



# MATRIX-MATRIX-MULTIPLY ANALYSIS

SEQUENTIAL, THREAD, BLAS

Author

Xiu Zhou

1/27/2024

# Contents

1. GEMM Overview .....	4
1.1. GEMM introduction .....	4
1.1.1. Introduction .....	4
1.1.2. Math Definition .....	4
1.1.3. GEMM Interface (blas_gemm) .....	5
1.1.4. Common GEMM Implementations in Market.....	6
2. My GEMM Solution Focus Clarification.....	8
2.1. Sequential Algorithm Selection for Programs .....	8
2.1.1. Environment Clarification.....	8
2.1.2. Standard Algorithm Implementation and Test.....	8
2.1.3. Block Optimization Implementation and Test.....	11
2.1.3.1. Define Block Size .....	11
2.1.3.2. Algorithm Explanation.....	13
2.1.3.3. Algorithm FLOPs Calculation .....	14
2.1.3.4. Some Screenshots of Program Compiling .....	15
2.1.4. Transpose and Unrolled Optimization Implementation and Test .....	15
2.1.4.1. Algorithm Explanation.....	15
2.1.4.2. Algorithm FLOPs Calculation .....	16
2.1.4.3. Some Screenshots of Program Compiling .....	17
2.1.5. Comparing and Analysis of Three algorithms .....	17
2.1.5.1. Raw Data Files .....	17
2.1.5.2. Plot of Tree Algorithms .....	17
2.1.5.3. Analysis of Plot .....	18
2.1.6. Sequential Algorithm Selection Conclusion .....	18
2.2. Thread solution Algorithm Selection for Programs.....	18
2.2.1. Algorithm Implementation Explanation Based on Sequential Algorithm .....	18
2.2.1.1. multySequentialUnrolledSingleThread() Function.....	19
2.2.1.2. multySequentialUnrolledMultiThread() Function.....	19
2.2.2. Algorithm FLOPs Calculation .....	20
2.2.3. Some Screenshots of Program Compiling .....	21
3. Experiment Results Correctness Explanation.....	22
3.1. Theoretical Method for Getting Correct Results.....	22
3.2. calculateResidual() Function .....	22
3.3. Specific Application in Sequential Solution.....	23
3.3.1. In main() Part.....	23

3.3.2.	Screen Shots of Partial Result Screen Output .....	24
3.3.3.	All Residuals .....	25
3.4.	Specific Application in Thread Solution.....	25
3.4.1.	In main() Part.....	25
3.4.2.	Screen Shots of Partial Result Screen Output .....	25
3.4.3.	All Residuals .....	26
4.	Sequential-Thread-BLAS FLOPs Comparing Experiments (in my computer) .....	27
4.1.	Environment Clarification.....	27
4.2.	Sequential Solution Based on Transpose & Unrolled Algorithm.....	27
4.3.	Thread Solution Based on Transpose & Unrolled Sequential Solution .....	28
4.4.	BLAS Solution Based on OpenBLAS Function.....	28
5.	Sequential-Thread-BLAS FLOPs Comparing Analysis (in my computer).....	29
5.1.	Raw Data Files .....	29
5.2.	BLAS-Sequential-Thread FLOPs Plot-1.....	29
5.3.	Plot-1 Analysis .....	30
5.3.1.	Why Sequential Is the Slowest .....	30
5.3.2.	Why Threaded Is Better than Sequential .....	31
5.3.3.	Why BLAS Is So Much Faster .....	31
5.4.	BLAS-Sequential-Thread FLOPs Plot-2.....	32
5.5.	Plot-2 Combining Plot-1 Analysis .....	33
5.5.1.	BLAS Sharp Climbing – Fast Climbing - Stable .....	33
5.5.2.	Sequential Solution Slightly Dropping as Size Increasing .....	33
5.5.3.	Thread Solution Sharp Climbing - Sharp Dropping - Slight Dropping.....	33
6.	My Computer Information (CPU, System, IDE) .....	34
6.1.	CPU Specifications of My Computer .....	34
6.2.	Operating System of My Computer.....	34
6.3.	Compiler of Programs for BLAS-Sequential-Thread FLOPs (in my Computer) .....	34
6.4.	IDE .....	34
7.	Calculate Theoretical Peak Performance .....	34
7.1.	My CPU Model Details .....	34
7.2.	Formula .....	35
7.3.	Theoretical Peak Performance (FLOPS) of my computer .....	35
7.4.	Compare Solutions FLOPs (in my computer) with Peak Performance .....	36
7.4.1.	Theoretical Peak Performance .....	36
7.4.2.	Actual Performance Results .....	36
7.4.3.	Analysis.....	36

8.	Sequential-Thread-BLAS FLOPs Comparing Experiments (in different computers) .....	38
8.1.	Environment Clarification (Compiler – Computer – Test Case Design) .....	38
8.2.	Programs of Conducting Experiments.....	39
8.3.	Partial Output Screenshots on My Computer .....	39
8.4.	Partial Output Screenshots on Another Computer .....	42
9.	Sequential-Thread-BLAS FLOPs Comparing and Analysis (in different computers) .....	45
9.1.	Raw Data Files .....	45
9.2.	BLAS-Sequential-Thread FLOPs Plot-11.....	45
9.3.	BLAS-Sequential-Thread FLOPs Plot-22.....	46
9.4.	Plot-11 and Plot-22 Analysis.....	46
9.4.1.	Finding form Plot-11 and Plot-22 .....	46
9.4.2.	Explanation for Finding .....	46
10.	Reference .....	47

# 1. GEMM Overview

## 1.1. GEMM introduction

### 1.1.1. Introduction

GEMM stands for General Matrix Multiply, and it refers to a fundamental operation in linear algebra where two matrices are multiplied to produce a third matrix. The general form of matrix multiplication involves taking the dot product of each row of the first matrix with each column of the second matrix and summing up the results to get the corresponding element in the resulting matrix.

The GEMM operation is not only a fundamental building block in linear algebra but also plays a crucial role in various scientific and engineering applications, such as machine learning, signal processing, computer graphics, and numerical simulations. Efficient implementations of GEMM are essential for optimizing the performance of applications that involve large-scale matrix operations. Modern hardware architectures often include specialized instructions and libraries to accelerate GEMM computations. Libraries like BLAS (Basic Linear Algebra Subprograms) and cuBLAS (NVIDIA's CUDA Basic Linear Algebra Subprograms) provide highly optimized implementations of GEMM and related operations.

GEMM is defined as the operation  $C = \alpha AB + \beta C$ , with A and B as matrix inputs,  $\alpha$  and  $\beta$  as scalar inputs, and C as a pre-existing matrix which is overwritten by the output. A plain matrix product AB is a GEMM with  $\alpha$  equal to one and  $\beta$  equal to zero [1].

### 1.1.2. Math Definition

If **A** is an  $m \times n$  matrix and **B** is an  $n \times p$  matrix,

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

the *matrix product*  $\mathbf{C} = \mathbf{AB}$  (denoted without multiplication signs or dots) is defined to be the  $m \times p$  matrix

$$\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix}$$

such that

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^n a_{ik}b_{kj},$$

for  $i = 1, \dots, m$  and  $j = 1, \dots, p$ .

That is, the entry  $C_{ij}$  of the product is obtained by multiplying term-by-term the entries of the  $i$ th row of **A** and the  $j$ th column of **B**, and summing these  $n$  products. In other words,  $C_{ij}$  is the dot product of the  $i$ th row of **A** and the  $j$ th column of **B** [2].

Therefore, **AB** can also be written as

$$\mathbf{C} = \begin{pmatrix} a_{11}b_{11} + \cdots + a_{1n}b_{n1} & a_{11}b_{12} + \cdots + a_{1n}b_{n2} & \cdots & a_{11}b_{1p} + \cdots + a_{1n}b_{np} \\ a_{21}b_{11} + \cdots + a_{2n}b_{n1} & a_{21}b_{12} + \cdots + a_{2n}b_{n2} & \cdots & a_{21}b_{1p} + \cdots + a_{2n}b_{np} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{11} + \cdots + a_{mn}b_{n1} & a_{m1}b_{12} + \cdots + a_{mn}b_{n2} & \cdots & a_{m1}b_{1p} + \cdots + a_{mn}b_{np} \end{pmatrix}$$

### 1.1.3. GEMM Interface (blas\_gemm)

```
void blas_gemm(
    const enum Transpose transA,
    const enum Transpose transB,
    const int M,
    const int N,
    const int K,
    const double alpha,
    const double* A,
    const int lda,
    const double* B,
    const int ldb,
    const double beta,
    double* C,
    const int ldc
```

);

### **Interface introduction:**

#### **1) transA and transB:**

These parameters determine whether to transpose the matrices A and B before performing the matrix multiplication.

Possible values:

- NoTrans: Do not transpose.
- Trans: Transpose the matrix.
- ConjTrans: Conjugate transpose (commonly used in complex arithmetic).

#### **2) M, N, and K:**

These parameters define the dimensions of the matrices involved in the operation.

- M: Number of rows in matrix C.
- N: Number of columns in matrix C.
- K: Common dimension shared by matrices A and B.

#### **3) alpha and beta:**

These are scalar coefficients used in the matrix multiplication.

- alpha: Scalar multiplier for the product of A and B.
- beta: Scalar multiplier for matrix C.

#### **4) A, B, and C:**

These are pointers to the matrices involved in the operation.

- A: Pointer to matrix A.
- B: Pointer to matrix B.
- C: Pointer to matrix C.

#### **5) lda, ldb, and ldc:**

- These parameters represent the leading dimensions of matrices A, B, and C.
- The leading dimension is the number of elements between successive rows (for A and B) or columns (for C).
- This is crucial for handling submatrices efficiently.

### **1.1.4. Common GEMM Implementations in Market**

There are several optimized GEMM implementations existing in the market, and they are often part of high-performance numerical libraries. Some notable implementations include:

**1) BLAS (Basic Linear Algebra Subprograms):**

BLAS is a widely-used collection of routines for linear algebra operations, including GEMM.

Different implementations of BLAS exist, and they are optimized for various hardware architectures.

Examples include OpenBLAS, Intel MKL (Math Kernel Library), and ATLAS (Automatically Tuned Linear Algebra Software).

**2) OpenBLAS:**

OpenBLAS is an open-source implementation of BLAS that aims to provide high-performance linear algebra routines.

It is designed to be architecture-agnostic and can take advantage of hardware-specific optimizations.

**3) Intel MKL (Math Kernel Library):**

Intel MKL is a library of optimized mathematical routines for Intel processors.

It includes highly optimized GEMM implementations that leverage Intel's advanced hardware features.

**4) cuBLAS (CUDA Basic Linear Algebra Subprograms):**

cuBLAS is part of the NVIDIA CUDA toolkit and provides BLAS routines optimized for NVIDIA GPUs.

It allows developers to offload GEMM computations to the GPU for parallel processing.

**5) ARM Performance Libraries:**

ARM provides optimized libraries for various mathematical operations, including GEMM, targeted for ARM architecture-based processors.

**6) OpenMP and MPI Parallel Implementations:**

Many GEMM implementations leverage parallelization techniques such as OpenMP (for shared-memory systems) or MPI (Message Passing Interface) for distributed memory systems.

**7) Custom Hardware Accelerators:**

Some applications use custom hardware accelerators, like FPGAs (Field-Programmable Gate Arrays) or TPUs (Tensor Processing Units), to perform GEMM operations efficiently.

My GEMM Solutions Implementation



## 2. My GEMM Solution Focus Clarification

In this report, it focusses on implementation of plain matrix product ( $C = A * B$ ), which is enough to reveal the different performance caused by algorithms or hardware.

### 2.1. Sequential Algorithm Selection for Programs

#### 2.1.1. Environment Clarification

In order to conducting tests under the same environment, there are some measures to be adopted.

##### 1) About compiler

Use Visual Studio to compile all the programs.

##### 2) About test cases

All the algorithm tests use the same cases generated by `generateA()` and `generateB()`. The case's dimension size increases from 10 to 1000 with step 10.

```
87 // Function to generate and return a vector of random double values with 0 decimals
88 std::vector<double> generateA(int size) {
89     std::vector<double> vec(size);
90     for (int i = 0; i < size; ++i) {
91         vec[i] = 1.0000 + i % 3;
92     }
93     return vec;
94 }
95
96 // Function to generate and return a vector of value 0.0001
97 std::vector<double> generateB(int size) {
98     std::vector<double> vec(size);
99     for (int i = 0; i < size; ++i) {
100         vec[i] = 0.0001;
101     }
102     return vec;
103 }
```

Fig 1, `generateA()` and `generateB()` functions

##### 3) About results correction

In every case, calculate residual between my solution and `cblas_dgemm` (BLAS) to verify the results correction. Besides, provide give a boolean to indicate whether the residual is within a tolerance ( $1e-6$ ).

##### 4) About running computer and network

All the tests are conducted in the same computer and network.

#### 2.1.2. Standard Algorithm Implementation and Test

##### 2.1.2.1. Algorithm Explanation

```

35 // Function to do Matrix-Matrix multiplication with standard way
36 void multiplySequential(const vector<double>& A, const vector<double>& B, vector<double>& C, int dimension) {
37     for (int i = 0; i < dimension; ++i) {
38         for (int j = 0; j < dimension; ++j) {
39             int idx = i * dimension + j;
40             double sum = 0.0;
41
42             for (int k = 0; k < dimension; ++k) {
43                 sum += A[i * dimension + k] * B[k * dimension + j];
44             }
45
46             C[idx] = sum;
47         }
48     }
49 }

```

Fig 2, multiplySequential( ) function

The function uses two nested loops to iterate over each row (i) of Matrix A and each column (j) of the resulting matrix C.

The index idx is calculated as the linear index corresponding to the current position in the resulting matrix.

### 1) Outer Loop (i and j):

The function uses two nested loops to iterate over each row (i) of Matrix A and each column (j) of the resulting matrix C.

The index idx is calculated as the linear index corresponding to the current position in the resulting matrix.

### 2) Inner Loop (k):

The innermost loop iterates over the common dimension of the matrices, which is the number of columns in A or the number of rows in B. This dimension must be the same for the matrices to be multiplied.

The variable sum is initialized to zero for each element in the resulting matrix.

The loop calculates the dot product of the i-th row of matrix A and the j-th column of matrix B by iterating over the elements of the row and column and accumulating the products.

### 3) Calculation and Storage:

The calculated sum represents the value of the element at position (i, j) in the resulting matrix C.

The final result is stored in the vector C at the corresponding index idx.

#### 2.1.2.2. Algorithm FLOPs Calculation

```

81     int rows = 0;
82     for (int dimension = 10; dimension <= 1000; dimension += 10) {
83         vector<double> A = generateA(dimension * dimension);
84         vector<double> B = generateB(dimension * dimension);
85         vector<double> C_Blas(dimension * dimension, 0.0);
86         vector<double> C_My(dimension * dimension, 0.0);
87
88         rows = dimension;
89
90         outputFile << dimension << " ";
91
92         auto startTime = chrono::high_resolution_clock::now();
93         cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, dimension, dimension, dimension, 1.0,
94             A.data(), dimension, B.data(), dimension, 0.0, C_Blas.data(), dimension);
95         auto endTime = chrono::high_resolution_clock::now();
96         chrono::duration<double, milli> elapsedMilliseconds = endTime - startTime;
97         double flops = (2.0 * dimension * dimension * dimension) / (elapsedMilliseconds.count() / 1000.0);
98         outputFile << flops << " ";
99
100        startTime = chrono::high_resolution_clock::now();
101        multiplySequential(A, B, C_My, dimension);
102        endTime = chrono::high_resolution_clock::now();
103        chrono::duration<double, milli> elapsedMilliseconds1 = endTime - startTime;
104        double flops1 = (2.0 * dimension * dimension * dimension) / (elapsedMilliseconds1.count() / 1000.0);
105        outputFile << flops1 << " ";
106
107        double residual = calculateResidual(C_Blas, C_My);
108        outputFile << residual << " ";
109
110        bool withinTolerance = (residual < 1e-6);
111        outputFile << withinTolerance << "\n";

```

**Fig 3,** Algorithm FLOPs calculation module

1) Matrix Initialization:

Iterates over matrix dimensions from 10 to 1000 with increments of 10.

Generates random matrices A and B of the specified dimension.

2) BLAS (Basic Linear Algebra Subprograms) Matrix Multiplication:

Uses the cblas\_dgemm function (BLAS routine) to perform matrix multiplication.

Measures the execution time and calculates the floating-point operations per second (FLOPS) achieved during the operation.

Stores the result in vector C\_Blas.

3) Standard Sequential Matrix Multiplication:

Uses the multiplySequential() function to perform matrix multiplication.

Measures the execution time and calculates the FLOPS achieved during the operation.

Stores the result in vector C\_My.

4) Residual Calculation:

Calculates the residual (difference) between the results obtained from BLAS and the custom multiplication.

5) Output to File:

Writes the dimension, FLOPS for BLAS, FLOPS for standard sequential multiplication, residual, and a boolean indicating whether the residual is within a tolerance (1e-6) to an output file.

### 2.1.2.3. Screenshots of Program Compiling

```
CASE DIMENSION: 1000:

BLAS Results:
Time of BLAS: 38.1298
FLOPs of BLAS: 5.24524e+10

Standard Sequential Results:

Time of Standard Sequential: 18691.9
FLOPs of Standard Sequential: 1.06998e+08
Residual of Standard Sequential: 7.6828e-13
Is results match within tolerance? YES

C:\Users\xiumary\source\repos\css535\program\Program1A\x64\Debug\Program1A.exe (process 34704) exited with code 0.
```

Fig 4, Screenshot 2 of Standard Sequential solution

## 2.1.3. Block Optimization Implementation and Test

### 2.1.3.1. Define Block Size

#### 1) Formula to calculate block size

The blocked algorithm has computational intensity  $q \approx b$ , so the larger the block size, the more efficient our algorithm will be. However, there is a limit, which is all three blocks from A,B,C must fit in fast memory (cache 1), so we cannot make these blocks arbitrarily large[3]:

Assume your fast memory has size  $M_{fast}$ :  $3b^2 \leq M_{fast}$ , so  $q \approx b \leq (M_{fast}/3)^{1/2}$

My computer's L1\_CACHE\_SIZE is 256K. So, the block size should be less than 512 because of  $\text{sqrt}(L1\_CACHE\_SIZE / 3.0) = \text{sqrt}(256 * 1024 / 3.0) = 512$ .

#### 2) Tests for finding block size

Conclusion: After testing different block sizes, block size = 64 brings a little bit better efficient in my computer. So, the latter performance comparing analysis will use the results of static block sequential with 64 block size.

Here are some screenshots of different block size tests:

```
Microsoft Visual Studio Debug Console
CASE DIMENSION: 1000:
Marix A:
Marix B:
  BLAS Results:
    Time of BLAS: 72.7654
    FLOPs of BLAS: 2.74856e+10
  Static Block Sequential Results:
    Time of Static Block Sequential: 25499.8
    FLOPs of Static Block Sequential: 7.84321e+07
    Residual of Static Block Sequential: 7.68161e-10
    Is results match within tolerance? YES
```

Fig 5, Screenshot bock size = 32

```
Microsoft Visual Studio Debug Console
CASE DIMENSION: 1000:
Marix A:
Marix B:
  BLAS Results:
    Time of BLAS: 27.8535
    FLOPs of BLAS: 7.18043e+10
  Static Block Sequential Results:
    Time of Static Block Sequential: 24424.6
    FLOPs of Static Block Sequential: 8.18847e+07
    Residual of Static Block Sequential: 7.68161e-10
    Is results match within tolerance? YES
```

Fig 6, Screenshot bock size = 64

```
Microsoft Visual Studio Debug Console
CASE DIMENSION: 1000:
Marix A:
Marix B:
  BLAS Results:
    Time of BLAS: 31.1311
    FLOPs of BLAS: 6.42444e+10
  Static Block Sequential Results:
    Time of Static Block Sequential: 24549.4
    FLOPs of Static Block Sequential: 8.14684e+07
    Residual of Static Block Sequential: 7.68161e-10
    Is results match within tolerance? YES
```

Fig 7, Screenshot bock size = 128

```

Microsoft Visual Studio Debug Console
CASE DIMENSION: 1000:
Marix A:
Marix B:
  BLAS Results:
  Time of BLAS: 38.7488
  FLOPs of BLAS: 5.16145e+10
  Static Block Sequential Results:
  Time of Static Block Sequential: 28527.6
  FLOPs of Static Block Sequential: 7.01076e+07
  Residual of Static Block Sequential: 7.68161e-10
  Is results match within tolerance? YES

```

Fig 8, Screenshot bock size = 256

```

Microsoft Visual Studio Debug Console
CASE DIMENSION: 1000:
  BLAS Results:
  Time of BLAS: 29.5278
  FLOPs of BLAS: 6.77328e+10
  Static Block Sequential Results:
  Time of Static Block Sequential: 24604.7
  FLOPs of Static Block Sequential: 8.12854e+07
  Residual of Static Block Sequential: 7.68161e-10
  Is results match within tolerance? YES

```

Fig 9, Screenshot bock size = 512

### 2.1.3.2. Algorithm Explanation

```

29 // Function to do block-wise Matrix-Matrix multiplication
30 void multySequentialBlocked(const vector<double>& A, const vector<double>& B, vector<double>& C, int dimension, int blockSize) {
31     for (int ii = 0; ii < dimension; ii += blockSize) {
32         for (int jj = 0; jj < dimension; jj += blockSize) {
33             for (int kk = 0; kk < dimension; kk += blockSize) {
34                 for (int i = ii; i < min(ii + blockSize, dimension); ++i) {
35                     for (int j = jj; j < min(jj + blockSize, dimension); ++j) {
36                         for (int k = kk; k < min(kk + blockSize, dimension); ++k) {
37                             C[i * dimension + j] += A[i * dimension + k] * B[k * dimension + j];
38                         }
39                     }
40                 }
41             }
42         }
43     }
44 }

```

Fig 10, Screenshot of multySequentialBlocked() function

#### 1) Outer Loops (ii, jj, kk):

The outer loops (ii, jj, and kk) iterate over the blocks of the matrices A, B, and C.

ii and jj iterate over the blocks in the rows and columns of the resulting matrix C, and kk iterates over the blocks in the shared dimension of A and B.

#### 2) Middle Loops (i, j, k):

The middle loops (i, j, and k) iterate within each block.

i and j iterate over the rows and columns of the resulting block in matrix C.

k iterates over the shared dimension of the matrices A and B within the block.

### 3) Matrix Multiplication Operation:

Within the innermost loop, the actual matrix multiplication operation takes place.

For each element (i, j) within the resulting block of matrix C, the function calculates the dot product of the corresponding row of A (row i) and column of B (column j) within the block.

### 4) Accumulation in Matrix C:

The result of the dot product is accumulated in the corresponding element of the resulting block in matrix C.

The element at position (i, j) in matrix C is updated as  $C[i * \text{dimension} + j] += A[i * \text{dimension} + k] * B[k * \text{dimension} + j]$ .

### 5) Block Size Control:

The loop bounds are adjusted to ensure that the iteration does not go beyond the matrix dimensions.

The  $\min(ii + \text{blockSize}, \text{dimension})$ ,  $\min(jj + \text{blockSize}, \text{dimension})$ , and  $\min(kk + \text{blockSize}, \text{dimension})$  conditions prevent the loops from accessing elements outside the current block or the matrix.

#### 2.1.3.3. Algorithm FLOPs Calculation

```
79 int rows = 0;
80 const double tolerance = 1e-6;
81 bool withinTolerance = false;
82 int blockSize = STATIC_BLOCK_SIZE;
83 for (int dimension = 10; dimension <= 1000; dimension += 10) {
84     vector<double> A = generateA(dimension * dimension);
85     vector<double> B = generateB(dimension * dimension);
86     vector<double> C_Blas(dimension * dimension, 0.0);
87     vector<double> C_My(dimension * dimension, 0.0);
88     rows = dimension;
89     outputFile << dimension << " ";
90
91     auto startTime = chrono::high_resolution_clock::now();
92     cbblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, dimension, dimension, dimension, 1.0, A.data(), dimension,
93     auto endTime = chrono::high_resolution_clock::now();
94     chrono::duration<double, milli> elapsedMilliseconds = endTime - startTime;
95     double flops = (2.0 * dimension * dimension * dimension) / (elapsedMilliseconds.count() / 1000.0);
96     outputFile << flops << " ";
97
98     startTime = chrono::high_resolution_clock::now();
99     multySequentialBlocked(A, B, C_My, dimension, blockSize);
100    endTime = chrono::high_resolution_clock::now();
101    chrono::duration<double, milli> elapsedMilliseconds1 = endTime - startTime;
102    double flops1 = (2.0 * dimension * dimension * dimension) / (elapsedMilliseconds1.count() / 1000.0);
103    outputFile << flops1 << " ";
104    double residual = calculateResidual(C_Blas, C_My);
105    outputFile << residual << " ";
106    if (residual < tolerance) {
107        withinTolerance = true;
108    }
109    else {
110        withinTolerance = false;
111    }
112    outputFile << withinTolerance << "\n";
}
```

Fig 11, Screenshot of Static Block Sequential Solution's FLOPs calculation

### 1) Matrix Initialization:

Matrices A, B, and C are initialized with random values (using generateA and generateB functions).

## 2) BLAS Matrix Multiplication:

BLAS matrix multiplication (cblas\_dgemm) is performed, and its execution time and FLOPs (Floating Point Operations Per Second) are measured.

## 3) Custom Matrix Multiplication:

Custom block-wise matrix multiplication (multySequentialBlocked) is performed, and its execution time and FLOPs are measured.

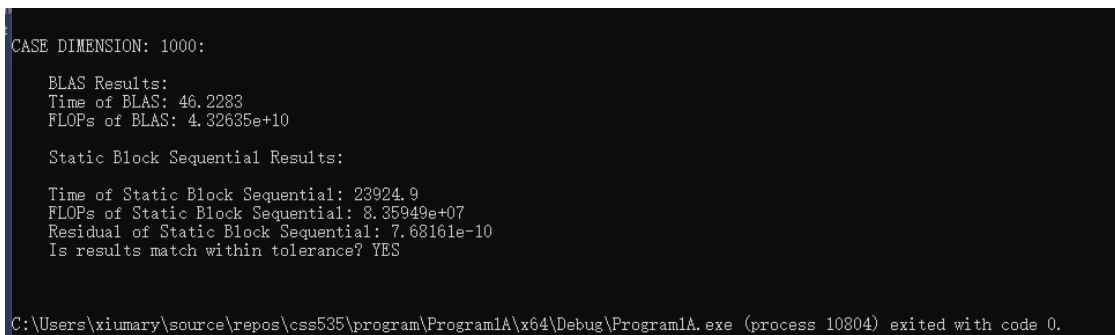
## 4) Residual Calculation:

The residual between the results of BLAS and the custom multiplication is calculated.

## 5) Output to File:

Writes the dimension, FLOPS for BLAS, FLOPS for static block sequential multiplication, residual, and a boolean indicating whether the residual is within a tolerance ( $1e-6$ ) to an output file.

### 2.1.3.4. Some Screenshots of Program Compiling



```
CASE DIMENSION: 1000:

  BLAS Results:
    Time of BLAS: 46.2283
    FLOPs of BLAS: 4.32635e+10

  Static Block Sequential Results:
    Time of Static Block Sequential: 23924.9
    FLOPs of Static Block Sequential: 8.35949e+07
    Residual of Static Block Sequential: 7.68161e-10
    Is results match within tolerance? YES

C:\Users\xiumary\source\repos\css535\program\Program1A\x64\Debug\Program1A.exe (process 10804) exited with code 0.
```

Fig 12, Screenshot of case with dimension 1000

## 2.1.4. Transpose and Unrolled Optimization Implementation and Test

### 2.1.4.1. Algorithm Explanation

#### 1) transpose function:

This function has been introduced in 1.2.4.1.

#### 2) multySequentialUnrolled() function



```

36 // Function to do Matrix-Matrix multiplication with loop unrolling
37 void multySequentialUnrolled(const vector<double>& A, const vector<double>& B, vector<double>& C, int dimension) {
38     for (int i = 0; i < dimension; ++i) {
39         for (int j = 0; j < dimension; ++j) {
40             int idx = i * dimension + j;
41             double sum = 0.0;
42
43             // Loop unrolling with step size 4
44             for (int k = 0; k < dimension; k += 4) {
45                 if (k + 3 < dimension) {
46                     sum += A[i * dimension + k] * B[k * dimension + j];
47                     sum += A[i * dimension + k + 1] * B[(k + 1) * dimension + j];
48                     sum += A[i * dimension + k + 2] * B[(k + 2) * dimension + j];
49                     sum += A[i * dimension + k + 3] * B[(k + 3) * dimension + j];
50                 }
51                 else {
52                     // Handle the case when k + 3 exceeds dimension
53                     int remainingElements = dimension - k;
54                     for (int l = 0; l < remainingElements; ++l) {
55                         sum += A[i * dimension + k + l] * B[(k + l) * dimension + j];
56                     }
57                 }
58             }
59             C[idx] += sum;
60         }
61     }
62 }
63

```

Fig 13, Screenshot of multySequentialUnrolled() function

- 1) The function uses two input matrices, A and B, and an output matrix C.
- 2) It performs a matrix multiplication  $C = A * B$ , where the matrices are assumed to be square of dimension dimension.
- 3) The outer two loops iterate over the rows and columns of the resulting matrix C.
- 4) The inner loop performs the actual multiplication, and it is unrolled with a step size of 4 for optimization.
- 5) The loop unrolling is designed to handle the case where the dimension is not a multiple of 4 in the innermost loop by using a separate loop for the remaining elements.

#### 2.1.4.2. Algorithm FLOPs Calculation

```

105 int rows = 0;
106 for (int dimension = 10; dimension <= 1000; dimension += 10) {
107     vector<double> A = generateA(dimension * dimension);
108     vector<double> B = generateB(dimension * dimension);
109     vector<double> C_Blas(dimension * dimension, 0.0);
110     vector<double> C_My(dimension * dimension, 0.0);
111
112     rows = dimension;
113     outputFile << dimension << " ";
114
115     auto startTime = chrono::high_resolution_clock::now();
116     cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, dimension, dimension, dimension, 1.0, A.data(),
117               dimension, B.data(), dimension, 0.0, C_Blas.data(), dimension);
118     auto endTime = chrono::high_resolution_clock::now();
119     chrono::duration<double, milli> elapsedMilliseconds = endTime - startTime;
120     double flops = (2.0 * dimension * dimension * dimension) / (elapsedMilliseconds.count() / 1000.0);
121     outputFile << flops << " ";
122
123     transpose(B, rows);
124
125     startTime = chrono::high_resolution_clock::now();
126     multySequentialUnrolled(A, B, C_My, dimension);
127     endTime = chrono::high_resolution_clock::now();
128     chrono::duration<double, milli> elapsedMilliseconds1 = endTime - startTime;
129     double flops1 = (2.0 * dimension * dimension * dimension) / (elapsedMilliseconds1.count() / 1000.0);
130     outputFile << flops1 << " ";
131
132     double residual = calculateResidual(C_Blas, C_My);
133     outputFile << residual << " ";
134
135     bool withinTolerance = (residual < 1e-6);
136     outputFile << withinTolerance << "\n";

```

Fig 14, Screenshot of transpose and unrolled sequential FLOPs calculation

- 1) The code iterates over matrix dimensions from 10 to 1000 with a step size of 10.
- 2) Matrices A and B are generated.
- 3) BLAS matrix multiplication is performed, and its execution time and FLOPs are measured.
- 4) Matrix B is transposed for my custom matrix multiplication.
- 5) Custom matrix multiplication (unrolled) is performed, and its execution time and FLOPs are measured.
- 6) Residual between BLAS and custom results is calculated, and whether it's within tolerance is determined.
- 7) Dimension, FLOPs for BLAS, FLOPs for the custom implementation, residual, and tolerance status are written to the output file.

#### 2.1.4.3. Some Screenshots of Program Compiling



```

CASE DIMENSION: 1000:

  BLAS Results:
  Time of BLAS: 52.7686
  FLOPs of BLAS: 3.79013e+10

  Sequential Results:

  Time of Sequential: 16980.7
  FLOPs of Sequential: 1.1778e+08
  Residual of Sequential: 7.6828e-13
  Is results match within tolerance? YES

C:\Users\xiumary\source\repos\css535\program\Program1A\x64\Debug\Program1A.exe (process 20780) exited with code 0.



```

Fig 15, Screenshot of case result with dimension 1000

#### 2.1.5. Comparing and Analysis of Three algorithms

##### 2.1.5.1. Raw Data Files

The raw data coming from “data/ sequen\_algorithm\_selection” folder, includes:

-  mmm\_s\_standard\_g++\_4C\_1000.txt
-  mmm\_s\_staticBlock\_g++\_4C\_1000.txt
-  mmm\_s\_tran\_unroll\_g++\_4C\_1000.txt

The “mmm\_s\_standard\_g++\_4C\_1000.txt” provides data for Standard Algorithm, the “mmm\_s\_staticBlock\_g++\_4C\_1000.txt” provides data for Block Algorithm, the “mmm\_s\_tran\_unroll\_g++\_4C\_1000.txt” provides data for Trans and Unrolled Algorithm.

All the test cases are the same, which have been mentioned in the “2.1.1. Environment clarification”.

##### 2.1.5.2. Plot of Tree Algorithms

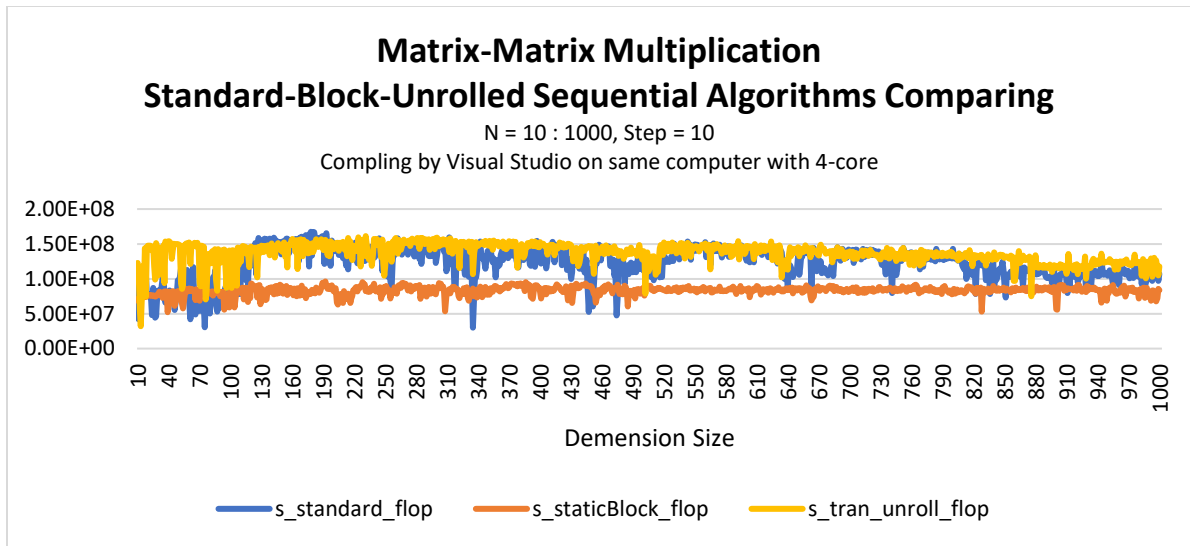


Fig 16, Standard-Block-Unrolled Sequential Matrix-Matrix Multiplication Algorithms Comparing Plot

### 2.1.5.3. Analysis of Plot

From Figure 15, it is easy to observed that block-wise optimization method is not a good way comparing with transpose & unrolled optimization method. The reason is that the additional loop structure introduced by block-wise optimization can lead to increased loop overhead. For smaller matrices, the cost of managing block-wise iterations may outweigh the potential benefits. Besides, block-wise should be good to use in the thread solution because its blocking thinking.

Another thing we can find from Figure 15 is that the transpose & unrolled optimization method is performing better than standard method as the dimension increase. The reasons could be: 1) Memory Access Patterns: Transposing one of the matrices changes the memory access patterns during multiplication. With transposed matrices, the access patterns become more cache-friendly, which can result in fewer cache misses. Efficient use of the CPU cache can significantly improve overall performance. 2) Loop Unrolling: Unrolling loops in the computation process reduces the overhead of loop control and allows the compiler to generate more efficient code. This is particularly beneficial when dealing with large matrices where the loop overhead becomes more significant.

### 2.1.6. Sequential Algorithm Selection Conclusion

In my latter experiment, I will use transpose & unrolled optimization in my sequential solution experiment, which could improve the performance significantly when dimensions size becomes large.

## 2.2. Thread solution Algorithm Selection for Programs

### 2.2.1. Algorithm Implementation Explanation Based on Sequential Algorithm

### 2.2.1.1. `multySequentialUnrolledSingleThread()` Function

```
85 // Function to do Matrix-Matrix multiplication with loop unrolling (single thread)
86 void multySequentialUnrolledSingleThread(const vector<double>& A, const vector<double>& B, vector<double>& C,
87 int dimension, int startRow, int endRow) {
88     for (int i = startRow; i < endRow; ++i) {
89         for (int j = 0; j < dimension; ++j) {
90             int idx = i * dimension + j;
91             double sum = 0.0;
92
93             // Loop unrolling with step size 4
94             for (int k = 0; k < dimension; k += 4) {
95                 if (k + 3 < dimension) {
96                     sum += A[i * dimension + k] * B[k * dimension + j];
97                     sum += A[i * dimension + k + 1] * B[(k + 1) * dimension + j];
98                     sum += A[i * dimension + k + 2] * B[(k + 2) * dimension + j];
99                     sum += A[i * dimension + k + 3] * B[(k + 3) * dimension + j];
100                 }
101                 else {
102                     // Handle the case when k + 3 exceeds dimension
103                     int remainingElements = dimension - k;
104                     for (int l = 0; l < remainingElements; ++l) {
105                         sum += A[i * dimension + k + l] * B[(k + l) * dimension + j];
106                     }
107                 }
108             }
109             C[idx] += sum;
110         }
111     }
112 }
113 }
```

Fig 17, `multySequentialUnrolledSingleThread()` function

#### 1) Loop Structure:

The outer loop (i) iterates over rows assigned to the thread. The middle loop (j) iterates over columns of the resulting matrix C. Within the innermost loop (k), the dot product is computed using loop unrolling with a step size of 4.

#### 2) Accumulation in Matrix C:

The dot product results are accumulated in the corresponding elements of the resulting block in matrix C.

### 2.2.1.2. `multySequentialUnrolledMultiThread()` Function

```
72 // Function to do Matrix-Matrix multiplication with loop unrolling (multi-threaded)
73 void multySequentialUnrolledMultiThread(const vector<double>& A, const vector<double>& B, vector<double>& C,
74 int dimension, int numThreads) {
75     vector<thread> threads;
76
77     int rowsPerThread = dimension / numThreads;
78
79     for (int i = 0; i < numThreads; ++i) {
80         int startRow = i * rowsPerThread;
81         int endRow = (i == numThreads - 1) ? dimension : (i + 1) * rowsPerThread;
82
83         threads.emplace_back(multySequentialUnrolledSingleThread, std::ref(A), std::ref(B), std::ref(C),
84 dimension, startRow, endRow);
85     }
86
87     // Join the threads to wait for them to finish
88     for (auto& t : threads) {
89         t.join();
90     }
91 }
```

Fig 18, `multySequentialUnrolledMultiThread()` function

#### 1) Thread Management:

The function creates multiple threads, each responsible for computing a subset of rows in the resulting matrix C. The number of threads is determined by the `numThreads` parameter.

#### 2) Loop Structure (Parallelized):

The function divides the matrix multiplication task among threads by distributing rows. Each thread executes the same unrolled matrix multiplication logic independently, ensuring parallel computation.

### 3) Thread Joining:

After the threads complete their tasks, the function waits for all threads to finish using the join operation.

### 4) Thread-Safe Accumulation:

Similar to the single-threaded version, a lock mechanism is used to ensure thread-safe accumulation in the shared resource C.

## 2.2.2. Algorithm FLOPs Calculation

```
136 int rows = 0;
137 for (int dimension = 10; dimension <= 1000; dimension += 1) {
138     vector<double> A = generateA(dimension * dimension);
139     vector<double> B = generateB(dimension * dimension);
140     vector<double> C_Blas(dimension * dimension, 0.0);
141     vector<double> C_My(dimension * dimension, 0.0);
142     rows = dimension;
143     outputFile << dimension << " ";
144
145     auto startTime = chrono::high_resolution_clock::now();
146     cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, dimension, dimension, dimension, 1.0, A.data(),
147               dimension, B.data(), dimension, 0.0, C_Blas.data(), dimension);
148     auto endTime = chrono::high_resolution_clock::now();
149     chrono::duration<double, milli> elapsedMilliseconds = endTime - startTime;
150     double flops = (2.0 * dimension * dimension * dimension) / (elapsedMilliseconds.count() / 1000.0);
151     outputFile << flops << " ";
152
153     transpose(B, rows);
154     auto startTImel = chrono::high_resolution_clock::now();
155     multySequentialUnrolledMultiThread(A, B, C_My, dimension, 8); // Use 8 threads
156     auto endTImel = chrono::high_resolution_clock::now();
157
158     chrono::duration<double, milli> elapsedMilliseconds1 = endTImel - startTImel;
159     double flops1 = (2.0 * dimension * dimension * dimension) / (elapsedMilliseconds1.count() / 1000.0);
160     outputFile << flops1 << " ";
161
162     double residual = calculateResidual(C_Blas, C_My);
163     outputFile << residual << " ";
164     bool withinTolerance = (residual < 1e-6);
165     outputFile << withinTolerance << "\n";
166 }
```

Fig 19, Thread algorithm FLOPs calculation of main()

### 1) Matrix Initialization:

Matrices A, B, and C are initialized with random values using the generateA and generateB functions. These matrices serve as input and output containers for the matrix multiplication operations.

### 2) BLAS Matrix Multiplication:

BLAS matrix multiplication (cblas\_dgemm) is performed to establish a baseline for comparison. The execution time and FLOPs (Floating Point Operations Per Second) for this operation are measured to quantify its performance.

### 3) Custom Matrix Multiplication (Multi-Threaded Unrolled):

Custom matrix multiplication, specifically the unrolled and multi-threaded version (multySequentialUnrolledMultiThread), is executed on matrices A and B to generate matrix C. This custom implementation aims to optimize performance through loop unrolling and parallelization.

#### 4) Performance Measurement:

Execution time and FLOPs for the custom multiplication are measured to evaluate its efficiency.

The residual between the results of BLAS multiplication and the custom implementation is calculated. This residual quantifies the numerical difference between the two approaches.

#### 5) Output to File:

The key metrics, including matrix dimension, FLOPs for BLAS, FLOPs for the custom implementation, residual, and a boolean indicating whether the residual is within a tolerance ( $1e-6$ ), are written to an output file. This information provides a comprehensive overview of the comparative performance and accuracy of the two matrix multiplication methods.

### 2.2.3. Some Screenshots of Program Compiling

```
C:\Users\xiumary\source\repos\css535\program\Program 1A\x64\Debug\Program 1A.exe
CASE DIMENSION: 10:
Matrix A:
1 2 3 1 2 3 1 2 3 1
2 3 1 2 3 1 2 3 1 2
3 1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1
2 3 1 2 3 1 2 3 1 2
3 1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1
2 3 1 2 3 1 2 3 1 2
3 1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1
Matrix B:
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
```

Figure 20, Screenshot 1 of thread algorithm when dimension = 10

```

C:\Users\xiumary\source\repos\css535\program\Program1A\Debug\Program1A.exe
BLAS Results:
0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019
0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021
0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019
0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021
0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019
0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021
0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019
0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021
0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019
Time of BLAS: 0.0751
FLOPs of BLAS: 2.66312e+07

Thread Results:
0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019
0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021
0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019
0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021
0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019
0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021
0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019
Time of Thread: 8.9332
FLOPs of Thread: 223884
Residual of Thread: 1.37142e-18
Is results match within tolerance? YES

```

Fig 21, Screenshot-2 of thread algorithm when dimension = 10

### 3. Experiment Results Correctness Explanation

#### 3.1. Theoretical Method for Getting Correct Results

In the implementation programs (matrixMatrixMutiply\_S2.cpp and matrixMatrixMutiply\_P.cpp), they calculate each test case's residual between the result of my solution and the result of BLAS. If all the residuals of test cases are less than tolerance ( $1 \times 10^{-6}$  (or  $1e-6$ ), commonly used default in numerical computing), the results of my solutions are correct, or are not correct.

Formula[4]:

$$\text{Residual} = \sqrt{\sum_{i=1}^n (Y_{\text{BLAS}}[i] - Y_{\text{My}}[i])^2}$$

- 1)  $Y_{\text{BLAS}}[i]$  represents the  $i$ -th element of the result obtained using BLAS.
- 2)  $Y_{\text{My}}[i]$  represents the  $i$ -th element of the result obtained using my solutions.

#### 3.2. calculateResidual() Function

```

11 // Function to calculate the residual between two vectors
12 double calculateResidual(const vector<double>& Y_BLAS, const vector<double>& Y_My) {
13     // Assuming Y_BLAS and Y_My have the same size
14     int n = Y_BLAS.size();
15
16     // Calculate the residual
17     double residual = 0.0;
18     for (int i = 0; i < n; ++i) {
19         double diff = Y_BLAS[i] - Y_My[i];
20         residual += diff * diff;
21     }
22
23     return sqrt(residual);
24 }

```

**Fig 22, calculateResidual() function**

### **1) Input Vectors:**

Y\_BLAS: Represents a vector of results obtained using a BLAS (Basic Linear Algebra Subprograms) operation or library.

Y\_My: Represents a vector of results obtained using custom or user-defined calculations.

### **2) Vector Size Check:**

The function assumes that both input vectors (Y\_BLAS and Y\_My) have the same size (n). This assumption is crucial for a meaningful comparison between the two vectors.

### **3) Residual Calculation:**

The function initializes a variable residual to zero.

It iterates through each element of the vectors using the loop variable i.

For each element, it calculates the difference between the corresponding elements of Y\_BLAS and Y\_My (i.e.,  $\text{diff} = Y\_BLAS[i] - Y\_My[i]$ ).

The squared difference ( $\text{diff} * \text{diff}$ ) is added to the residual.

This process is repeated for all elements in the vectors.

### **4) Euclidean Norm Calculation:**

After accumulating the squared differences, the function takes the square root of the total ( $\text{sqrt}(\text{residual})$ ).

The square root operation is the final step in calculating the Euclidean norm.

### **5) Return Value:**

The function returns the calculated Euclidean norm, representing the overall difference between the vectors.

## **3.3. Specific Application in Sequential Solution**

### **3.3.1. In main() Part**



“double residual = calculateResidual(C\_Blas, C\_My);” will get the residual.

```

109 vector<double> A = generateA(dimension * dimension);
110 vector<double> B = generateB(dimension * dimension);
111 vector<double> C_Blas(dimension * dimension, 0.0);
112 vector<double> C_My(dimension * dimension, 0.0);
113
114 rows = dimension;
115
116 outputFile << dimension << " ";
117
118 auto startTime = chrono::high_resolution_clock::now();
119 cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, dimension, dimension, dimension, 1.0, A.data(), dimension
120 auto endTime = chrono::high_resolution_clock::now();
121 chrono::duration<double, milli> elapsedMilliseconds = endTime - startTime;
122 double flops = (2.0 * dimension * dimension * dimension) / (elapsedMilliseconds.count() / 1000.0);
123 outputFile << flops << " ";
124
125 transpose(B, rows);
126
127 startTime = chrono::high_resolution_clock::now();
128 multySequentialUnrolled(A, B, C_My, dimension);
129 endTime = chrono::high_resolution_clock::now();
130 chrono::duration<double, milli> elapsedMilliseconds1 = endTime - startTime;
131 double flops1 = (2.0 * dimension * dimension * dimension) / (elapsedMilliseconds1.count() / 1000.0);
132 outputFile << flops1 << " ";
133
134 double residual = calculateResidual(C_Blas, C_My);
135 outputFile << residual << " ";
136
137 bool withinTolerance = (residual < 1e-6);
138 outputFile << withinTolerance << "\n";

```

Fig 23, calculateResidual() using in main part of sequential solution

### 3.3.2. Screen Shots of Partial Result Screen Output

Because it is hard to output all the results to the screen, the program just output partial results. All the useful data is written into file.

In the Fig 24, the residual is 1.37142e-18, which is less than the tolerance mentioned in the “3.1.”

“YES” means the result of residual is match within tolerance, so dimension-10 matrix \* matrix’s result is correct.

```

Command Prompt - sequentialM2.exe
FLOPs of BLAS: 4.34783e+07

Sequential Results:
0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019
0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021
0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019
0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021
0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019
0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021
0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019

Time of Sequential: 0.0171
FLOPs of Sequential: 1.16959e+08
Residual of Sequential: 1.37142e-18
Is results match within tolerance? YES

```

Fig 24, screenshots of Sequential Solution partial results with dimension 10

### 3.3.3. All Residuals

In each line of the data files generated by “matrixMatrixMutiply\_S2.cpp”, the last number showing the value of residual tolerance status is always 1, which mean that the results of matrix-matrix vector multiplication calculated by Sequential Solution are correct. Please check the raw data files

“mm\_s\_trans\_unroll\_g++\_4C\_1000.txt”, “mm\_s\_trans\_unroll\_g++\_4C\_5000.txt”, “mm\_s\_trans\_unroll\_g++\_4C\_8000.txt”, “mm\_s\_trans\_unroll\_g++\_4C\_10000.txt”, in the “data\4core\_myC” folder.

## 3.4. Specific Application in Thread Solution

### 3.4.1. In main() Part

“double residual = calculateResidual(C\_Blas, C\_My);” will get the residual.

```
118     auto startTime = chrono::high_resolution_clock::now();
119     cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, dimension, dimension, dimension, 1.0, A.data(),
120               dimension, B.data(), dimension, 0.0, C_Blas.data(), dimension);
121     auto endTime = chrono::high_resolution_clock::now();
122     chrono::duration<double, milli> elapsedMilliseconds = endTime - startTime;
123     double flops = (2.0 * dimension * dimension * dimension) / (elapsedMilliseconds.count() / 1000.0);
124     outputFile << flops << " ";
125
126     transpose(B, rows);
127
128     startTime = chrono::high_resolution_clock::now();
129     multySequentialUnrolled(A, B, C_My, dimension);
130     endTime = chrono::high_resolution_clock::now();
131     chrono::duration<double, milli> elapsedMilliseconds1 = endTime - startTime;
132     double flops1 = (2.0 * dimension * dimension * dimension) / (elapsedMilliseconds1.count() / 1000.0);
133     outputFile << flops1 << " ";
134
135     double residual = calculateResidual(C_Blas, C_My);
136     outputFile << residual << " ";
137
138     bool withinTolerance = (residual < 1e-6);
139     outputFile << withinTolerance << "\n";
```

Figure 25, calculateResidual using in main() of sequential solution

### 3.4.2. Screen Shots of Partial Result Screen Output

Because it is hard to output all the results to the screen, the program just output partial results. All the useful data is written into file.

In the Fig 27, the residual is 1.37142e-18, which is less than the tolerance mentioned in the “3.1.”.

“YES” means the result of residual is match within tolerance, so dimension-10 matrix \* matrix’s result is correct.

```

Command Prompt - threadM.exe
D:\xzAssignment\programl\PartB\test>g++ matrixMatrixMultiply_P.cpp -o threadM.exe -I"D:/xzAssignment/software/OpenBLAS-0.3.26-x64/include" -L"D:/xzAssignment/software/OpenBLAS-0.3.26-x64/lib" -lopenblas
D:\xzAssignment\programl\PartB\test>threadM.exe
CASE DIMENSION: 10:

Matrix A:
1 2 3 1 2 3 1 2 3 1
2 3 1 2 3 1 2 3 1 2
3 1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1
2 3 1 2 3 1 2 3 1 2
3 1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1
2 3 1 2 3 1 2 3 1 2
3 1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1

Matrix B:
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001

```

Fig 26, result screenshot-1 of Thread Solution with dimension 10

```

Command Prompt - threadM.exe
BLAS Results:
0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019
0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021
0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019
0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021
0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019
0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021
0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019

FLOPs of BLAS: 5.22193e+07

Thread Results:
0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019
0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021
0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019
0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021
0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019
0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021
0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019

Time of Thread: 1.6725
FLOPs of Thread: 1.19581e+06
Residual of Thread: 1.37142e-18
Is results match within tolerance? YES

```

Fig 27, result screenshot-2 of Thread Solution with dimension 10

### 3.4.3. All Residuals

In each line of the data files generated by “matrixMatrixMutiply\_P.cpp”, the last number showing the value of residual tolerance status is always 1, which mean that the results of matrix- matrix vector multiplication calculated by Thread Sequential Solution are correct. Please check the raw data files

“mmm\_p\_trans\_unroll\_g++\_4C\_1000.txt”, “mmm\_p\_trans\_unroll\_g++\_4C\_5000.txt”,  
“mmm\_p\_trans\_unroll\_g++\_4C\_8000.txt”, “mmm\_p\_trans\_unroll\_g++\_4C\_10000.txt”, in the “data\4core\_myC”  
folder.

## 4. Sequential-Thread-BLAS FLOPs Comparing Experiments (in my computer)

### 4.1. Environment Clarification

In order to conducting experiments under the same environment, there are some things to be declared.

#### 1) About compiler and computer

Use g++ to compile all the programs considering easy compiling.

All the experiments will be run in my computer, which information could be found in “6. Running Environment Declaring (CPU, System, IDE)”.

#### 2) About test cases

All the three solutions’ program experiments use the same cases generated by generateA() and generateB(), which could be found in figure 1. The case’s dimension size increases from 10 to 10000 using different increasements.

Here are the specific increasements:

- 10 – 1000: increasement is 1;
- 1000 – 5000: increasement is 50;
- 5000 – 8000: increasement is 200;
- 8000 – 10000: increasement is 400;

#### 3) About results correction

In every case, calculate residual between my solution and cblas\_dgemm (BLAS) to verify the results correction. Besides, provide give a boolean to indicate whether the residual is within a tolerance ( $1e-6$ ). The correction explanation has been done in “3. Experiment Results Correctness Explanation”.

#### 4) About running computer and network

All the three solutions’ program experiments are conducted in the same computer and network.

### 4.2. Sequential Solution Based on Transpose & Unrolled Algorithm

From Fig 23, it is clear that timer just records the real matrix – matrix multiplication time and then calculate the FLOPs for every case.

Here is screenshot of running Sequential Solution program.

```

Command Prompt
Time of Sequential: 6.14122e+06
FLOPs of Sequential: 1.54546e+08
Residual of Sequential: 9.57771e-10
Is results match within tolerance? YES

CASE DIMENSION: 8000:

Matrix A:

Matrix B:

BLAS Results:
Time of BLAS: 11906.6
FLOPs of BLAS: 8.60027e+10

Sequential Results:
Time of Sequential: 6.07092e+06
FLOPs of Sequential: 1.68673e+08
Residual of Sequential: 1.02141e-09
Is results match within tolerance? YES

D:\xzAssignment\program1\PartB\test>_

```

Fig 28, result screenshot of Sequential Solution Program with dimension 8000

### 4.3. Thread Solution Based on Transpose & Unrolled Sequential Solution

From Fig 25, it is clear that timer just records the real matrix – matrix multiplication time and then calculate the FLOPs for every case. Here is screenshot of running Thread Sequential Solution program.

```

CASE DIMENSION: 5000:

BLAS Results:
Time of BLAS: 2151.1
FLOPs of BLAS: 1.16219e+11

Thread Results:

Time of Thread: 481346
FLOPs of Thread: 5.19377e+08
Residual of Thread: 3.00872e-10
Is results match within tolerance? YES

D:\xzAssignment\program1\PartB\test>_

```

Fig 29, result screenshot of Thread Solution Program with dimension 5000

### 4.4. BLAS Solution Based on OpenBLAS Function

From Fig 30, it is clear that timer just records the real matrix – matrix multiplication time and then calculate the FLOPs for every case.

```

140 // Calculate C_Blas = A * B, and calculate Blas' flops
141 auto startTime = chrono::high_resolution_clock::now();
142 cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, dimension, dimension, dimension, 1.0,
143             A.data(), dimension, B.data(), dimension, 0.0, C_Blas.data(), dimension);
144 auto endTime = chrono::high_resolution_clock::now();
145 chrono::duration<double, milli> elapsedMilliseconds = endTime - startTime;
146 double flops = (2.0 * dimension * dimension * dimension) / (elapsedMilliseconds.count() / 1000.0);
147
148 // Write blas flops into outputFile
149 outputFile << flops << " ";
150

```

Fig 30, calculation part of BLAS

Here are several screenshots of running BLAS Solution program.

```
CASE DIMENSION: 5000:

BLAS Results:
Time of BLAS: 1823.31
FLOPs of BLAS: 1.37113e+11













D:\xzAssignment\program1\PartB\test>_
```

Fig 31, result screenshot of BLAS Solution Program with dimension 5000

## 5. Sequential-Thread-BLAS FLOPs Comparing Analysis (in my computer)

### 5.1. Raw Data Files

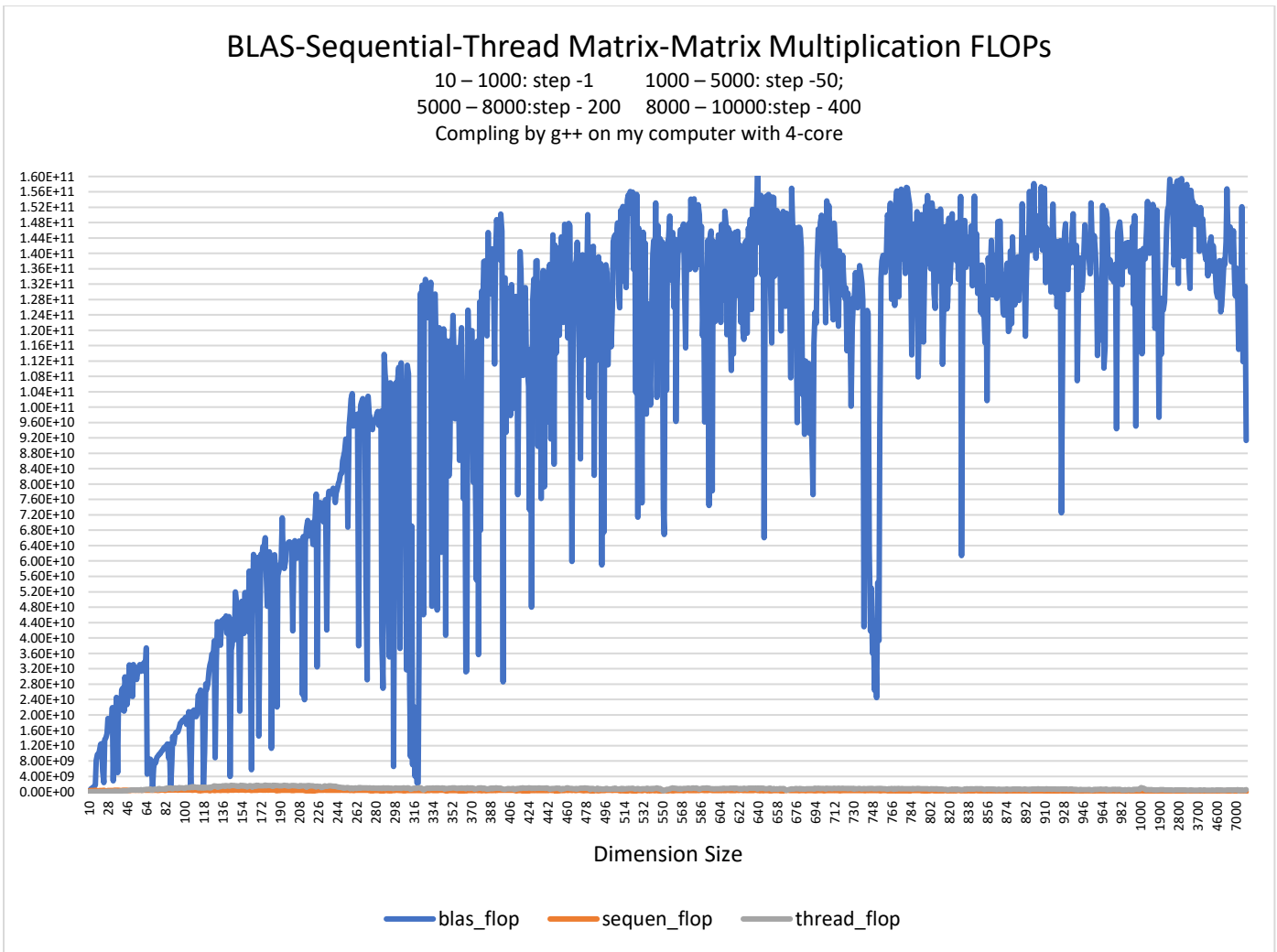
The raw data coming from “4-core\_g++\_myC” folder, includes:

 mm_b_g++_4C_1000.txt	 mm_s_trans_unroll_g++_4C_8000.txt
 mm_b_g++_4C_5000.txt	 mm_s_trans_unroll_g++_4C_10000.txt
 mm_b_g++_4C_8000.txt	 mmm_p_trans_unroll_g++_4C_1000.txt
 mm_b_g++_4C_10000.txt	 mmm_p_trans_unroll_g++_4C_5000.txt
 mm_s_trans_unroll_g++_4C_1000.txt	 mmm_p_trans_unroll_g++_4C_8000.txt
 mm_s_trans_unroll_g++_4C_5000.txt	 mmm_p_trans_unroll_g++_4C_10000.txt

The “mm\_b\_g++\_4C” series provide data for BLAS Sequential Solution, the “mm\_s\_trans\_unroll\_g++\_4C” series provide data for Sequential Solution, the “mmm\_p\_trans\_unroll\_g++\_4C” series provides data for Thread Solution.

All the test cases are the same, which have been mentioned in the “4.1.Environment clarification”.

### 5.2. BLAS-Sequential-Thread FLOPs Plot-1



**Fig 32,** BLAS-Sequential-Thread Matrix-Matrix Multiplication FLOPs Plot1 (on my computer with 4-core)

### 5.3. Plot-1 Analysis

From Fig 32, it is obvious that BLAS is much faster than Sequential and Thread. There are several reasons led this result.

#### 5.3.1. Why Sequential Is the Slowest

From the figure 13, my Sequential Solution's multiplication just uses unrolling optimization, not uses others, so it runs slowest.

##### 1) Limited Optimization in Sequential Solution:

The sequential solution primarily utilizes loop unrolling optimization to enhance its performance. However, it does not incorporate more advanced optimization techniques found in specialized libraries like BLAS. As a result, its execution speed may be comparatively slower, particularly for large matrices.

## 2) Sequential Computation and Lack of Parallelism:

In the sequential implementation, matrix multiplication is performed serially, calculating one element at a time. This lack of parallelism can lead to slower overall processing, especially when dealing with large matrices. The absence of concurrent computation may impact the solution's efficiency.

## 3) Suboptimal CPU Utilization on Multi-Core Systems:

On a multi-core CPU with hyper-threading (4 cores, 8 logical processors), the sequential version may not fully leverage the available computing resources. The limited parallelism in the algorithm restricts its ability to efficiently distribute the workload across multiple cores, resulting in underutilization of the hardware.

### 5.3.2. Why Threaded Is Better than Sequential

From Figure 17 and 18, Thread Solution is based on Sequential Solution, it adopts loop unrolling optimization, Parallel Execution(The threaded implementation divides the work among multiple threads, enabling parallel execution; Each thread works on a subset of the problem, leading to concurrent computation and potentially faster results) and Utilizing Multi-Core CPUs(threading allows to utilize multiple cores simultaneously, which is especially beneficial for computationally intensive tasks like matrix-matrix multiplication, where parallelism can significantly improve overall performance). These adding optimizations let Thread Solution performances better than Sequential Solution. However, comparing with BLAS, its performance is much slower, because it doesn't fully use all the possible optimizations like "5.3.3." analysis.

### 5.3.3. Why BLAS Is So Much Faster

When using OpenBLAS, the reasons for its high performance can be explained in the context of its implementation and optimization strategies:

- Highly Optimized Implementations:

Assembly Language and Low-Level Optimization: OpenBLAS is implemented in low-level languages, including Assembly, and employs various optimization techniques such as loop unrolling and architecture-specific optimizations. This enables the library to maximize the use of underlying hardware features for efficient execution.

- Parallelization:



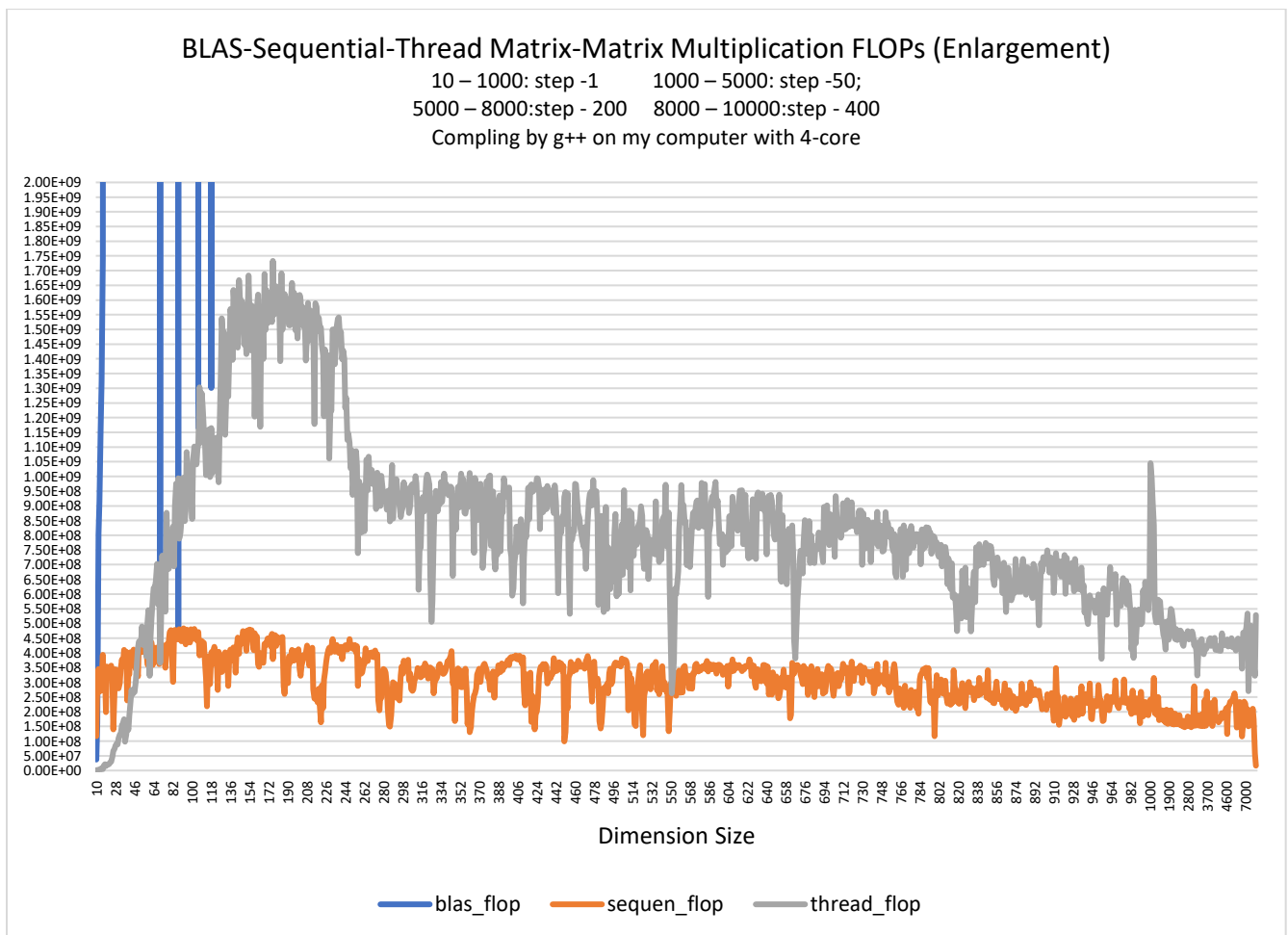
Multithreading: OpenBLAS incorporates multithreading techniques to parallelize operations across multiple CPU cores. This allows it to perform concurrent computations on subsets of data, leveraging the parallel processing capabilities of modern CPUs.

SIMD Vectorization: OpenBLAS utilizes SIMD (Single Instruction, Multiple Data) instructions for vectorized processing. This means that a single instruction can operate on multiple data elements simultaneously, enhancing parallelism and accelerating computation.

- Vendor-Specific Implementations:

Optimizations for Specific Architectures: OpenBLAS includes optimizations that are tailored for specific CPU architectures. These optimizations take advantage of the unique features and instruction sets provided by different processors, resulting in vendor-specific optimizations that enhance overall performance.

#### 5.4. BLAS-Sequential-Thread FLOPs Plot-2



**Fig 33,** BLAS-Sequential-Thread Matrix-Matrix Multiplication FLOPs Plot2 (on my computer with 4-core)

## 5.5. Plot-2 Combing Plot-1 Analysis

All the analysis of this part will combine Fig 32 and 33.

### 5.5.1. BLAS Sharp Climbing – Fast Climbing - Stable

The performance of BLAS is very slow at the beginning small matrix size comparing the large matrix size. For the first call, it almost increases vertically. Then it increases to size about 70, it has a sharp drop, then continuously climbs with fast speed until some value, then keep stable.

**The analysis is:**

Small Matrix Sizes: OpenBLAS, like other BLAS libraries, has initialization overhead for smaller matrices. The initial vertical increase could be associated with this setup cost.

Sharp Drop at Size about 70: Specific optimizations or algorithmic changes within OpenBLAS for matrices around size 70 might result in a sudden drop. The library might switch between different algorithms optimized for various matrix dimensions.

Stable for the large size: The stabilization of performance in BLAS (OpenBLAS) for larger matrix sizes could be attributed to the combination of various optimizations reaching a stable and efficient configuration.

### 5.5.2. Sequential Solution Slightly Dropping as Size Increasing

The sequential solution keeps relatively stable performance, and has a little dropping with the matrix size increasing.

**The analysis is:**

Stable Performance: The sequential solution's stable performance aligns with expectations. Sequential implementations generally provide consistent performance, but their lack of parallelism limits their speedup for larger matrices.

### 5.5.3. Thread Solution Sharp Climbing - Sharp Dropping - Slight Dropping

The Thread Solution has a sharp climbing at the beginning, then has an obvious dropping, then keep a little bit dropping as the matrix size increasing.

**The analysis is:**

Sharp Climbing: Initial performance climb in the threaded solution could be attributed to parallel computation through multiple threads, providing a speedup.

Dropping After Sharp Climbing: The subsequent drop after sharp climbing might occur due to increased overhead in managing threads for the given matrix size.

Additional Slow Dropping: It could result from heightened contention for resources as the matrix size increases.

## **6. My Computer Information (CPU, System, IDE)**

### **6.1. CPU Specifications of My Computer**

- Processor: Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz
- Base speed: 2.11 GHZ
- Socket: 1
- Cores: 4
- Logical processors: 8
- L1 cache: 256K
- L2 cache: 1.0M
- L3 cache: 8.0M

### **6.2. Operating System of My Computer**

System type: 64-bit operating system, x64-based processor

### **6.3. Compiler of Programs for BLAS-Sequential-Thread FLOPs (in my Computer)**

Use G++ for windows system.

### **6.4. IDE**

Use Visual Studio 2022 to debugger these three solution programs.

## **7. Calculate Theoretical Peak Performance**

### **7.1. My CPU Model Details**

Please check figure 34 and 35, which comes from Intel website

(<https://ark.intel.com/content/www/us/en/ark/products/124968/intel-core-i7-8650u-processor-8m-cache-up-to-4-20-ghz.html>).

CPU Specifications	
Total Cores ?	4
Total Threads ?	8
Max Turbo Frequency ?	4.20 GHz
Intel® Turbo Boost Technology 2.0 Frequency† ?	4.20 GHz
Processor Base Frequency ?	1.90 GHz
Cache ?	8 MB Intel® Smart Cache
Bus Speed ?	4 GT/s
TDP ?	15 W
Configurable TDP-up Base Frequency ?	2.10 GHz
Configurable TDP-up ?	25 W
Configurable TDP-down Base Frequency ?	800 MHz
Configurable TDP-down ?	10 W

**Fig 34, My CPU Details-1**

Instruction Set ?	64-bit
Instruction Set Extensions ?	Intel® SSE4.1, Intel® SSE4.2, Intel® AVX2
Intel® My WiFi Technology ?	Yes
Idle States ?	Yes
Enhanced Intel SpeedStep® Technology ?	Yes
Thermal Monitoring Technologies ?	Yes

**Fig 35, My CPU Details-2**

## 7.2. Formula

Theoretical Peak Performance (FLOPS)= (CPU speed in GHz) × (number of CPU cores) × (CPU instruction per clock cycle) × (number of CPUs per node)

- Number of CPU cores: 4
- CPU max speed in GHz: 4.20 GHz (see Fig 34)
- Instructions per Cycle (IPC): 8(see Fig 35, AVX2).
- Number of CPUs per node: 1 (my computer has one CPU socket)

## 7.3. Theoretical Peak Performance (FLOPS) of my computer

Theoretical Peak Performance (FLOPS) =  $4.2\text{GHz} * 4 * 8 * 1$

= 134.40 GFLOPs

## 7.4. Compare Solutions FLOPs (in my computer) with Peak Performance

### 7.4.1. Theoretical Peak Performance

My computer's theoretical peak performance is stated as 67.20 GFLOPs. This is the maximum performance my computer could achieve under ideal conditions.

### 7.4.2. Actual Performance Results

Sequential Solution average GFLOPs: 0.305 GFLOPs

Thread Solution average GFLOPs: 0.810 GFLOPs

BLAS Solution average GFLOPs: 106 GFLOPs

### 7.4.3. Analysis

#### 1) Sequential Solution:

**Significant Underutilization:** The sequential solution's average performance of 0.305 GFLOPs is far below the theoretical peak of 67.20 GFLOPs. This indicates that the sequential implementation is not fully utilizing the computational resources available on the computer.

#### Potential Reasons:

- **Limited CPU Utilization:** The sequential solution doesn't effectively utilize all available CPU cores. Since it operates sequentially, it cannot take full advantage of multi-core architectures.
- **Absence of Optimizations:** The sequential version lacks low-level optimizations and parallelization strategies found in optimized libraries like BLAS. This absence hinders its ability to achieve peak performance.
- **Memory Access Patterns:** Inefficient memory access patterns could contribute to slower performance for larger matrices. Optimizing memory access is crucial for efficient computation, and the sequential version may not have these optimizations.

#### 2) Thread Solution:

**Improvement but not Optimized:** The threaded solution's performance improvement to 0.810 GFLOPs from the sequential version suggests that multithreading has had a positive impact. However, it's

essential to note that this performance is still below the theoretical peak, indicating there's room for further optimization.

**Potential Reasons:**

- **Thread Overhead:** While the threaded solution shows improvement, it still falls short of the theoretical peak. This suggests that managing multiple threads introduces overhead, especially as the matrix size increases. The benefits of multithreading might be offset by the overhead associated with thread synchronization and management.
- **Suboptimal Thread Scaling:** The performance improvement indicates that multithreading has a positive impact, but the scaling might not be linear. As the matrix size increases, the benefits of additional threads may diminish, and the overhead may become more pronounced.

**3) BLAS Solution:**

**High Performance:** The BLAS solution demonstrates the highest performance among the tested implementations, achieving 106 GFLOPs on average, close to the theoretical Peak.

**Potential Reasons:**

- **Highly Optimized Implementations:**

**Low-Level Language Implementation:** OpenBLAS is implemented in low-level languages like Assembly and C, allowing for fine-grained control over hardware-specific optimizations. This results in efficient execution of matrix operations.

**Loop Unrolling and Vectorization:** Techniques like loop unrolling and vectorization are likely employed in OpenBLAS to maximize the usage of SIMD (Single Instruction, Multiple Data) instructions, enhancing parallelism and throughput.

**Architecture-Specific Optimizations:** OpenBLAS may include optimizations tailored for various CPU architectures. These optimizations exploit the specific features of a CPU, ensuring that the library performs efficiently on a wide range of hardware.

- **Parallelization:**

**Threaded Computations:** OpenBLAS can internally parallelize computations by utilizing multiple threads. This allows it to take advantage of multi-core CPUs, distributing the workload and significantly improving performance for parallelizable tasks like matrix multiplication.

**Thread Pool Management:** OpenBLAS likely employs an efficient thread pool management system to handle the parallel execution of tasks, minimizing overhead related to thread creation and destruction.

- **Vendor-Specific Implementations:**

Optimizations for Various Architectures: OpenBLAS may incorporate vendor-specific optimizations for CPUs from different manufacturers. This ensures that the library adapts to the nuances of various architectures, extracting maximum performance on each platform.

Flexible Configurations: OpenBLAS might provide users with configurations or parameters to fine-tune the library's behavior based on the underlying hardware. This adaptability contributes to achieving optimal performance on diverse systems.

## **8. Sequential-Thread-BLAS FLOPs Comparing Experiments (in different computers)**

### **8.1. Environment Clarification (Compiler – Computer – Test Case Design)**

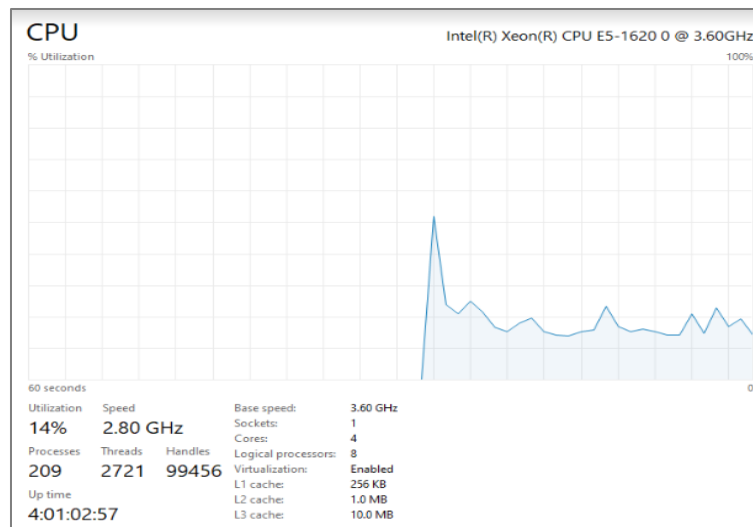
#### **1) Compiler**

Using compiler of Visual Studio 2002 IDE to compile programs, because another computer is my friend's company computer, it is not allowed to install software without permission.

#### **2) About my computer Information, please check "6."**

#### **3) About another computer information:**

- CPU Information (see Fig 36):
  - Processor: Intel(R) Xeon(R) CPU E5-1620 @ 3.60GHz
  - Base speed: 3.6 GHZ
  - Socket: 1
  - Cores: 4
  - Logical processors: 8
  - L1 cache: 256K
  - L2 cache: 1.0M
  - L3 cache: 10.0M
- Operating System: Microsoft Windows 11 Enterprise



**Fig 36, another computer CUP information**

#### 4) Test Cases Design

Because this part experiment is for comparing same solutions' different performance in different computers. So, the test ceases' dimension changes from 1 to 1000, step 1.

## 8.2. Programs of Conducting Experiments

There is no extra program, all the programs for conducting experiments here use the same programs in the one in “4. Sequential-Thread-BLAS FLOPs Comparing Experiments (in my computer)”, just the files generated by programs are different.

### 8.3. Partial Output Screenshots on My Computer

### 1) Sequential Solution Program Output on My Computer

[illegible]

**Fig 37,** Sequential Solution output screenshot-1 on my computer with dimension 10







## 8.4. Partial Output Screenshots on Another Computer

### 1) Sequential Solution program output on another computer

```
C:\Users\tengfei\source\repo: X + v
CASE DIMENSION: 10:
Marix A:
1 2 3 1 2 3 1 2 3 1
2 3 1 2 3 1 2 3 1 2
3 1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1
2 3 1 2 3 1 2 3 1 2
3 1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1
2 3 1 2 3 1 2 3 1 2
3 1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1
Marix B:
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
```

Fig 42, Sequential Solution output screenshot-1 on my computer with dimension 10

```
C:\Users\tengfei\source\repo: X + v
BLAS Results:
0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019
0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021
0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019
0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021
0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019
0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021
0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019
Time of BLAS: 0.0683
FLOPs of BLAS: 2.92826e+07

Sequential Results:
0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019
0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021
0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019
0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021
0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019
0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021 0.0021
0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019
Time of Sequential: 0.0148
FLOPs of Sequential: 1.35135e+08
Residual of Sequential: 0
Is results match within tolerance? YES
```

Fig 43, Sequential Solution output screenshot-2 on my computer with dimension 10

```
CASE DIMENSION: 1000:

  BLAS Results:
  Time of BLAS: 24.0294
  FLOPs of BLAS: 8.32314e+10

  Sequential Results:
  Time of Sequential: 20131
  FLOPs of Sequential: 9.93493e+07
  Residual of Sequential: 1.18429e-12
  Is results match within tolerance? YES

C:\Users\tengfei\source\repos\projects\x64\Debug\backup_exe (process 13924)
```

Fig 44, Sequential Solution output screenshot-2 on my computer with dimension 100

## 2) Thread Solution program output on another computer

```
C:\Users\tengfei\source\repos\ x + v

Matrix A:
1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1 2 3 1 2 3

Matrix B:
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
```

Fig 45, Thread Solution output screenshot-1 on my computer with dimension 15



## 9. Sequential-Thread-BLAS FLOPs Comparing and Analysis (in different computers)

### 9.1. Raw Data Files

My computer files list as follows:

mm\_b1\_myC\_1000.txt  
mmm\_p1\_myC\_1000.txt  
mmm\_s1\_myC\_1000.txt

Another computer files list as follows:

mm\_b1\_another\_1000.txt  
mmm\_p1\_another\_1000.txt  
mmm\_s1\_another\_1000.txt

### 9.2. BLAS-Sequential-Thread FLOPs Plot-11

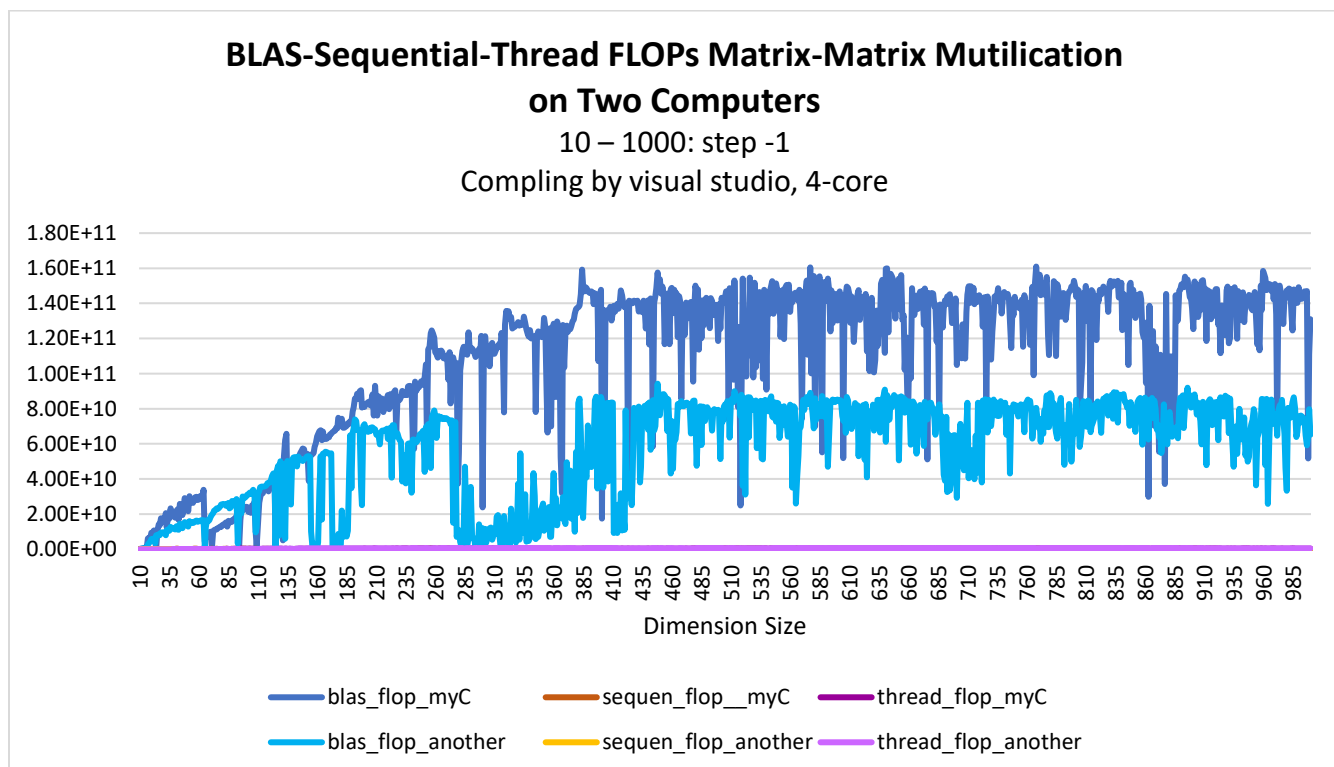


Fig 48, BLAS-Sequential-Thread Matrix-Matrix Multiplication FLOPs Plot-11 (Two computers)

### 9.3. BLAS-Sequential-Thread FLOPs Plot-22

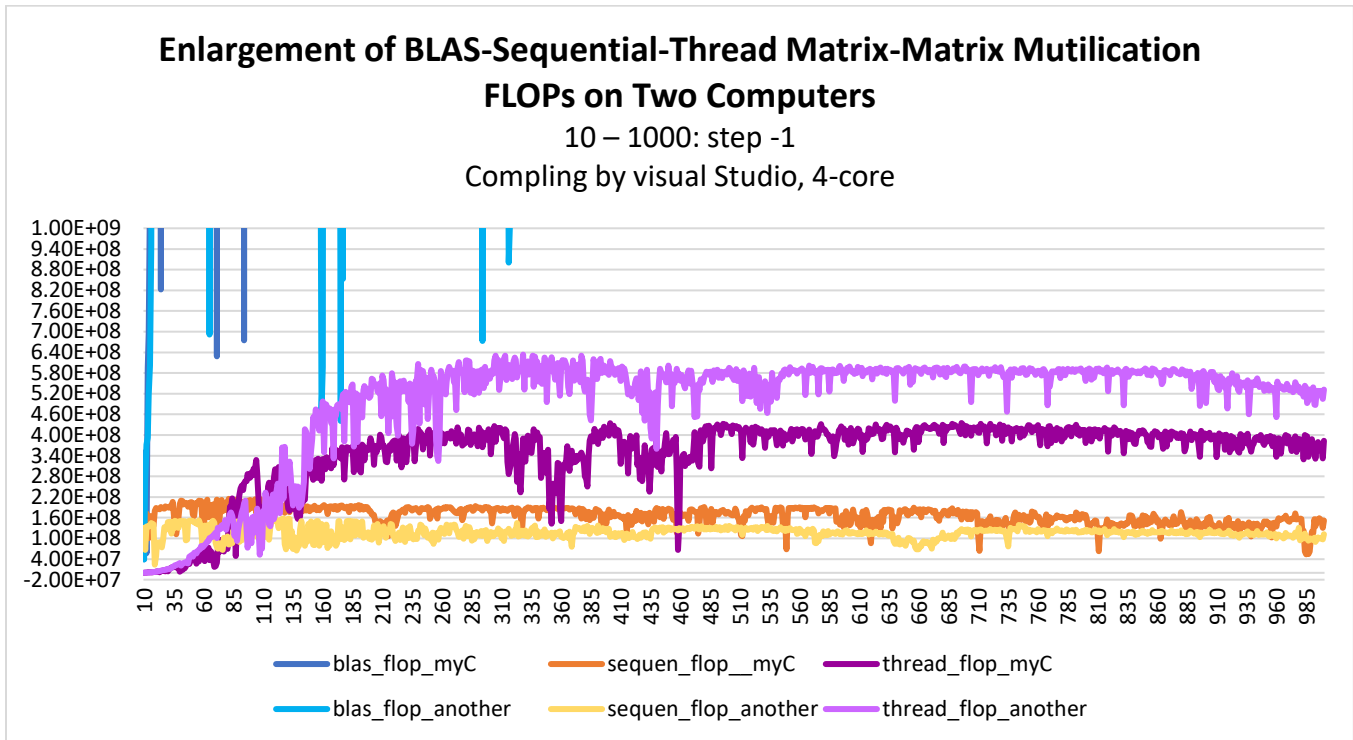


Fig 49, BLAS-Sequential-Thread Matrix-Matrix Multiplication FLOPs Plot-22 (Two computers)

### 9.4. Plot-11 and Plot-22 Analysis

#### 9.4.1. Finding form Plot-11 and Plot-22

From Fig 48 and 49, it is easy to get these findings:

- 1) For BLAS Solution's FLOPs line:  
The line of my computer is higher than the line of another computer.
- 2) For Sequential Solution's FLOPs line:  
The line of my computer is higher than the line of another computer.
- 3) For Thread Solution's FLOPs line:  
The line of my computer is lower than the line of another computer.

#### 9.4.2. Explanation for Finding

- 1) BLAS Solution of my computer has better performance than another one:  
It is possible that OpenBLAS fits my computer hardware and operating system better.
- 2) Sequential Solution of my computer has a little bit better performance than another one:

It is possible that the single-threaded efficiency of my CPU is a little bit better than that of the other computer.

- 3) Thread Solution of my computer has an obvious worse performance than another one.

It is possible that the other computer has better parallel design than my computer, because the other computer is a workstation, which often features enhanced parallel processing capabilities, contributing to their ability to handle multi-threaded workloads more effectively.

## 10. Reference

- [1] NVIDIA Corporation, "Matrix Multiplication Background User's Guide", <https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html>, 2023.
- [2] Wikipedia, "Matrix multiplication", [https://en.wikipedia.org/wiki/Matrix\\_multiplication](https://en.wikipedia.org/wiki/Matrix_multiplication), 2020.
- [3] University of California, Santa Barbara, "Optimizing Cache Performance in Matrix Multiplication", <https://sites.cs.ucsb.edu/~tyang/class/240a17/slides/Cache3.pdf>, 2017.
- [4] Richard L. Burden and J. Douglas Faires, "Numerical Analysis", page 41