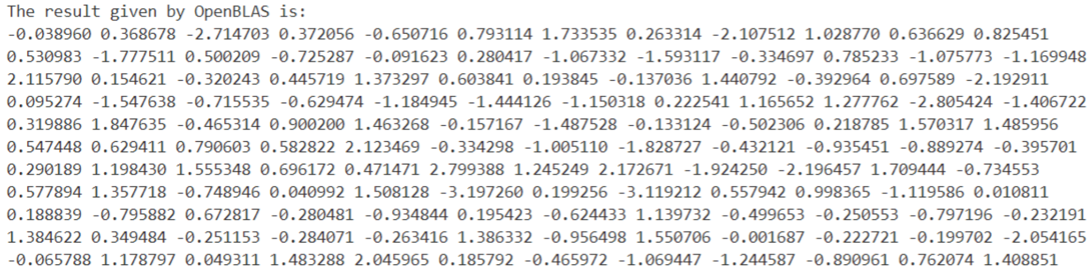
In order to directly observe the speed change of matrix multiplication before and after optimization, we fill the above recorded data into the table:



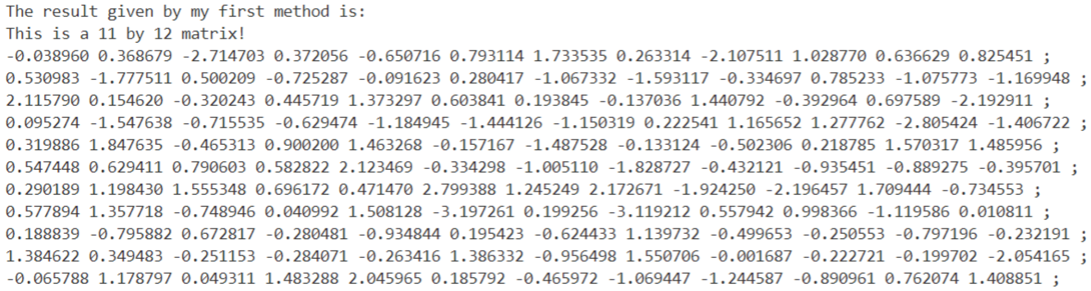
size of matrices	time of normal method	time of improved method	time of improved method with new compile	final speed/original speed	final speed/original speed (new compile method)
128	0.016745	0.001267	0.000371	13.21625888	45.13477089
1000	8.04236	0.438109	0.153117	18.35698422	52.52427882
2000	66.746035	3.615737	1.186319	18.45987001	56.26314254
4000	541.488855	27.610319	7.746543	19.611829	69.9007099
8000	6342.34609	221.556913	64.936793	28.62626133	97.66953058

According to the running results, it can be seen that our optimizations do play a good effect (for large size matrices), and this effect is more obvious when the matrix dimension is large. For multiplication of $8k \times 8k$ matrices, the improved method is about thirty times faster than the normal method. And if you use the new way to compile, the improved method is about 100 times faster than the original method when they deal with the multiplication of two $8k \times 8k$ matrices.

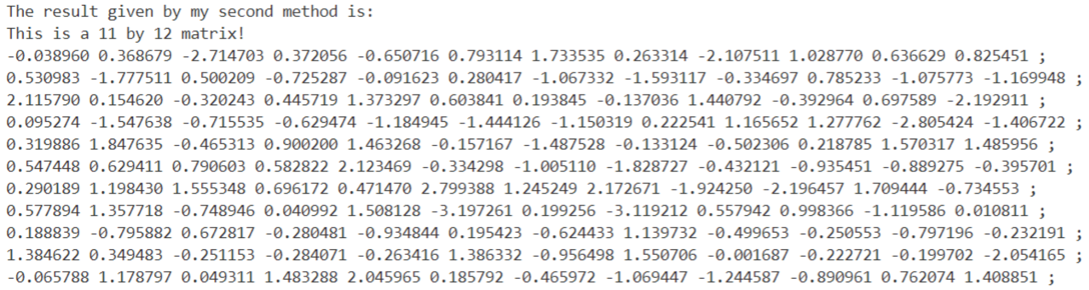
Now, let us introduce OpenBLAS and verify the correctness of my methods. Without loss of generality, I create a 11*13 matrix and a 13*12 matrix, and compute the product with them by my methods and OpenBLAS, the result is here:



```
The result given by OpenBLAS is:
-0.038960 0.368678 -2.714703 0.372056 -0.650716 0.793114 1.733535 0.263314 -2.107512 1.028770 0.636629 0.825451
0.530983 -1.777511 0.500209 -0.725287 -0.091623 0.280417 -1.067332 -1.593117 -0.334697 0.785233 -1.075773 -1.169948
2.115790 0.154621 -0.320243 0.445719 1.373297 0.603841 0.193845 -0.137036 1.440792 -0.392964 0.697589 -2.192911
0.095274 -1.547638 -0.715535 -0.629474 -1.184945 -1.444126 -1.150318 0.222541 1.165652 1.277762 -2.805424 -1.406722
0.319886 1.847635 -0.465314 0.900200 1.463268 -0.157167 -1.487528 -0.133124 -0.502306 0.218785 1.570317 1.485956
0.547448 0.629411 0.790603 0.582822 2.123469 -0.334298 -1.005110 -1.828727 -0.432121 -0.935451 -0.889274 -0.395701
0.290189 1.198430 1.555348 0.696172 0.471471 2.799388 1.245249 2.172671 -1.924250 -2.196457 1.709444 -0.734553
0.577894 1.357718 -0.748946 0.040992 1.508128 -3.197260 0.199256 -3.119212 0.557942 0.998365 -1.119586 0.010811
0.188839 -0.795882 0.672817 -0.280481 -0.934844 0.195423 -0.624433 1.139732 -0.499653 -0.250553 -0.797196 -0.232191
1.384622 0.349484 -0.251153 -0.284071 -0.263416 1.386332 -0.956498 1.550706 -0.001687 -0.222721 -0.199702 -2.054165
-0.065788 1.178797 0.049311 1.483288 2.045965 0.185792 -0.465972 -1.069447 -1.244587 -0.890961 0.762074 1.408851
```



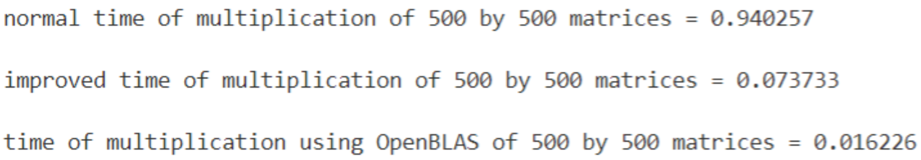
```
The result given by my first method is:
This is a 11 by 12 matrix!
-0.038960 0.368679 -2.714703 0.372056 -0.650716 0.793114 1.733535 0.263314 -2.107511 1.028770 0.636629 0.825451 ;
0.530983 -1.777511 0.500209 -0.725287 -0.091623 0.280417 -1.067332 -1.593117 -0.334697 0.785233 -1.075773 -1.169948 ;
2.115790 0.154620 -0.320243 0.445719 1.373297 0.603841 0.193845 -0.137036 1.440792 -0.392964 0.697589 -2.192911 ;
0.095274 -1.547638 -0.715535 -0.629474 -1.184945 -1.444126 -1.150319 0.222541 1.165652 1.277762 -2.805424 -1.406722 ;
0.319886 1.847635 -0.465313 0.900200 1.463268 -0.157167 -1.487528 -0.133124 -0.502306 0.218785 1.570317 1.485956 ;
0.547448 0.629411 0.790603 0.582822 2.123469 -0.334298 -1.005110 -1.828727 -0.432121 -0.935451 -0.889275 -0.395701 ;
0.290189 1.198430 1.555348 0.696172 0.471470 2.799388 1.245249 2.172671 -1.924250 -2.196457 1.709444 -0.734553 ;
0.577894 1.357718 -0.748946 0.040992 1.508128 -3.197261 0.199256 -3.119212 0.557942 0.998366 -1.119586 0.010811 ;
0.188839 -0.795882 0.672817 -0.280481 -0.934844 0.195423 -0.624433 1.139732 -0.499653 -0.250553 -0.797196 -0.232191 ;
1.384622 0.349483 -0.251153 -0.284071 -0.263416 1.386332 -0.956498 1.550706 -0.001687 -0.222721 -0.199702 -2.054165 ;
-0.065788 1.178797 0.049311 1.483288 2.045965 0.185792 -0.465972 -1.069447 -1.244587 -0.890961 0.762074 1.408851 ;
```



```
The result given by my second method is:
This is a 11 by 12 matrix!
-0.038960 0.368679 -2.714703 0.372056 -0.650716 0.793114 1.733535 0.263314 -2.107511 1.028770 0.636629 0.825451 ;
0.530983 -1.777511 0.500209 -0.725287 -0.091623 0.280417 -1.067332 -1.593117 -0.334697 0.785233 -1.075773 -1.169948 ;
2.115790 0.154620 -0.320243 0.445719 1.373297 0.603841 0.193845 -0.137036 1.440792 -0.392964 0.697589 -2.192911 ;
0.095274 -1.547638 -0.715535 -0.629474 -1.184945 -1.444126 -1.150319 0.222541 1.165652 1.277762 -2.805424 -1.406722 ;
0.319886 1.847635 -0.465313 0.900200 1.463268 -0.157167 -1.487528 -0.133124 -0.502306 0.218785 1.570317 1.485956 ;
0.547448 0.629411 0.790603 0.582822 2.123469 -0.334298 -1.005110 -1.828727 -0.432121 -0.935451 -0.889275 -0.395701 ;
0.290189 1.198430 1.555348 0.696172 0.471470 2.799388 1.245249 2.172671 -1.924250 -2.196457 1.709444 -0.734553 ;
0.577894 1.357718 -0.748946 0.040992 1.508128 -3.197261 0.199256 -3.119212 0.557942 0.998366 -1.119586 0.010811 ;
0.188839 -0.795882 0.672817 -0.280481 -0.934844 0.195423 -0.624433 1.139732 -0.499653 -0.250553 -0.797196 -0.232191 ;
1.384622 0.349483 -0.251153 -0.284071 -0.263416 1.386332 -0.956498 1.550706 -0.001687 -0.222721 -0.199702 -2.054165 ;
-0.065788 1.178797 0.049311 1.483288 2.045965 0.185792 -0.465972 -1.069447 -1.244587 -0.890961 0.762074 1.408851 ;
```

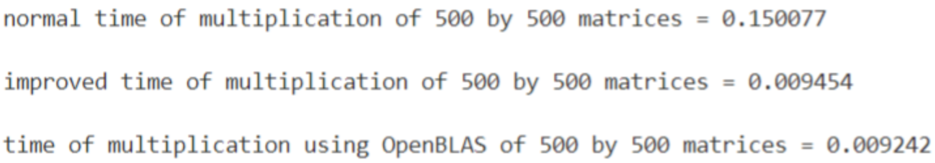
Compare the three result matrices, we can conclude that my methods are correct.

To see how fast OpenBLAS can reach, I computed the product of two 500*500 matrix with my methods and OpenBLAS, and record their time. The result with normal compile is here:



```
normal time of multiplication of 500 by 500 matrices = 0.940257

improved time of multiplication of 500 by 500 matrices = 0.073733

time of multiplication using OpenBLAS of 500 by 500 matrices = 0.016226
```

The result of new compile ('-O3') is here:



```
normal time of multiplication of 500 by 500 matrices = 0.150077

improved time of multiplication of 500 by 500 matrices = 0.009454

time of multiplication using OpenBLAS of 500 by 500 matrices = 0.009242
```

The above results show that the final version improved method (SIMD + OpenMP + '-O3' compile) we have used has achieved a high speed.

Part 4 - Difficulties & Solutions

1. The use of one-dimensional array to store matrix elements is conducive to the coherence of memory and the speed of reading and writing data.
2. Using the transpose of the matrix allows us to compute with less memory jumps when reading data.
3. SIMD and OpenMP can be used to significantly improve computing efficiency.
4. Using '-O3' to compile can significantly improve computing efficiency.
5. OpenBLAS is about the same speed as our final version (SIMD + OpenMP + '-O3' compile).
6. $64k \times 64k$ matrix is hard to create since it need at least $64000 \times 64000 \times 4\, byte \approx 15.26\, GB$ memory. And based on the time of previous multiplications of different size of matrices, we can estimate that multiplication of $64k \times 64k$ matrices will take a huge long time since the complexity of matrix multiplication is $O(n^3)$.
7. There are ways to optimize matrix multiplication based on the algorithm itself, such as Strassen algorithm which has complexity $O(n^{2.7})$ but its superiority only comes into play when n is very huge.