# CS205:C/C++ Program - Project Report 3

# Matrix Multiplication in C


> ***Name:*** 张闻城
>
> ***SID:*** *12010324*


# Contents



# 1. Analysis

## 1.1 Forget what have been learned in C++

This porject constrains the programming language can only be **C**, instead of **C++**. Unlike **C++** is object oriented, **C** is procedure oriented. There are many pratical features included in **C++**, but not included in **C**. So it is neccessary to know the difference between **C** and **C++** first.

## 1.2 Matirx multiplication

### 1.2.1 Brute force matrix multiplication

The most trivial way to implement matrix multiplication is to do calculation in order **i->j->k**. But the time complexity is $O(n^3)$, which means the time consumption will grow in an extremely fast speed when the size `n` is very large. Obviously, this method is the most unadvisable.

### 1.2.2 Some optimization of data access [1]

The most easy way to speed up the calculation progress is to optimize access of data. In the original order **i->j->k**, it requires $n^3 + n^2 + n$ times jump to access discontinuous data(data in different rows). If the order is changed into **i->k->j**, the times of jump is $2n^2 + n$. Thus the hit rate of cache rises and the speed of memory access is much quicker.

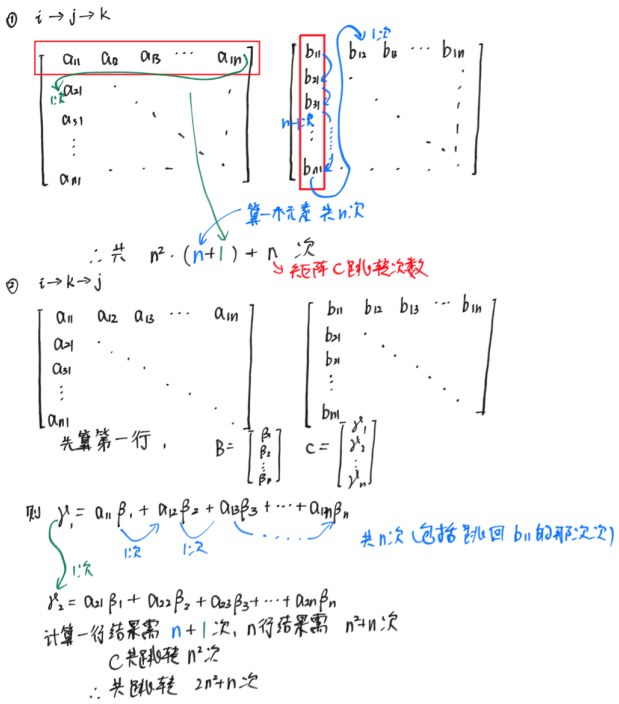



*Figure 1. Schematic explaining how **i->k->j** speed up the multiplication*

### 1.2.3 Strassen's algorithm [2]

The basic thought used in **Strassen's algorithm** is **Divide & Conquer** and **Recursion**. **Strassen** use some tricks to construct block matrices of the original two matrix, which can reduce the times of multiplication. And then calculate multiplication between two certain matrices recursively. Fianlly, the time complexity of **Strassen's algorithm** is reduced to $O(n^{log_2 7})$.

For more details of **Strassen's algorithm**, you can click the hyperlink above.

### 1.2.4 Use SIMD to speed up dot product [3]

**SIMD**, means **Single Instruction, Multiple Data**. It is the extension of basic instruction set in CPU. [4] (The main **SIMD** instrcution sets contain **AVX**, **AVX2**, **AVX512...** on x86 or x64 architecture CPU, and **NEON** on arm architecture CPU). By using **SIMD**, multiple data can be fetched by one instruction and calculated as vector, which fits for matirx multiplication very well. Compared with traditional float operation, **SIMD** can at most process several **float** type data at one time. Thus the speed of **float** type data's operation will be faster to some extent.

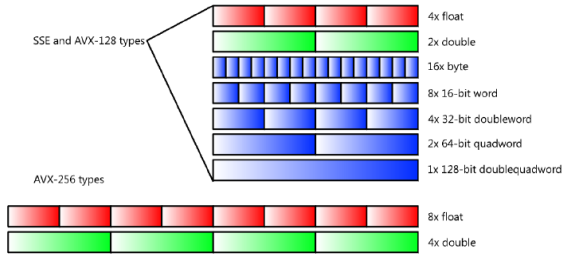



Figure 2. *Intel® AVX and Intel® SSE data types*

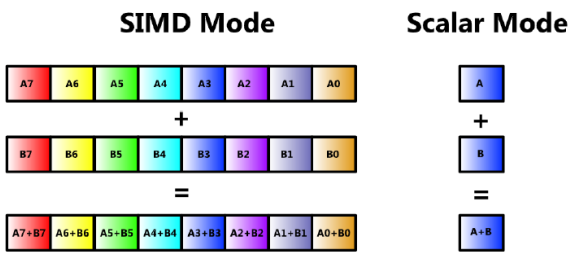



Figure 3. *SIMD versus scalar operations*

*Figure 2. Schematic of **AVX2**'s storage structure and **SIMD***

### 1.2.5 Multi-threads

In matrix multiplication, every elements in the result matrix is calculated independently. We don't need the data in certaion elements of result matrix to calculate out the other element's value. Obviously, matrix multiplication is a classic parallel issue. Thus, if we want to calculate $A_{n\times n} \cdot B_{n\times n}$ , we can divide $A_{n\times n}$ into several parts, and calculate the corresponding elements by different threads respevtively.


*Figure 3. Schematic of matrix multiplication using multi-thread*

### 1.2.6 O3 optimization of **GCC** compiler

**GCC** compiler offers several ways to optimize your program.[5] To speed up the matrix multiplication as much as possible, I turn on the option of O3 optimization in **GCC** compiler (for more details of compiler optimization, please click the hyperlink above).

## 1.3 Checkout of result's error

- The progress of matrix multiplication using brute force method(i->j->k) is quite simple and there is not much interference within the progress. So the result calculated by this way can be treated as the reference result to some extent(if the error casused by **float** can be ignored).
- To get more precise result, external library can be used(such as **OpenBLAS**, **OpenCV** and so on).

# 2. Codes

---

> ***Notification: To make the report brief as far as I can, I didn't display the codes in header files here. For more details, you can check source codes in the attachment.***

## 1.1 matrix.c

### 1.1.1 Baisc definition of **struct** Matrix and some auxiliary functions

```c
typedef struct Matrix {
    float **data;
    size_t row;
    size_t col;
} MATRIX;

/**
 * @brief <p>Create matrix's instantiation from the specific file,
 *        and then store its pointer into matrix.</p>
 */
void Matrix(MATRIX *matrix, char *filepath) {
    setRow(matrix, filepath);
    setCol(matrix, filepath);

    FILE *fp;
    if ((fp = fopen(filepath, "rt")) == NULL) {
        printf("Fail to open fp \"%s\". Exit.\n", filepath);
        exit(0);
    }

    printf("Start to read file and construct matrix.\n");
    TIME_START
    float **array = allocate2DArray(matrix->row, matrix->col);
    for (size_t i = 0; i < matrix->row; i++) {
        for (size_t j = 0; j < matrix->col; j++) {
            fscanf(fp, "%f", &array[i][j]);
        }
    }
    matrix->data = array;
    TIME_END("Reading file and constructing matrix completed. ")

    printf("Construct matrix from file \"%s\" successfully.\n", filepath);
    fclose(fp);
}

/**
 * @brief <p>Copy data from src to dest.</p>
 */
void matrixcpy(MATRIX *dest, const MATRIX *src) {
    if (dest->data != NULL) {
        free2DArray(dest->data);
    }

    dest->data = allocate2DArray(src->row, src->col);
    for (size_t i = 0; i < src->row; i++) {
        for (size_t j = 0; j < src->col; j++) {
            dest->data[i][j] = src->data[i][j];
        }
    }
}

/**
 * @brief <p>Allocate memory in heap for a certain matrix.</p>
 */
MATRIX *prepareMat(size_t row, size_t col) {
    MATRIX *resMat = (MATRIX *) malloc(sizeof(MATRIX));
    resMat->row = row;
    resMat->col = col;
    resMat->data = allocate2DArray(row, col);
```

```
    return resMat;
}

void setRow(MATRIX *matrix, char *filename) {
    FILE *fp;
    if ((fp = fopen(filename, "rt")) == NULL) {
        printf("Fail to open fp \"%s\". Exit.\n", filename);
        exit(0);
    }

    char *line = NULL;
    size_t len = 0;
    size_t rowCnt = 0;
    while (getline(&line, &len, fp) != -1) rowCnt++;
    matrix->row = rowCnt;
    fclose(fp);
}

void setCol(MATRIX *matrix, char *filename) {
    FILE *fp;
    if ((fp = fopen(filename, "rt")) == NULL) {
        printf("Fail to open fp \"%s\". Exit.\n", filename);
        exit(0);
    }

    size_t colCnt = 0;
    char *line = NULL;
    size_t len = 0;
    ssize_t read;
    if ((read = getline(&line, &len, fp)) == -1) {
        printf("Fail to read line.");
        exit(0);
    }
    for (ssize_t i = 0; i < read; i++) {
        if (line[i] == 32 || line[i] == 10) colCnt++;
    }
    matrix->col = colCnt;
    fclose(fp);
}

void freeMatrix(MATRIX *matrix) {
    free2DArray(matrix->data);
    free(matrix);
}
```

### 1.1.2 Multiplication in order i->j->k

```
void mul_ijk(MATRIX *mat_A, MATRIX *mat_B) {
    float **resArr = allocate2DArray(mat_A->row, mat_B->col);

    printf("Start to execute multiplication between two %ldx%ld matrices in order i->j->k.\n", mat_A->row, mat_A->col);
    TIME_START
    for (size_t i = 0; i < mat_A->row; i++) {
        for (size_t j = 0; j < mat_B->col; j++) {
            if (use_avx2 == FALSE) {
                float temp = 0.0f;
                for (size_t k = 0; k < mat_A->col; k++) {
                    temp += mat_A->data[i][k] * mat_B->data[k][j];
                }
                resArr[i][j] = temp;
            } else {
```

```
                resArr[i][j] = avx2_dotProduct(mat_A->data[i], &(mat_B->data[0][j]),
mat_A->row);
            }
        }
    }
    TIME_END("<<< Multiplication in order i->j->k completed. >>> ")

    writeMatrix(resArr, mat_A->row, mat_B->col, getOuFileName(mat_A->row, mat_B->col));

    free2DArray(resArr);
}
```

### 1.1.3 Multiplication in order i->k->j

```
void mul_ikj(MATRIX *mat_A, MATRIX *mat_B) {
    float **resArr = allocate2DArray(mat_A->row, mat_B->col);

    printf("Start to execute multiplication of two %ldx%ld matrices in order i->k->j.\n",
mat_A->row, mat_A->col);
    TIME_START
    for (size_t i = 0; i < mat_A->row; i++) {
        for (size_t k = 0; k < mat_A->col; k++) {
            float temp = mat_A->data[i][k];
            for (size_t j = 0; j < mat_B->col; j++) {
                resArr[i][j] += temp * mat_B->data[k][j];
            }
        }
    }
    TIME_END("<<< Multiplication in order i->k->j completed. >>> ")

    writeMatrix(resArr, mat_A->row, mat_B->col, getOuFileName(mat_A->row, mat_B->col));

    free2DArray(resArr);
}
```

### 1.1.4 Multiplication using **Strassen's Algorithm**

```
void strassen(MATRIX *mat_A, MATRIX *mat_B) {

    printf("Start to execute multiplication of two %ldx%ld matrices using Strassen.\n",
mat_A->row, mat_A->col);
    MATRIX *res;

    TIME_START
    res = strassen_(mat_A, mat_B);
    TIME_END("<<< Multiplication using Strassen completed. >>> ")

    writeMatrix(res->data, mat_A->row, mat_B->col, getOuFileName(mat_A->row, mat_B-
>col));
}

MATRIX *strassen_(MATRIX *mat_A, MATRIX *mat_B) {
    size_t n = mat_A->row;

    if (n <= 256) {
        MATRIX *res = prepareMat(mat_A->row, mat_B->col);

        if (use_avx2 == FALSE) {
            for (size_t i = 0; i < mat_A->row; i++) {
                for (size_t k = 0; k < mat_A->col; k++) {
                    float temp = mat_A->data[i][k];
```

```
                    for (size_t j = 0; j < mat_B->col; j++) {
                        res->data[i][j] += temp * mat_B->data[k][j];
                    }
                }
            }
        } else {
            if (n % 8 != 0) {
                printf("Size of matrices is not fit for SIMD optimization. Exit.\n");
                exit(0);
            }
            for (size_t i = 0; i < n; i++) {
                for (size_t j = 0; j < n; j++) {
                    res->data[i][j] = avx2_dotProduct(mat_A->data[i], &(mat_B->data[0]
[j]), n);
                }
            }
        }

        return res;
    }

    n /= 2;

    MATRIX *A11 = subMatrix(mat_A, 0, 0, n, n);
    MATRIX *A12 = subMatrix(mat_A, 0, n, n, n);
    MATRIX *A21 = subMatrix(mat_A, n, 0, n, n);
    MATRIX *A22 = subMatrix(mat_A, n, n, n, n);

    MATRIX *B11 = subMatrix(mat_B, 0, 0, n, n);
    MATRIX *B12 = subMatrix(mat_B, 0, n, n, n);
    MATRIX *B21 = subMatrix(mat_B, n, 0, n, n);
    MATRIX *B22 = subMatrix(mat_B, n, n, n, n);

    MATRIX *S1 = matSubtraction(B12, B22);
    MATRIX *S2 = matAddition(A11, A12);
    MATRIX *S3 = matAddition(A21, A22);
    MATRIX *S4 = matSubtraction(B21, B11);
    MATRIX *S5 = matAddition(A11, A22);
    MATRIX *S6 = matAddition(B11, B22);
    MATRIX *S7 = matSubtraction(A12, A22);
    MATRIX *S8 = matAddition(B21, B22);
    MATRIX *S9 = matSubtraction(A11, A21);
    MATRIX *S10 = matAddition(B11, B12);

    MATRIX *P1 = strassen_(A11, S1);
    MATRIX *P2 = strassen_(S2, B22);
    MATRIX *P3 = strassen_(S3, B11);
    MATRIX *P4 = strassen_(A22, S4);
    MATRIX *P5 = strassen_(S5, S6);
    MATRIX *P6 = strassen_(S7, S8);
    MATRIX *P7 = strassen_(S9, S10);

    //region Free Useless Matrix
    freeMatrix(A11);
    freeMatrix(A12);
    freeMatrix(A21);
    freeMatrix(A22);
    freeMatrix(B11);
    freeMatrix(B12);
    freeMatrix(B21);
    freeMatrix(B22);
    freeMatrix(S1);
    freeMatrix(S2);
    freeMatrix(S3);
    freeMatrix(S4);
```

```
    freeMatrix(S5);
    freeMatrix(S6);
    freeMatrix(S7);
    freeMatrix(S8);
    freeMatrix(S9);
    freeMatrix(S10);
    //endregion

    MATRIX *C11 = matAddition(matSubtraction(matAddition(P5, P4), P2), P6);
    MATRIX *C12 = matAddition(P1, P2);
    MATRIX *C21 = matAddition(P3, P4);
    MATRIX *C22 = matSubtraction(matSubtraction(matAddition(P5, P1), P3), P7);

    return mergeMat(C11, C12, C21, C22);
}

MATRIX *mergeMat(MATRIX *C11, MATRIX *C12, MATRIX *C21, MATRIX *C22) {
    MATRIX *mergeRes = prepareMat(C11->row + C21->row, C11->col + C12->col);

    for (size_t i = 0; i < C11->row; i++) {
        for (size_t j = 0; j < C11->col; j++) {
            mergeRes->data[i][j] = C11->data[i][j];
            mergeRes->data[i][j + C11->col] = C12->data[i][j];
            mergeRes->data[i + C21->row][j] = C21->data[i][j];
            mergeRes->data[i + C21->row][j + C22->col] = C22->data[i][j];
        }
    }

    //region Free Useless Matrix
    freeMatrix(C11);
    freeMatrix(C12);
    freeMatrix(C21);
    freeMatrix(C22);
    //endregion

    return mergeRes;
}
```

### 1.1.5 Dot product using **AVX2(SIMD)**

```
float avx2_dotProduct(const float *v1_, const float *v2_, size_t n) {
    float sum[8] = {0.0f};

    __m256 r1, r2;
    __m256 r = _mm256_setzero_ps();
    float *v1 = (float *) aligned_alloc(32, sizeof(float) * n);
    float *v2 = (float *) aligned_alloc(32, sizeof(float) * n);
    alignedCopy(v1_, v2_, v1, v2, n);

    for (size_t k = 0; k < n; k += 8) {
        r1 = _mm256_load_ps(v1 + k);
        r2 = _mm256_load_ps(v2 + k);
        r = _mm256_add_ps(r, _mm256_mul_ps(r1, r2));
    }
    _mm256_store_ps(sum, r);

    free(v1);
    free(v2);

    return (sum[0] + sum[1] + sum[2] + sum[3] + sum[4] + sum[5] + sum[6] + sum[7]);
}
```

I use **AVX2** of Intel here. In x86 CPU supporting **AVX2**, there are many registers with 256 bit-width. Thus one register can load up to 8 **float** data at one time. **Thus theoretically**, the speed of operation with **float** data using **AVX2** is 8 times faster as much as the traditional one(I will explain why I emphasize "theoretically" here later).

### 1.1.6 Multiplication using multi-threads

```
void matMul_Threads(MATRIX *mat_A, MATRIX *mat_B) {
    MATRIX *res;
    int threadsNum = 16;

    printf("Start to execute multiplication between two %ldx%ld matrices using multi-threadsNum (threadsNum >>> %d).\n",
           mat_A->row, mat_B->col, threadsNum);
    TIME_START
    res = threadMulWith_n_Threads(mat_A, mat_B, threadsNum);
    TIME_END("<<< Multiplication using multi-thread completed. >>> ")

    writeMatrix(res->data, res->row, res->col, getOuFileName(mat_A->row, mat_B->col));

    freeMatrix(res);
}

MATRIX *threadMulWith_n_Threads(MATRIX *mat_A, MATRIX *mat_B, size_t threadsNum) {
    size_t rowNumOfEveryThread = mat_A->row / threadsNum;
    MATRIX *resultMatrix = prepareMat(mat_A->row, mat_B->col);

    pthread_t *pthreads = (pthread_t *) malloc(threadsNum * sizeof(pthread_t));
    for (size_t i = 0; i < threadsNum; i++) pthreads[i] = (pthread_t) NULL;
    ResultSet **resultSets = (ResultSet **) malloc(sizeof(ResultSet *) * threadsNum);
    for (size_t i = 0; i < threadsNum; i++) {
        resultSets[i] = prepareResultSet(resultMatrix, mat_A, mat_B, i * rowNumOfEveryThread, rowNumOfEveryThread);
    }
    for (size_t i = 0; i < threadsNum; i++) {
        pthread_create(&pthreads[i], NULL, singleThreadMatMul, (void *) resultSets[i]);
    }
    for (size_t i = 0; i < threadsNum; i++) {
        if (pthread_join(pthreads[i], NULL) == 0) {
            printf("Thread[%ld] exits successfully. ", i);
        } else {
            printf("Thread[%ld] exits unsuccessfully. Exit.\n", i);
            exit(0);
        }
    }
    printf("\n");

    for (size_t i = 0; i < threadsNum; i++) {
        free(resultSets[i]);
    }
    free(resultSets);

    return resultMatrix;
}

void *singleThreadMatMul(void *args) {
    ResultSet *set = (ResultSet *) args;

    for (size_t i = 0; i < set->rowNumToCal; i++) {
        for (size_t k = 0; k < set->mat_A->col; k++) {
            float temp = set->mat_A->data[set->beginRowIndex + i][k];
            for (size_t j = 0; j < set->mat_B->col; j++) {
                set->resMat->data[set->beginRowIndex + i][j] += temp * set->mat_B->data[k][j];
```

```
            }
        }
    }

    pthread_exit(NULL);
}
```

As you can see, I didn't take any measures to guarantee the synchronization of different threads. The main reasons is that:

- Firstly, different threads in matrix multiplication actually will manipulate the same piece of data, but there is no conditions that restrict the threads when it can do some operation with the data and when it can't. So there won't appear some problem caused by the desynchronization of different threads.

  > *One of the most classic example of thread-desynchronization is the **Ticketing issue**. The ticket counter only sell tickets if there are still some available tickets. If the synchronization of different counters, i.e. different threads, isn't guaranteed, some problem will happen. When there is only one ticket left, and two counters find there is still an available ticket at the same time, they will sell this ticket one time respectively, causing the number of remaining ticket is negative.*

- Besides that, the efficiency of matrix multiplication will slump if making different threads synchronized(by using mutex lock, semaphore and so on). It is because that synchronization will force the threads to wait if there is already a thread that is manipulating the piece of data. But actually it don't need to wait, the only thing the thread should do is to calculate out the data and add it onto the result matrix.

## 1.2 `util.c`

```
char prefix[64] = "out-";
extern TIMER timer;
extern char suffix[];

/**
 * @brief <p>This struct is used to passes data and stores the result in multi-threads.
</p>
 *        <p>mat_A and mat_B is the two matrices which are multiplied together.</p>
 *        <p>resMat is the result of matrices multiplication</p>
 *        <p>beginRowIndex is the index of the first row's index that is calculated in
 *        current thread.</p>
 *        <p>rowNumToCal is the row's number of the current thread to calculate.</p>
 */
typedef struct RESULT_SET {
    MATRIX *resMat;
    MATRIX *mat_A;
    MATRIX *mat_B;
    size_t beginRowIndex;
    size_t rowNumToCal;
} ResultSet;

/**
 * @brief <p>Copy v1_ and v2_ 's data to v1 and v2.</p>
 * @param v1_ The pointer of a certain row in matrix.
 * @param v2_ The pointer of a certain column's first element in matrix.
 * @param v1 Aligned dynamic array used in SIMD operation(row)
 * @param v2 Aligned dynamic array used in SIMD operation(column)
 * @param n Size of matrix.
 */
inline void alignedCopy(const float *v1_, const float *v2_, float *v1, float *v2, size_t
n) {
    for (size_t i = 0; i < n; i++) {
        v1[i] = v1_[i];
        v2[i] = v2_[i * n];
    }
}
```

```
35
36  /**
37   * @brief <p>Allocate memory in heap for an 2D array with specified
38   *           size of row and column and then initialize the data.</p>
39   */
40  float **allocate2DArray(size_t row, size_t col) {
41      float **res = (float **) malloc(row * sizeof(float *));
42      float *p = (float *) malloc(row * col * sizeof(float));
43      if (res && p) {
44          for (size_t i = 0; i < row; i++) {
45              res[i] = p + i * col;
46          }
47      }
48
49      //Initialization
50      for (size_t i = 0; i < row; i++) {
51          for (size_t j = 0; j < col; j++) {
52              res[i][j] = 0.0f;
53          }
54      }
55      return res;
56  }
57
58  void free2DArray(float **arr) {
59      free(arr[0]);
60      free(arr);
61  }
62
63  void print2DArray(float **arr, size_t row, size_t col) {
64      for (size_t i = 0; i < row; i++) {
65          for (size_t j = 0; j < col; j++) {
66              printf("%-4.1f ", arr[i][j]);
67          }
68          printf("\n");
69      }
70  }
71
72  /**
73   * @brief <p>Get file's path according to its name(the file should be
74   *           stored in folder txt).</p>
75   */
76  char *getFilePath(const char *root, char *filename) {
77      char *path = (char *) malloc(sizeof(char) * strlen(root));
78      strcpy(path, root);
79      strcat(path, filename);
80      return path;
81  }
82
83  /**
84   * @brief <p>Get path of the output file.
85   *         eg. "out-2048x2048.txt"(stored in bin)</p>
86   */
87  char *getOuFileName(size_t row, size_t col) {
88      char *name = (char *) malloc(sizeof(char) * 64);
89      memcpy(name, prefix, sizeof(char) * 64);
90      char buffer[64] = {};
91      sprintf(buffer, "%ld", row);
92      strcat(name, buffer);
93      strcat(name, "x");
94      sprintf(buffer, "%ld", col);
95      strcat(name, buffer);
96      strcat(name, suffix);
97      return name;
98  }
99
```

```
/**
 * @brief <p>Write the data of arr into output file.</p>
 */
void writeMatrix(float **arr, size_t row, size_t col, char *outFileName) {
    printf("Start to write matrix into %s. \n", outFileName);

    TIME_START
    FILE *fp;
    if ((fp = fopen(outFileName, "w")) == NULL) {
        printf("Fail to open or create output file -> %s. Exit.", outFileName);
        exit(0);
    }

    for (size_t i = 0; i < row; i++) {
        for (size_t j = 0; j < col; j++) {
            char *buffer = (char *) malloc(sizeof(char) * 32);
            sprintf(buffer, "%.1f", arr[i][j]);
            fputs(buffer, fp);
            if (j < col - 1) fputs(" ", fp);
            else fputs("\r\n", fp);
            free(buffer);
        }
    }
    TIME_END("Writing matrix into file completed. ")

    fclose(fp);
}

/**
 * @brief <p>Allocate memory in heap for struct ResultSet</p>
 */
ResultSet *prepareResultSet(MATRIX *resMat, MATRIX *mat_A, MATRIX *mat_B, size_t
beginRowIndex, size_t rowNumToCal) {
    ResultSet *set = (ResultSet *) malloc(sizeof(ResultSet));
    set->resMat = resMat;
    set->mat_A = mat_A;
    set->mat_B = mat_B;
    set->beginRowIndex = beginRowIndex;
    set->rowNumToCal = rowNumToCal;
    return set;
}
```

### 1.3 `main.c`

```
char root[64] = "../txt/";
char suffix[64] = ".txt";
extern bool use_avx2;

int main(int argc, char **argv) {
    char filename0[64] = {};
    char filename1[64] = {};

    if (argc == 1) {
        while (TRUE) {
            printf("Please input names of two file or [Q] to quit: \n");
            scanf("%s", filename0);
            if (filename0[0] == 'Q' || filename0[0] == 'q') {
                printf("Program exits.\n");
                exit(0);
            }
            scanf("%s", filename1);

            MATRIX mat_A;
```

```
            Matrix(&mat_A, getFilePath(root, filename0));
            MATRIX mat_B;
            Matrix(&mat_B, getFilePath(root, filename1));

            char methodID;
            printf("\nWhich method do you want to use to calculate multiplication?\n");
            printf("    1 -->> Multiplication in order i->j->k.\n");
            printf("    2 -->> Multiplication in order i->k->j.\n");
            printf("    3 -->> Multiplication using Strassen.\n");
            printf("    4 -->> Multiplication using multi-thread completed.\n");
            scanf("\n%c", &methodID);

            switch (methodID) {
                case '1':
                    mul_ijk(&mat_A, &mat_B);
                    break;
                case '2':
                    mul_ikj(&mat_A, &mat_B);
                    break;
                case '3':
                    strassen(&mat_A, &mat_B);
                    break;
                case '4':
                    matMul_Threads(&mat_A, &mat_B);
                    break;
                default:
                    printf("Wrong ID. Exit.\n");
                    exit(0);
            }
        }
    } else {
        if (argc < 4) {
            puts("Not enough parameter. Exit.");
            exit(100);
        }

        char *ptr;
        if (strtol(argv[3], &ptr, 10)) {
            use_avx2 = TRUE;
        }

        strncpy(filename0, argv[1], 64);
        strncpy(filename1, argv[2], 64);
    }

    MATRIX mat_A;
    Matrix(&mat_A, getFilePath(root, filename0));
    MATRIX mat_B;
    Matrix(&mat_B, getFilePath(root, filename1));

    mul_ijk(&mat_A, &mat_B);

    mul_ikj(&mat_A, &mat_B);

    strassen(&mat_A, &mat_B);

    matMul_Threads(&mat_A, &mat_B);

    if (use_avx2 == TRUE) {
        printf("All multiplication above use AVX2 to speed up calculation.\n");
    } else {
        printf("All multiplication above don't use AVX2 to speed up calculation.\n");
    }

    return 0;
```

```
85  }
```

I do a little improvement of user experience. User can choose to input names of two files from command line, or just input the names after the program start to run.

# 3. Result & Verification

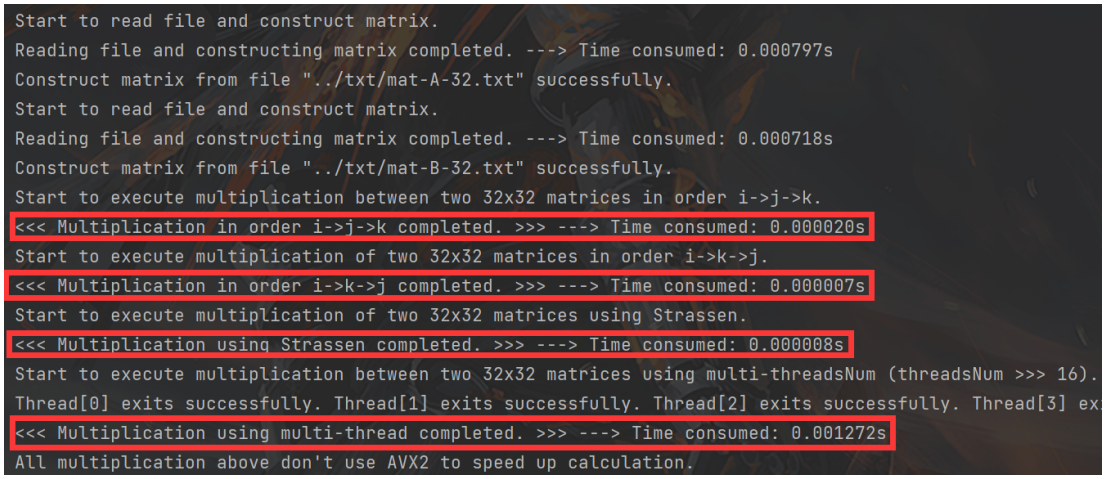
---

> *Environment:*
>
> *OS: Windows 10*
>
> *CPU: AMD Ryzen 9 5900HS with Radeon Graphics @ 3.30 GHz*
>
> *RAM :16.0 GB (15.4 GB avaliable)*
>
> *GPU: NVIDIA GeForce RTX 3060 Laptop GPU*
>
> ---
>
> *Notes: The text files that store the data of matrices should be put into the floder **txt**. And the output executable file is in the **bin** floder.*

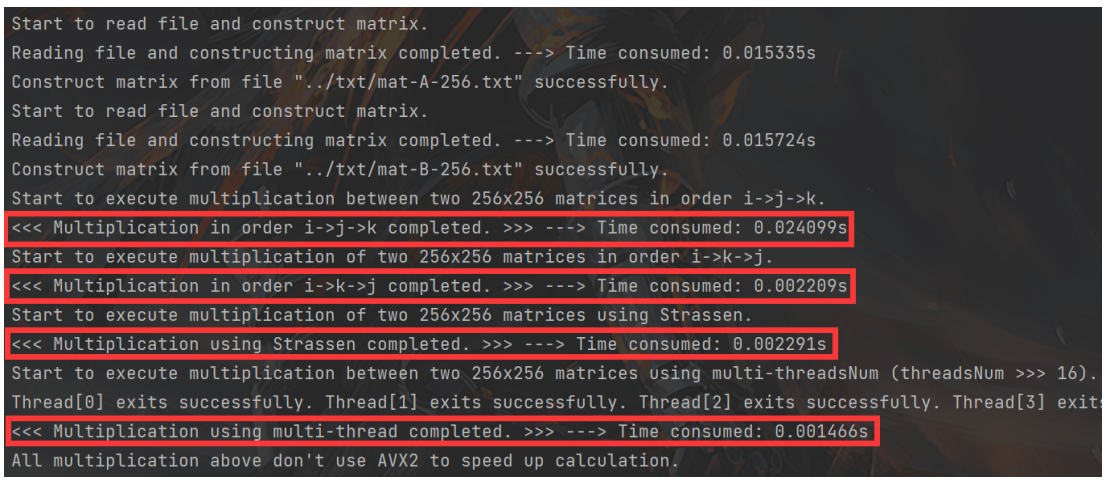
## 3.1 Speed

### 3.1.1 Result not using **AVX2** (with O3 optimization of **GCC** compiler )

- 32x32:




- 256x256:




- 2048x2048:

```
C:\Windows\system32\wsl.exe --distribution Ubuntu-20.04 --exec /bin/sh -c "cd '/mnt/d/OneDrive - Office/Fi
Start to read file and construct matrix.
Reading file and constructing matrix completed. ---> Time consumed: 0.892689s
Construct matrix from file "../txt/mat-A-2048.txt" successfully.
Start to read file and construct matrix.
Reading file and constructing matrix completed. ---> Time consumed: 0.906549s
Construct matrix from file "../txt/mat-B-2048.txt" successfully.
Start to execute multiplication between two 2048x2048 matrices in order i->j->k.
<<< Multiplication in order i->j->k completed. >>> ---> Time consumed: 59.607111s
Start to execute multiplication of two 2048x2048 matrices in order i->k->j.
<<< Multiplication in order i->k->j completed. >>> ---> Time consumed: 1.567758s
Start to execute multiplication of two 2048x2048 matrices using Strassen.
<<< Multiplication using Strassen completed. >>> ---> Time consumed: 0.990255s
Start to execute multiplication between two 2048x2048 matrices using multi-threadsNum (threadsNum >>> 16).
Thread[0] exits successfully. Thread[1] exits successfully. Thread[2] exits successfully. Thread[3] exits
<<< Multiplication using multi-thread completed. >>> ---> Time consumed: 0.160539s
All multiplication above don't use AVX2 to speed up calculation.
```

| METHOD/(S) | 32X32 | 256X256 | 2048X2048 |
|---|---|---|---|
| i->j->k | 0.000021 | 0.024151 | 59.523100 |
| i->k->j | 0.000007 | 0.002213 | 1.593960 |
| Strassen | 0.000008 | 0.002227 | 0.966595 |
| multi-thread | 0.001839 | 0.001778 | 0.178024 |

*Table 1. Average time consumption of matrices multiplication with different mehtods*

### 3.1.2 Somting abnormal in multiplication using multi-thread

In the multiplication of 32x32 mtrices, the efficiency of simple matrix multiplication is higher than that of multi-thread, which is abnormal. After I test another several times, the result is same.

| THREAD'S NUM/(S) | 32X32 | 256X256 | 2048X2048 |
|---|---|---|---|
| 2 | 0.0001540 | 0.0013426 | 0.942142 |
| 4 | 0.0003250 | 0.0008922 | 0.533527 |
| 8 | 0.0006944 | 0.0009334 | 0.307052 |
| 16 | 0.0009616 | 0.0009434 | 0.187013 |

*Table 2. Average time consumption of matrices multiplication with different thread's number*

I believe the main reason is that allocating memory for threads will also cause time consumption, sometimes even very large. When the size of matrices isn't large, the threads' allocation make the multiplication cost redundant time consumption and thus slow down.

Besides that, the efficiency won't go up anymore when threads' number is more than 16. This is because that CPU of this desktop supports 16 threads at most.

*Figure 4. Details of the current CPU's specification*

### 3.1.3 Result using **AVX2** (with O3 optimization of **GCC** compiler )

> *To highlight the point of the problem, here matrix multiplication using multi-thread is not included.*

- 32x32:

```
Start to read file and construct matrix.
Reading file and constructing matrix completed. ---> Time consumed: 0.000663s
Construct matrix from file "../txt/mat-A-32.txt" successfully.
Start to read file and construct matrix.
Reading file and constructing matrix completed. ---> Time consumed: 0.000580s
Construct matrix from file "../txt/mat-B-32.txt" successfully.
Start to execute multiplication between two 32x32 matrices in order i->j->k.
<<< Multiplication in order i->j->k completed. >>> ---> Time consumed: 0.000126s
Start to execute multiplication of two 32x32 matrices in order i->k->j.
<<< Multiplication in order i->k->j completed. >>> ---> Time consumed: 0.000015s
Start to execute multiplication of two 32x32 matrices using Strassen.
<<< Multiplication using Strassen completed. >>> ---> Time consumed: 0.000100s
All multiplication above use AVX2 to speed up calculation.
```

- 256x256:

```
Start to read file and construct matrix.
Reading file and constructing matrix completed. ---> Time consumed: 0.012677s
Construct matrix from file "../txt/mat-A-256.txt" successfully.
Start to read file and construct matrix.
Reading file and constructing matrix completed. ---> Time consumed: 0.013731s
Construct matrix from file "../txt/mat-B-256.txt" successfully.
Start to execute multiplication between two 256x256 matrices in order i->j->k.
<<< Multiplication in order i->j->k completed. >>> ---> Time consumed: 0.041226s
Start to execute multiplication of two 256x256 matrices in order i->k->j.
<<< Multiplication in order i->k->j completed. >>> ---> Time consumed: 0.001776s
Start to execute multiplication of two 256x256 matrices using Strassen.
<<< Multiplication using Strassen completed. >>> ---> Time consumed: 0.042405s
All multiplication above use AVX2 to speed up calculation.
```

- 2048x2048:

```
Start to read file and construct matrix.
Reading file and constructing matrix completed. ---> Time consumed: 0.881794s
Construct matrix from file "../txt/mat-A-2048.txt" successfully.
Start to read file and construct matrix.
Reading file and constructing matrix completed. ---> Time consumed: 0.891814s
Construct matrix from file "../txt/mat-B-2048.txt" successfully.
Start to execute multiplication between two 2048x2048 matrices in order i->j->k.
<<< Multiplication in order i->j->k completed. >>> ---> Time consumed: 66.959151s
Start to execute multiplication of two 2048x2048 matrices in order i->k->j.
<<< Multiplication in order i->k->j completed. >>> ---> Time consumed: 1.695450s
Start to execute multiplication of two 2048x2048 matrices using Strassen.
<<< Multiplication using Strassen completed. >>> ---> Time consumed: 18.725813s
All multiplication above use AVX2 to speed up calculation.
```

It is strange that all matrix multiplication using **AVX2** is slower than that without **AVX2**. I'm inclined to believe this is also caused by memory allocation. **AVX2** constrains that the address of **float** data, which will be loaded in the 256-bit-width register, must be aligned. If it doesn't, there will be `Segamentation Fault` error during runtime. Thus before loading the data into registers, you must make the address aligned (make it multiple of 32, since you need to load 8 **float** data and every occupies 4 bytes). I use `aligned_alloc(size_t alignment, size_t size)` to make address aligned. This function will allocate memory which is aligned with `alignment` (bytes) and returns its address. Then I will copy 8 data in a row and another 8 data in a column. This is where redundant time consumption comes from. To uss **AVX2**, I must make address aligned. To make address

aligned, I must copy data one by one, which means I still need to access every element in the two matrices before calculating (to copy data from origin address to aligned address) and then access them all again (during the process of loading data from aligned address in heap into registers).

But I still has some question. **AVX2** can manipulate 8 **float** data at one time, thus its efficiency is suppoesd to be 8 times faster as much as simple data manipulation. Assessing roughly, the efficiency is half of simple matrix multiplication because of double access to the same piece of data, but then is boosted to 8 times the original since 8 **float** data are manipulated at the same time. The final efficiency calculated by this way is $\frac{1}{2} \times 8 = 4$ times faster as much as simple matrix multiplication. However,the fact is that the multiplication is slown down anyway. 😭

Another possible reason is that there are some problem in my codes, such as unsuitable use of functions or redundant statement. Till now, I have tried many ways to fix this bug, but I'm still no closer to the solution. Maybe you can offer me some help.

## 3.2 Data error

### 3.2.1 Comparison with simple matrix multiplication's result

> *As I have mentioned before, there are not much interference during the mtrix multiplication in order **i->j->k**, thus it can be taken as the standard result to some extent.*

The way I compare the error is to count the difference between each element and the corresponding standard result's element. Then take the difference's square and add them all. The more its sum is, the more error its matrix multiplication causes.

| SIZE/DIFFERENCE_SUM | IKJ | STRASSEN | MULTI-THREADS |
|---|---|---|---|
| 32x32 | 0.00 | 0.00 | 0.00 |
| 256x256 | 0.00 | 0.00 | 0.00 |
| 2048x2048 | 0.00 | 12.25 | 0.00 |

*Table 3. Difference's square sum(with O3 optimization on)*

As you can see, every difference's square sum is zero except 2048 matrix multiplication using **Strassen**. I guess one of the reasons caue its sum isn't zero is recursion. When the recursive threshold is relatively small, which means the recursive ground is deep, there will be extremely large data being pushed into stack. Thus it can't avoid error and manipulate so many data at the same time.

| RECURSIVE THRESHOLD | DIFFERENCE_SUM |
|---|---|
| 1024 | 42.25 |
| 512 | 9.00 |
| 256 | 12.25 |
| 128 | 1444.00 |
| 64 | 3906.25 |
| 32 | 361.00 |

*Table 4. Difference's square sum with different recursive threshlod(with O3 optimization on)*

As the recursive ground going deeper, the overall treand of difference's square sum is going up, whih indicates that the recursive depth will affect the result's accuracy to some extent. But I don't know why the recursive threshold is 32, its sum is less than the previous one till now.

Besides the factor of recursive depth, I think it another possible reason is **GCC O3 Optimization**. The O3 optimization will do many things to speed up the program, such as optimize memory operation's order and loop. When the program is complicated, the logic may be different after O3 optimization, compared with the original one. I turn off the O3 optimization and test anothe gruop.

| RECURSIVE THRESHOLD | DIFFERENCE_SUM |
|---|---|
| 1024 | 12.25 |
| 512 | 4.00 |
| 256 | 20.25 |
| 128 | 841.00 |
| 64 | 1892.25 |
| 32 | 256.00 |

*Table 5. Difference's square sum with different recursive threshlod(with O3 optimization off)*

As the table demonstrates, the sum is smaller than the sum in the previous table, which indicates that **GCC O3 Optimization** will also affect the reault's accuracy to some extent.

### 3.2.2 Comparison with OpenBLAS

With the help of **OpenBLAS**, we can count the difference's square sum with a more precise result matrix.

| SIZE/DIFFERENCE_SUM | IJK | IKJ | STRASSEN | MULTI-THREADS |
|---|---|---|---|---|
| 32x32 | 0.00 | 0.00 | 0.00 | 0.00 |
| 256x256 | 0.02 | 0.02 | 0.02 | 0.02 |
| 2048x2048 | 30.25 | 30.25 | 462.25 | 30.25 |

*Table 6. Difference's square sum with different size compared with **OpenBLAS**(with O3 optimization on)*

Take the average of the sum(divide by size's square)

| AVERAGE | IJK | IKJ | STRASSEN | MULTI-THREADS |
|---|---|---|---|---|
| 32x32 | 0 | 0 | 0 | 0 |
| 256x256 | $1.95313 \times 10^{-6}$ | $1.95313 \times 10^{-6}$ | $1.95313 \times 10^{-6}$ | $1.95313 \times 10^{-6}$ |
| 2048x2048 | $7.21216 \times 10^{-6}$ | $7.21216 \times 10^{-6}$ | $1.10209 \times 10^{-4}$ | $7.21216 \times 10^{-6}$ |

*Table 7. Average difference's square sum with different size compared with **OpenBLAS**(with O3 optimization on)*

As the table demonstrates, the average error of each element is overall less than or near $10^{-5}$. Thus the accuracy of my program is not too bad. And as the size of the matrices is going up, the error is going up, too. The main possible reason that leads to the error, compared with results calculated by **OpenBLAS**, is caused by **float** type data. The data stored in the form of **float** can't be precise absolutely. There will always be some tiny difference between the data stored in memory and the number you input.

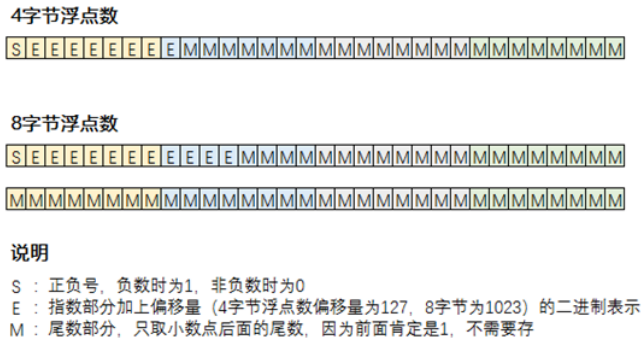



*Figure 5. Schematic of how floating-point number is stored in computer*

**OpenBLAS** may take some methods inside ints function to minimize the error caused by **float** during calculation. (the details of how **OpenBLAS** implements it may be too difficult for, so I didn't mention here)Thus there will be error between my results and **OpenBLAS**'s results. And as the size is going up, the **float** error accumulates.

# 4. Difficulties & Solutions

- **AVX2** doesn't work well in my program. And till now I am still confused with the reason. If **Pro. Yu** can offer me a little help, I would be really grateful.
- When I use `strcat()` and `strcpy()` to manipulate string in the process of write file, there will be always `Sementation Fault`. After testing many times, I found it may be caused by declaring the string as `char*`. So I declared all string as `char[]` later.

# 5. Summary and comparison with the last project

- In the last project, I only complete the definition of class Matrix, and the implemention of marix multiplication in **i->j->k** and **i->k->j**. But this time I tried my best to optimize the multiplication by using **Strassen's Algorithm**, **SIMD**, **GCC O3 Optimization** and **Multi-thread**. The program running time of 2048 matrix multiplication was reduced from nearly 1 minute to only **0.16s**. But in the last project, the fastest one cost 28.05s.
- I pay more attention to the accuracy of different ways to implement matrix multiplication. Then find out the accuracy of the results calculated by my program is fairly good within the testcases provided by the professor.
- The most important thing I learn from this project is that why you get a problem and you want to write codes to solve it. The very first thing you should do is not to start to write immediately, but think twice before implementing the codes. You should consider how to solve the problem in an elegant way. The first algorithm or solution is always the brute force one and the badest one. Taking one day to construct the algorithm or draw a mind map to let you have a clear mind during sloving the problem. After that you begin to write codes, your program will be improved, maybe very much.

# Reference


1. C++加速矩阵乘法的最简单方法 - 知乎 (zhihu.com) ↩

2. 详解矩阵乘法中的Strassen算法 - 知乎 (zhihu.com) ↩

3. C/C++：从基础语法到优化策略 - 南方科技大学 - 学堂在线 (xuetangx.com) ↩

4. SIMD指令集 - 知乎 (zhihu.com) ↩

5. gcc -O0 -O1 -O2 -O3 四级优化选项及每级分别做什么优化_Aikenlan的博客-CSDN博客 ↩