**Report Template:**

**CS205 C/ C++ Program Design**

**Project 1**

**Name**: 邵冠方, **SID**: 11912009

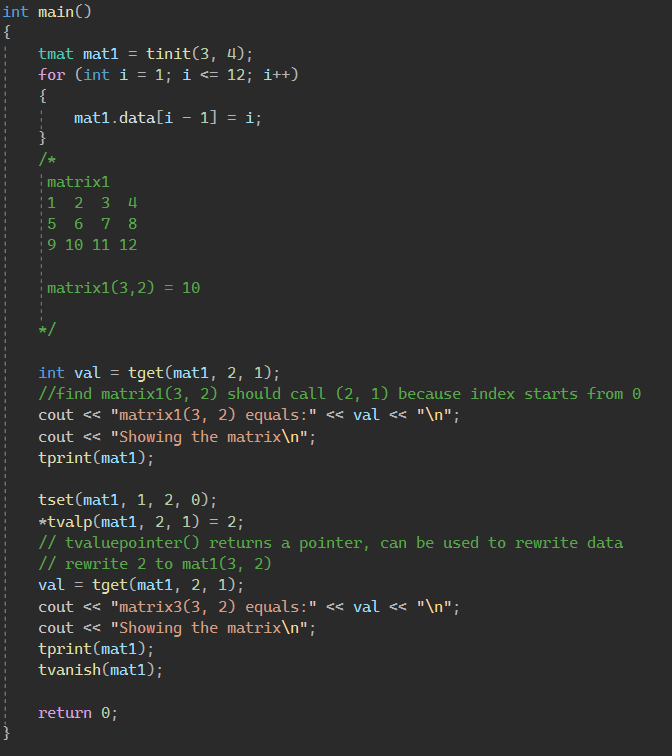
## Part 1. Source Code

https://github.com/Tokasumi/CS205midterm

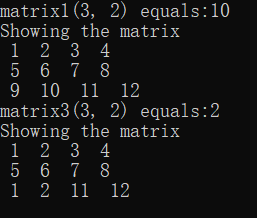
## Part 2. Result & Verification

In this part, you should present the result of your program by listing the output of test cases and optionally add a screen-shot of the result.

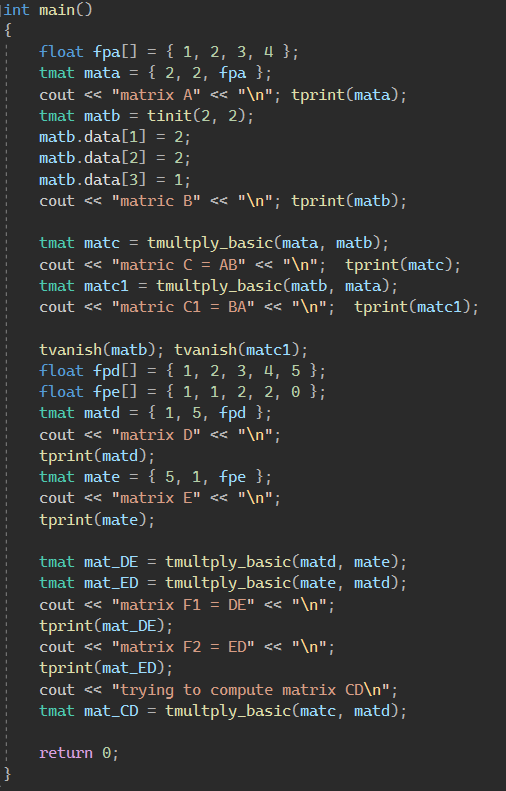
Test case #1:



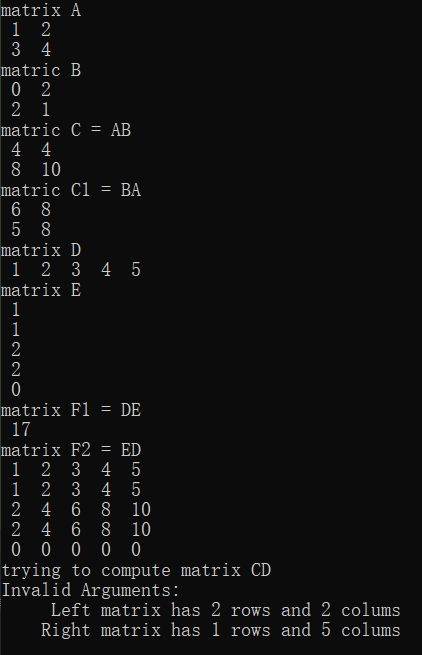
Run result:



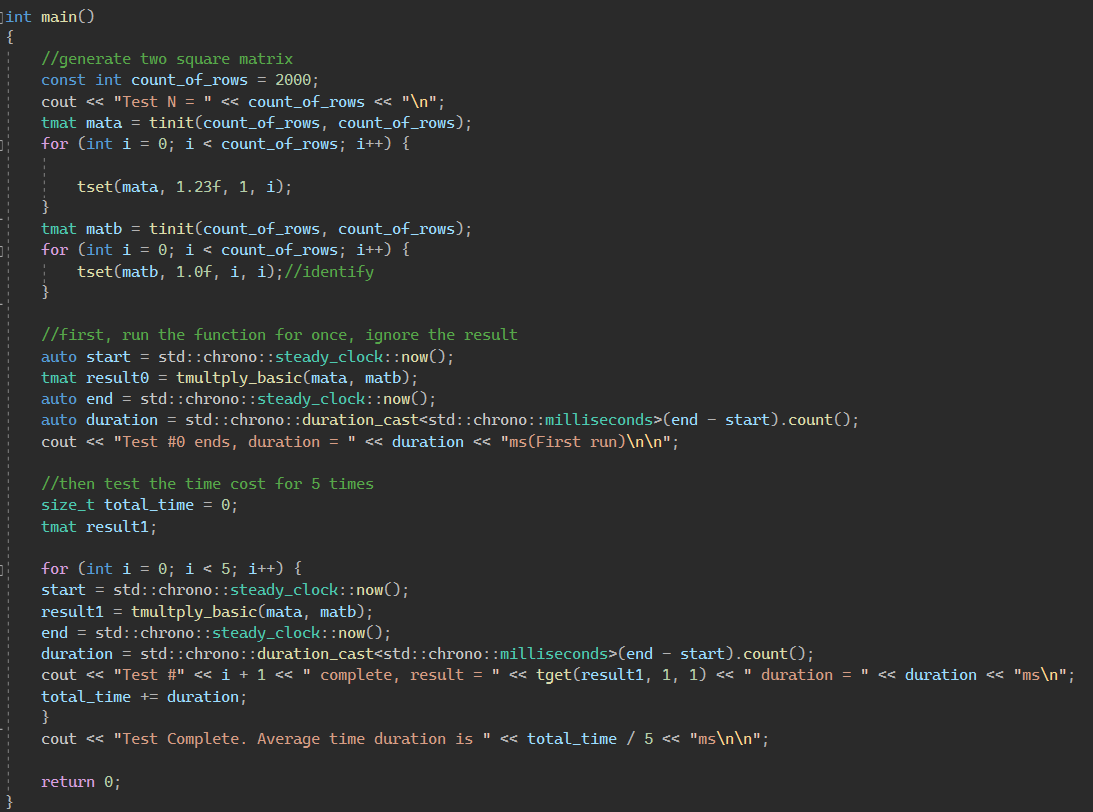
Test case #2:



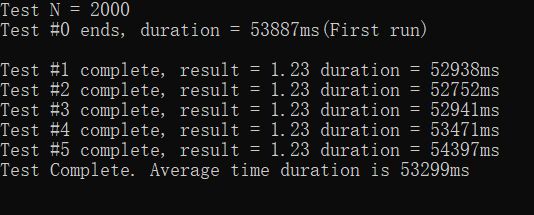
Run result:



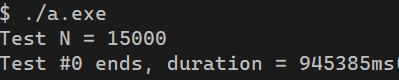
Test case #3



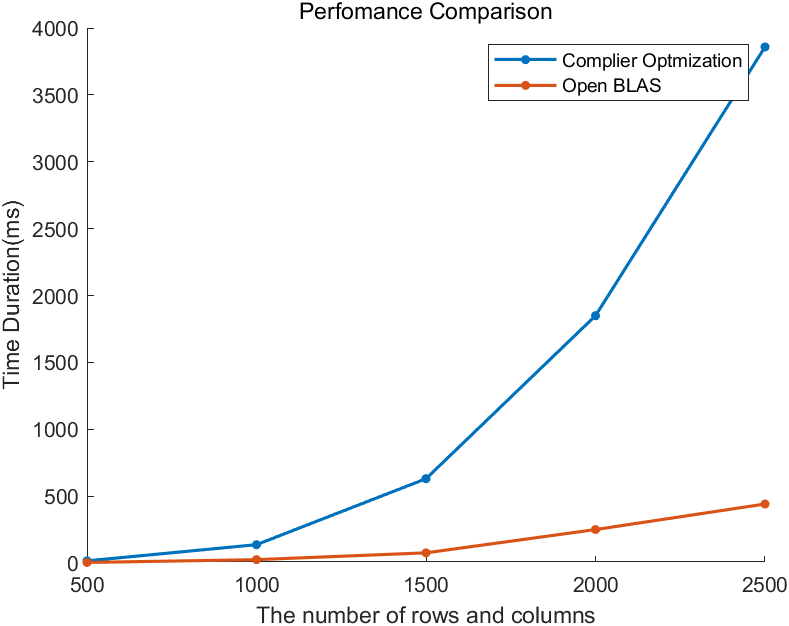
Run result:



This is 15000 \* 15000 = 225000000 elements’ matrix multiplication



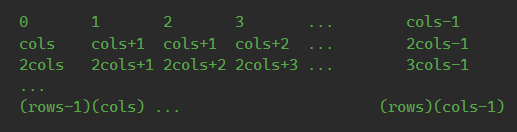
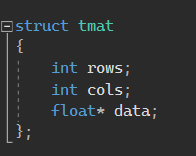
Test compare Open BLAS



## Part 3. Difficulties & Solutions, or others

**Part.1 use struct to represent a matrix**

In this part, I use a float\* to store a matrix in memory. The struct includes 3 member variables: rows and cols represent the number of rows and columns, \*data points to the float array in the memory. Code sizeof(**tmat**) will return 12.



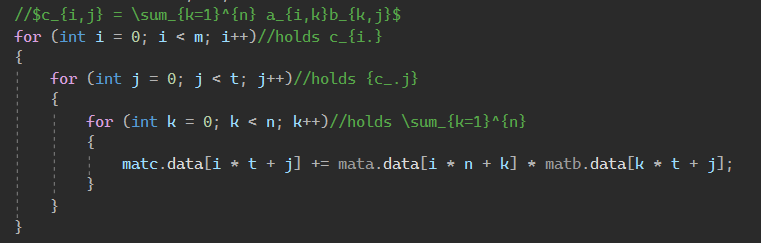
Since the data stores linear. The operations of matrices need to transform index pairs to index of pointer. According to the upright picture, the transformation from index pair to the pointer index is:

Knowing the upper relationship, we can access any matrix using the index pair or rewrite it.

**Part.2 Implement function computing the product of matrices**

The basic matrix computing algorithm is designed to be the same as we compute a product of matrix. That is breaking matrix production to multiples of vector-matrix productions, then breaking vector-matrix productions to multiples of vector scaling. The final algebraic result is:

The implementation of this algorithm is shown below.



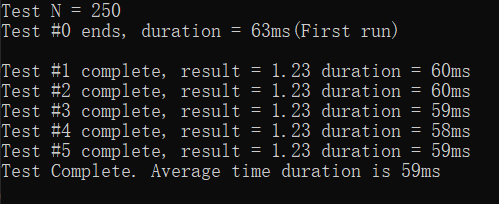
**Part.3 Performance Measure**

The code below will use to count how many milliseconds the algorithm costs.

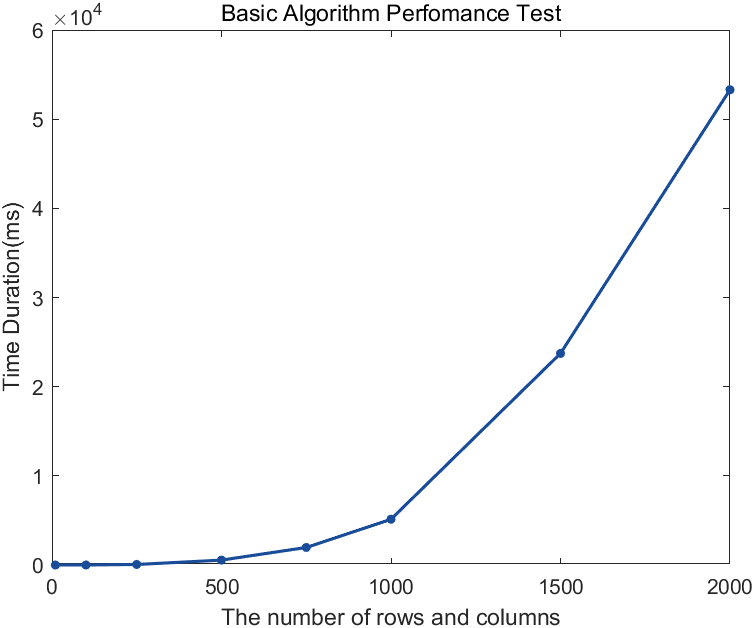
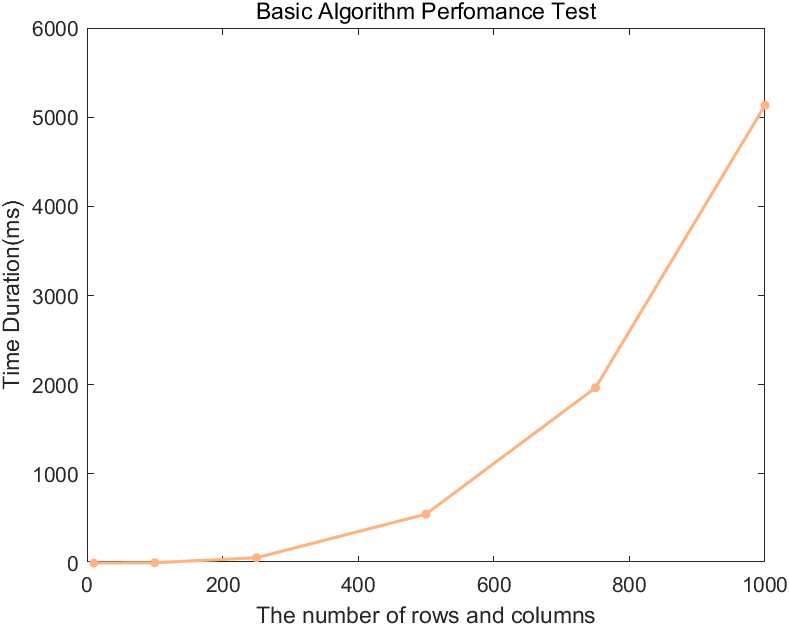
1. auto start = std::chrono::steady\_clock::now();
2. result = function(type parameters);
3. auto end = std::chrono::steady\_clock::now();
4. auto duration = std::chrono::duration\_cast<std::chrono::milliseconds>(end - start).count();
5. cout << "result = " << result << ", duration = " << duration << std::endl;

The time will be counted while program is processing the product of two square matrices. If the number of columns of the square is **n**, the time complexity of this algorithm can be easily figure out, it’s . Note that one of two square matrices are identify matrix and the other one has **1.23f** in the second row and **0.00f** in other rows.

Here’s a result of two matrices with n = 250 (125,000 elements). The compilation is completed automatically with default settings by the Visual Studio.



The average time duration will be recorded. The graph below will show how this algorithm costs time as the number of **n** increases.



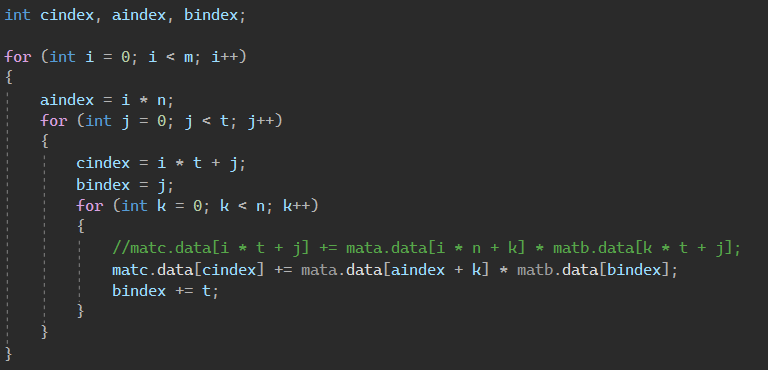
Left figure shows time duration up to n = 1000, right picture shows time duration up to n = 2000 (40,00,000 elements in total).

**Part.4 Optimization**

Matrix multiplication has time complexity using iterative or divide-and-conquer algorithm. Though it’s proved that there exist algorithm, but those algorithms lead to unacceptable rounding-errors and small range of usage. The way of optimization will focus on the aspect of computer programming.

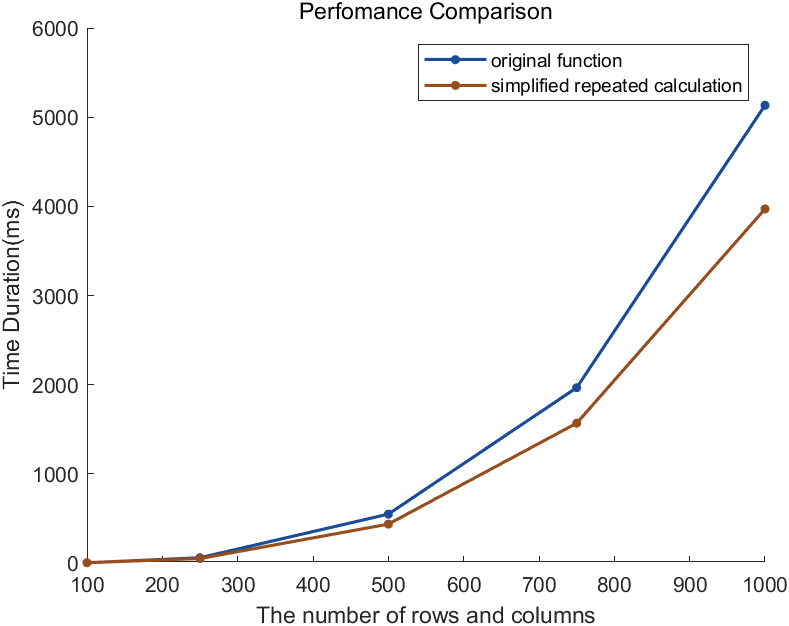
Idea 1: Simplify Repeated Calculation

When calculating , Every time calculate the index of and is necessary. Since computing multiplications costs a lot time than computing addition, the time cost in linear index search can be reduced.



The original code is written in the annotation. The variable aindex, bindex, cindex are set to save the well-calculated data in order to reduce calculation and for bindex, the += operator is for replacing the multiplication.

The calculation of aindex and cindex are reduced from to and . Faster index changing is expected.

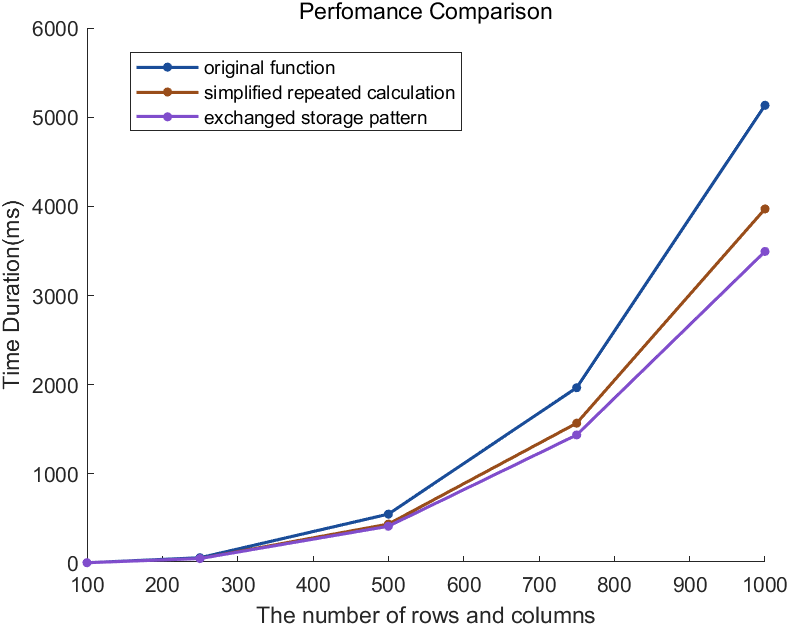


The improved function has improved 20% efficiency compared to the original one, more time saved expected as the number of n grows larger.

Idea 2: Change Storage Pattern

We compute matrix multiplications column-by-column instead of row-by-row. This leads to the idea of changing the storage pattern in memory. In Part.1, the matrix is stored row-by-row instead column-by-column. Exchanging the storage pattern can make the columns consecutive.

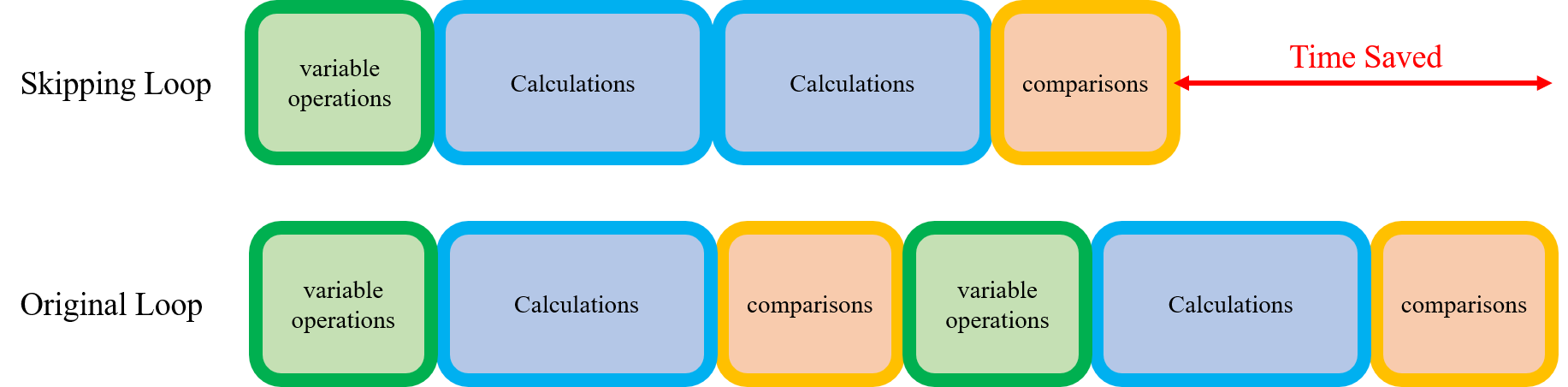
The pattern is shown below:

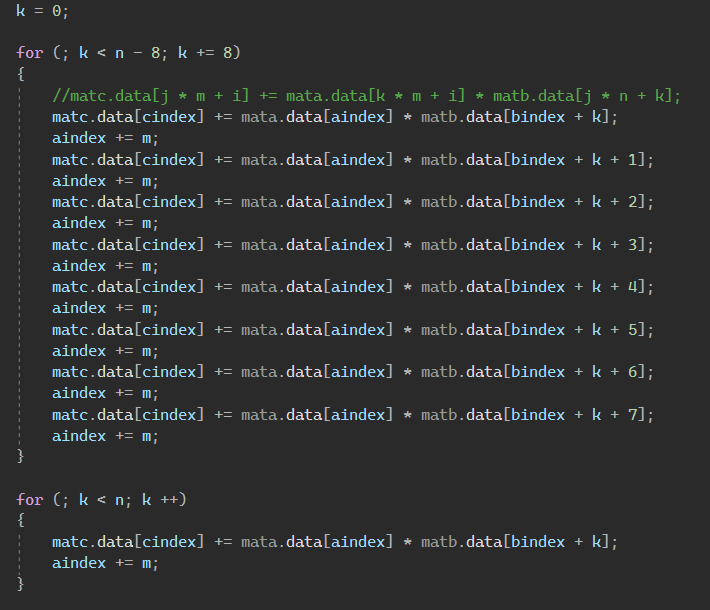


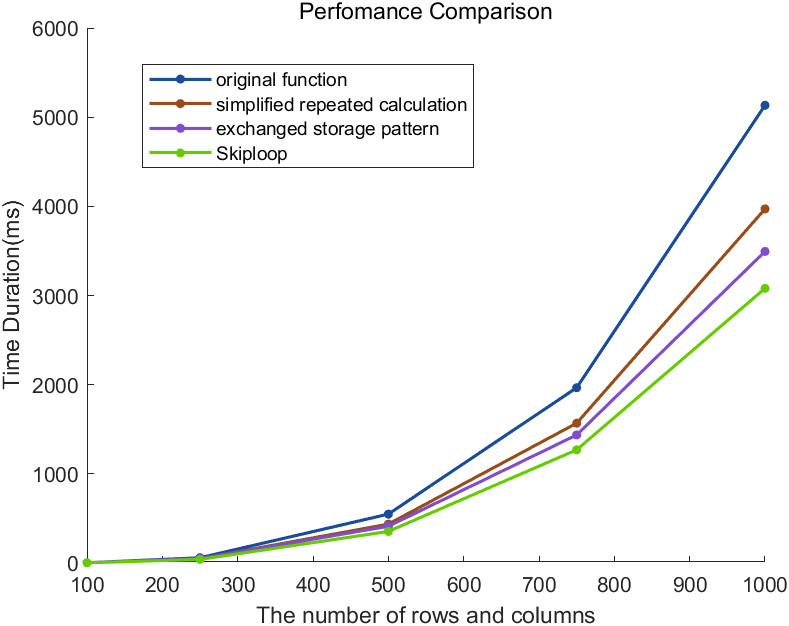
The result is shown above. Exchanging rows and columns is a slight change but still does 10% effect to the time duration. That shows the importance of the I/O of the memory. Furthermore, this pattern of storage makes the following optimization idea easier to achieve.

Idea 3: Skipping Loop

For-loop and while-loop takes a lot of time. Each loop an unnecessary variable addition and comparison takes a small amount of time but it will accumulate. The idea and the implementation are shown below.

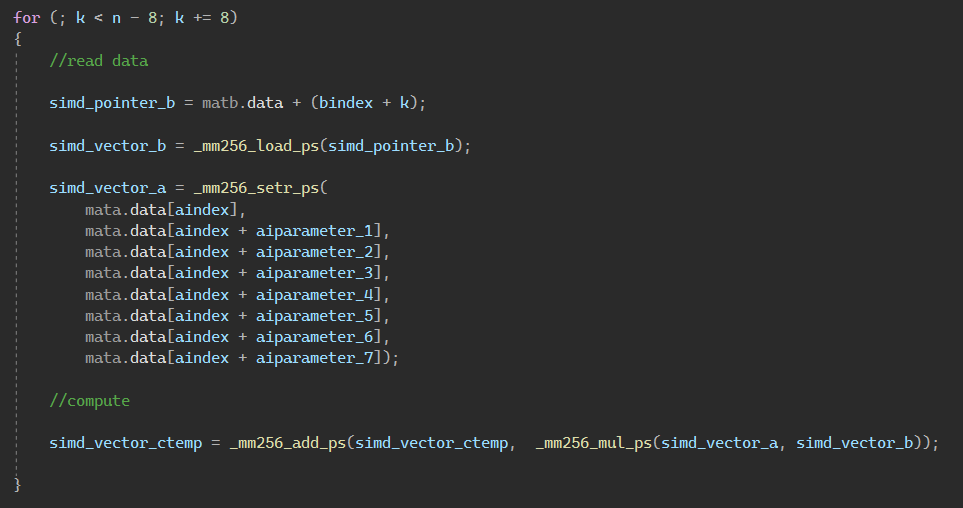


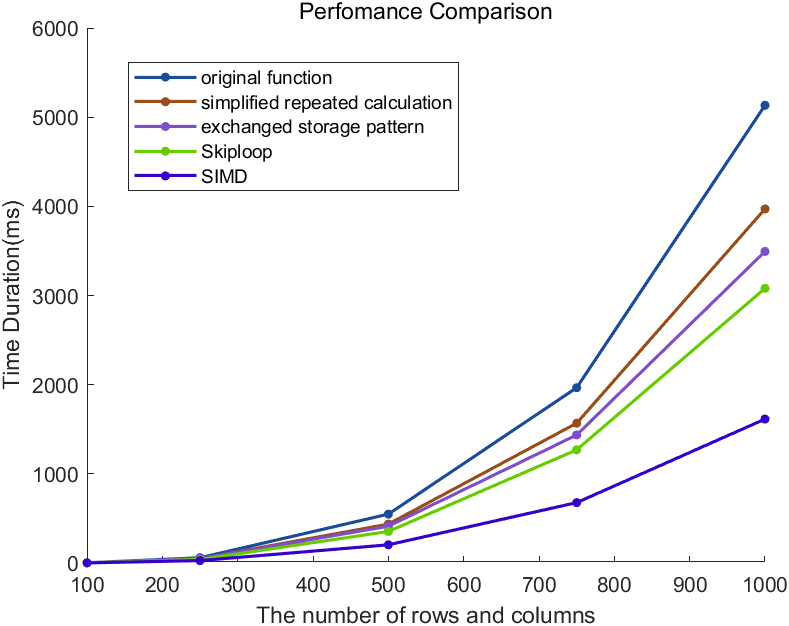




This is the preparation of the next step of optimization.

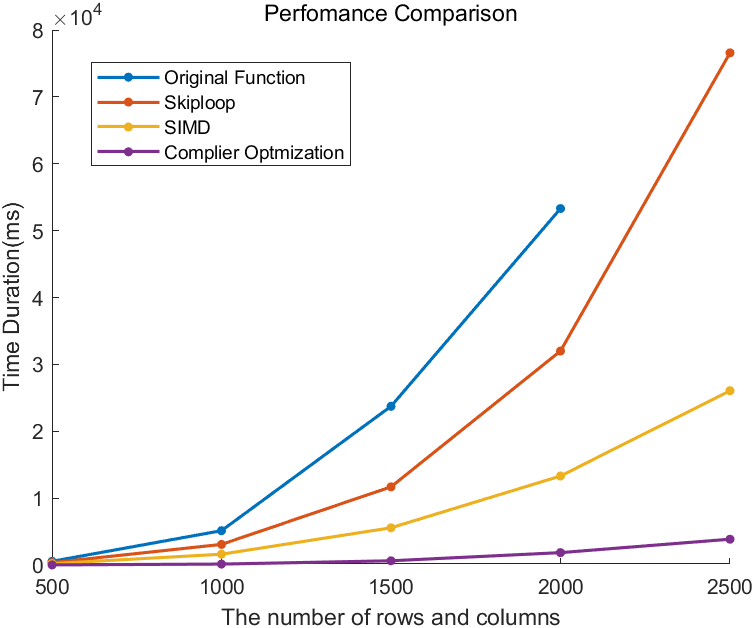
Idea 4: SIMD (intel AVX2)

Change the code in the skip-for-loop to SIMD operations. For one operation, 8 values can be multiplied at once. The expected efficiency promotion is 8 times, but the result shows it is 2 time faster than the Skiploop idea. 



Idea 5: Compiler Optimization

Use g++ -O3 to complete the compiler optimization.



Here is the final result. Computing two matrix multiplication with n=2500 (6,250,000 elements) is now 20x faster.

**Part.4 Comparison With Open BLAS**

