MATRIX-VECTOR-MULTIPLY ANALYSIS

SEQUENTIAL, THREAD, BLAS

Author
Xiu Zhou

1/19/2024

Contents

1. Results Correctness Explanation

1.1. Method for Getting Correct Results

In the implementation programs (matrixVectorMutiply_S.cpp and matrixVectorMultiply_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 (1×10−6 (or 1e−6), commonly used default in numerical computing), the results of my solutions are correct, or are not correct.

Formula:

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

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

1.2. Getting Residual Tool Function

```
// Function to calculate the residual between two vectors
double calculateResidual(const vector<double>& Y_BLAS, const vector<double>& Y_My) {

  // Assuming Y_BLAS and Y_My have the same size
  int n = Y_BLAS.size();

  // Calculate the residual
  double residual = 0.0;
  for (int i = 0; i < n; ++i) {
    double diff = Y_BLAS[i] - Y_My[i];
    residual += diff * diff;
  }

  return sqrt(residual);

}
```

1.3. Specific Application in Sequential Solution

1.3.1. In the main() part of matrixVectorMutiply_S.cpp

```
const double tolerance = 1e-15; // For residual tolerance checking
bool withinTolerance = false; // For record sequential residual tolerance status

double flops1 = 0;
double residual1 = 0;
chrono::duration<double, milli> elapsedMilliseconds1 = std::chrono::milliseconds(1);
generateSequentialResult(A, X, Y, Z, dimension, flops1, residual1, elapsedMilliseconds1);
outputFile << flops1 << "  ";
// Write residual of  matrix-vector-multiply sequential solution and BLAS into the outputFile
outputFile << residual1 << "  ";
// Check the tolerance
if (residual1 < tolerance) {
   withinTolerance = true;
}
else {
   withinTolerance = false;
}
// Write residual tolerance status into into the outputFile, ture (1) for within tolerance false (0 ) for not
outputFile << withinTolerance << "\n";
```

1.3.2. In the generateSequentialResult() tool function of matrixVectorMutiply_S.cpp

❖ generateSequentialResult's interface introduction:

```
void generateSequentialResult(const vector<vector<double>>& A, const vector<double>& X, vector<double>& Y1, vector<double>& Z, int dimension, double& flops, double& residual, chrono::duration<double, milli>& elapsedMilliseconds1)
```

- A is 2-D vector parameter, pass a matrix;
- X is 1-D vector parameter, pass a vector that is multiplied by the matrix A;
- Y1 is 1-D vector parameter, used to store the results of A * X operation by Sequential Solution;
- Z is 1-D vector parameter, pass a vector that contains the results of A * X operation by BLAS;
- D is an integer parameter, pass the dimension;
- residual is a double parameter, used to store the residual result between Sequential Solution and BLAS;
- elapsedMilliseconds1 is a duration parameter, unit is millisecond, used to store the consuming time of specific A * X operation by Sequential Solution, here is the consuming of running multySequential() function.

- ❖ Key code module

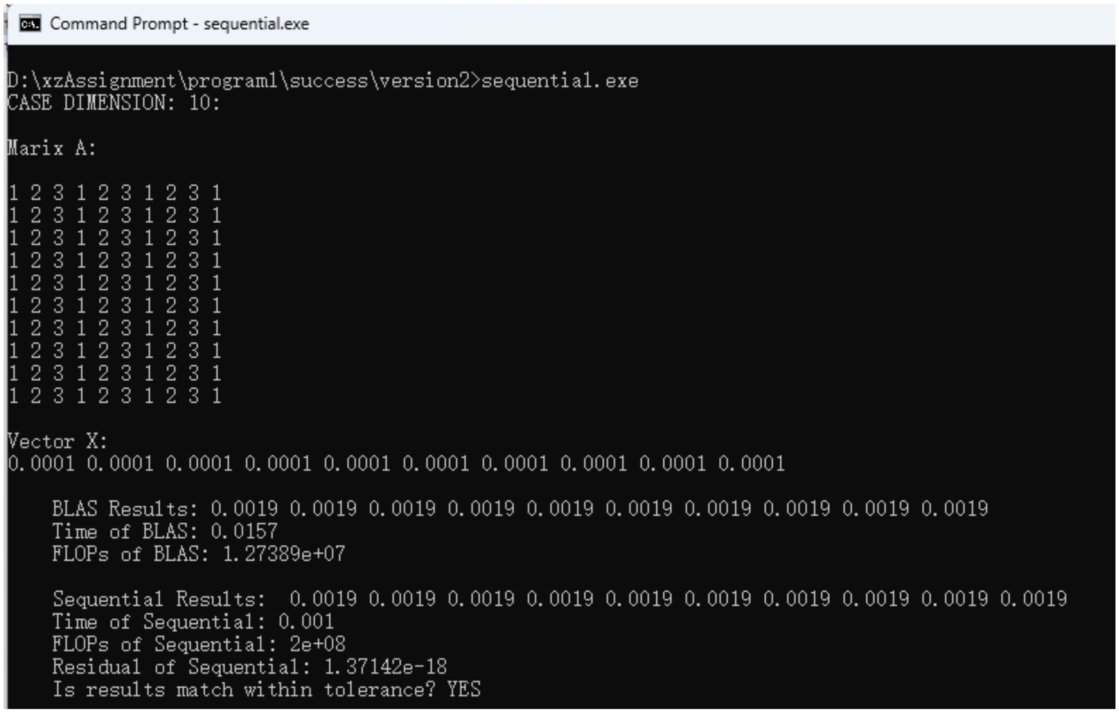
```
void generateSequentialResult(const vector<vector<double>>& A, const vector<double>& X, vector<double>&
Y1, vector<double>& Z, int dimension, double& flops, double& residual, chrono::duration<double, milli>&
elapsedMilliseconds1) {
   // Measure the start time, Time (Milliseconds)
   auto startTime = chrono::high_resolution_clock::now();
   // Perform matrix-vector multiplication
   multySequential(A, X, Y1, dimension);

   // Measure the end time
   auto endTime = chrono::high_resolution_clock::now();
   chrono::duration<double, milli> elapsedMilliseconds = endTime - startTime;
   elapsedMilliseconds1 = elapsedMilliseconds;
   // Calculate FLOPS
   double flops1 = (2.0 * dimension * dimension) / (elapsedMilliseconds1.count() / 1000.0);
   flops = flops1;
   // Get residual of matrix-vector-multiply Sequential solution and BLAS
   double residual1 = 0.0;
   residual1 = calculateResidual(Z, Y1);
   residual = residual1;
}
```

1.3.3. Screen Shots of Partial Result Screen Output

Because it is hard to output all the results to the screen, the program just output the results when dimension <= 60.

In the screen, the residual is 1.37142e-18, which is less than the tolerance mentioned in the “1.1. Method for Getting Correct Results”.

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



```
D:\xzAssignment\program1\success\version2>sequential.exe
CASE DIMENSION: 10:

Marix A:

1 2 3 1 2 3 1 2 3 1
1 2 3 1 2 3 1 2 3 1
1 2 3 1 2 3 1 2 3 1
1 2 3 1 2 3 1 2 3 1
1 2 3 1 2 3 1 2 3 1
1 2 3 1 2 3 1 2 3 1
1 2 3 1 2 3 1 2 3 1
1 2 3 1 2 3 1 2 3 1
1 2 3 1 2 3 1 2 3 1
1 2 3 1 2 3 1 2 3 1

Vector X:
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001

    BLAS Results: 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019
    Time of BLAS: 0.0157
    FLOPs of BLAS: 1.27389e+07

    Sequential Results:  0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019
    Time of Sequential: 0.001
    FLOPs of Sequential: 2e+08
    Residual of Sequential: 1.37142e-18
    Is results match within tolerance? YES
```

Fig 1, screen shots Sequential Solution partial results screen output, dimension = 10

1.3.4. All Residuals

In each line of the data file “matrix_vector_sequential_g++_4.txt”, the last number is to show the value of residual tolerance status. In each line of this file, this number is always 1, which mean that the results of matrix-vector multiplication calculated by Sequential Solution are correct. Please check the raw data file “matrix_vector_sequential_g++_4.txt” in the “data\4core_myComputer” folder.

1.4. Specific Application in Thread Solution

1) in the main() part of “matrixVectorMultiply_P.cpp”:

```
    const double tolerance = 1e-15; // For residual tolerance checking
    bool withinTolerance = false; // For record sequential residual tolerance status

    int num_threads = 8;
    // Declaring threads
    vector<thread> threads;
    // Declaring core #
    // When running other 8-core computer, coreNumber will be 8
    int coreNumber = 0;
    // Measure the start time, Time (Milliseconds)
    auto startTime1 = chrono::high_resolution_clock::now();

    // Creating threads, each evaluating its own part
    int chunk_size = rows / num_threads;
    for (int i = 0; i < num_threads; ++i) {
        int start = i * chunk_size;
        int end = (i == num_threads - 1) ? rows : (i + 1) * chunk_size;
        threads.push_back(thread([=]() { multyThread(cols, start, end, coreNumber); }));
        // For 4-core computer, update coreNumber to ensure two threads share the same core
 coreNumber = (coreNumber + 1) % (num_threads / 2);

 // For 8-core computer, thread number is 8, 1 thread per core.
 //coreNumber
    }

    // Joining and waiting for all threads to complete
    for (auto& t : threads) {
        t.join();
    }
```

```
    // Calculate time consuming of metrix-vector-multiply thread solution
    chrono::duration<double, milli> elapsedMilliseconds1 = endTime1 - startTime1;
    // Write FLOPS of BLAS into the outputFile
    double flops1 = (2.0 * rows * cols) / (elapsedMilliseconds1.count() / 1000.0);
    // Write FLOPs of metrix-vector-multiply thread solution into the outputFile
    outputFile << flops1 << "  ";
    // Get residual of matrix-vector-multiply thread solution and BLAS
    double residual1 = 0.0;
    for (int i = 0; i < rows; ++i) {
        double diff = fabs(Y1[i] - Z[i]);
        residual1 += diff * diff;
    }
    // Write residual of  matrix-vector-multiply thread solution and BLAS into the outputFile
    outputFile << residual1 << "  ";

    // Check the tolerance
    if (residual1 < tolerance) {
        withinTolerance = true;
    }
    else {
        withinTolerance = false;
    }
    // Write residual tolerance status into into the outputFile
    outputFile << withinTolerance << "\n";
```

2) In the multyThread() tool function of “matrixVectorMultiply_P.cpp”:

- multyThread’s interface introduction:

`void multyThread(int cols, int start, int end, int core)`

cols is an integer parameter, pass the size of Y1(1-D vector);

start is an integer parameter, pass the start row index for the current thread to take charge of;

end is an integer parameter, pass the end row index for the current thread to take charge of;

core is an integer parameter, pass the core number of computer;

- Key code

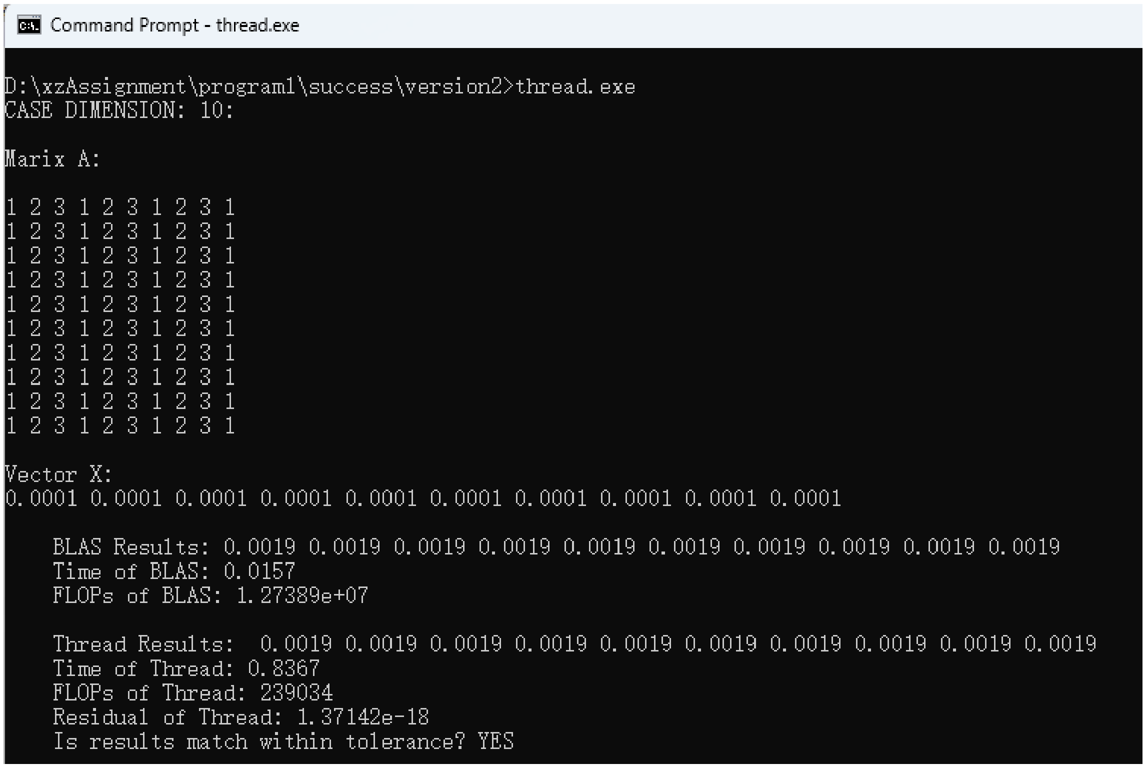
```
// Function to do Matrix-Vector multiplication for thread solution
void multyThread(int cols, int start, int end, int core) {

    // Set affinity for the current thread
    DWORD_PTR mask = 1ull << core;
    SetThreadAffinityMask(GetCurrentThread(), mask);

    for (int i = start; i < end; ++i) {
        Y1[i] = 0;
        for (int j = 0; j < cols; ++j) {
            Y1[i] += A[i][j] * X[j];

            //If want to see affinitation works, can cout this line
            //std::cout << "Thread " << core << ": Y1[" << i << "] = " << Y1[i]
<< std::endl;

        }
    }
}
```

1.4.1. Screen Shots of Partial Result Screen Output

Because it is hard to output all the results to the screen, the program just output the results when dimension <= 60.

In the screen, the residual is 1.37142e-18 (see Fig 2), which is less than the tolerance mentioned in the "1.1. Method for Getting Correct Results".

"YES" means the result of residual is match within tolerance, so dimension-10 matrix * vector's result is correct.



```
D:\xzAssignment\program1\success\version2>thread.exe
CASE DIMENSION: 10:

Marix A:

1 2 3 1 2 3 1 2 3 1
1 2 3 1 2 3 1 2 3 1
1 2 3 1 2 3 1 2 3 1
1 2 3 1 2 3 1 2 3 1
1 2 3 1 2 3 1 2 3 1
1 2 3 1 2 3 1 2 3 1
1 2 3 1 2 3 1 2 3 1
1 2 3 1 2 3 1 2 3 1
1 2 3 1 2 3 1 2 3 1
1 2 3 1 2 3 1 2 3 1

Vector X:
0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001

    BLAS Results: 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019
    Time of BLAS: 0.0157
    FLOPs of BLAS: 1.27389e+07

    Thread Results:  0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019 0.0019
    Time of Thread: 0.8367
    FLOPs of Thread: 239034
    Residual of Thread: 1.37142e-18
    Is results match within tolerance? YES
```

Fig 2, screen shots Thread Solution partial results screen output, dimension = 10

1.4.2. All Residuals

In each line of the data file "matrix_vector_thread_affini_g++_4.txt", the last number is to show the value of residual tolerance status. In each line of this file, this number is always 1, which mean that the results of Thread Solution are correct. Please check the details in the file "matrix_vector_thread_affini_g++_4.txt" of the "data\4core_myComputer" folder.

2. Results Analysis Based on Plots

2.1. Three Solution's Flops Plot and Analysis (by 4-core & g++ complier)

2.1.1. Using Raw Data Files

The raw data coming from data folder, includes:

- matrix_vector_blas_g++_4.txt;
- matrix_vector_sequential_g++_4.txt ;
- matrix_vector_thread_affini_g++_4.txt

2.1.2. Plot of Three Solutions (4-core & g++ complier)

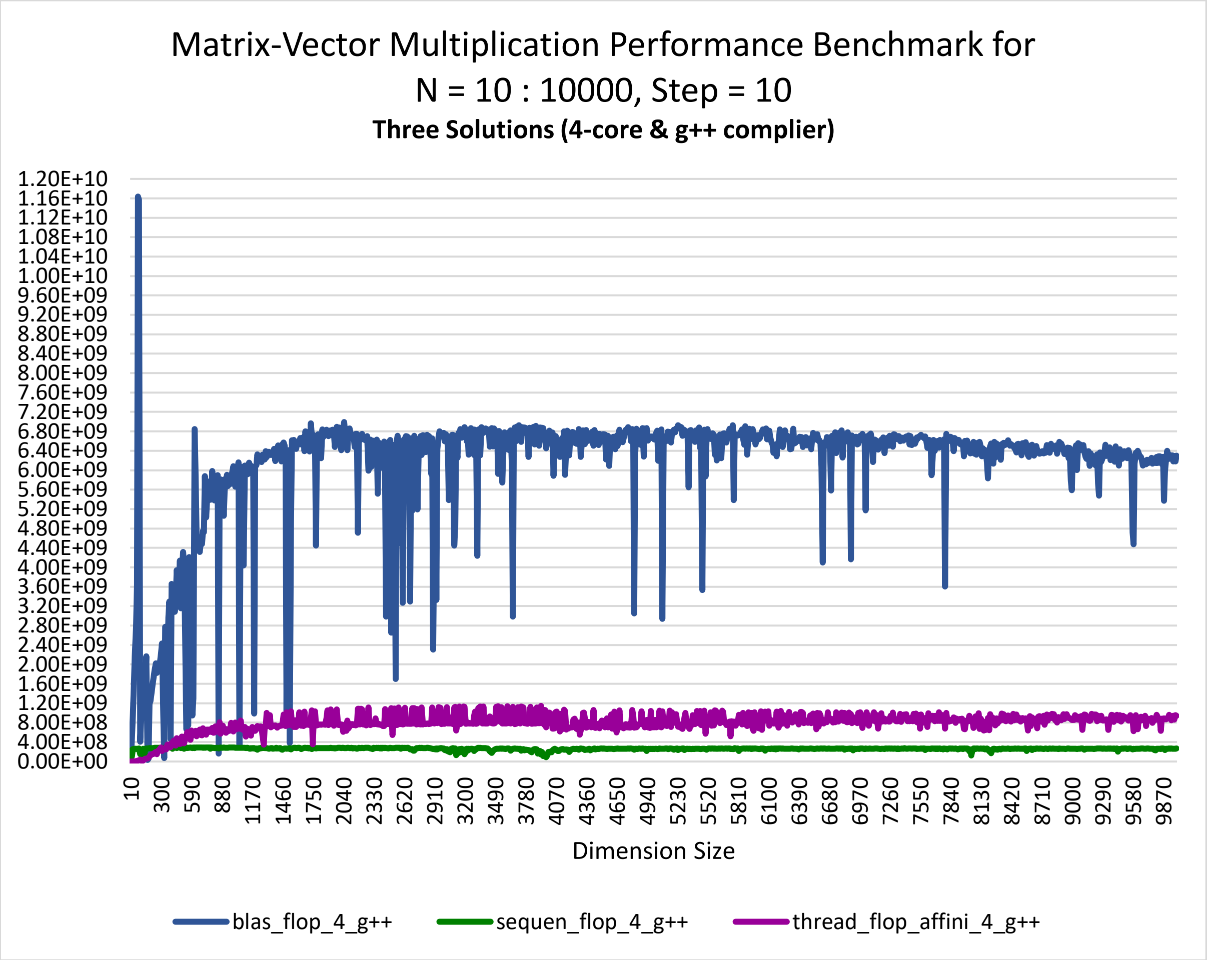


Fig 3, Plot of Three Solutions (4-core & g++ complier)

2.1.3. Analysis

From Fig 3, it is obvious that BLAS is much faster than Sequential and Thread. There are several reasons leaded this result.

2.1.3.1. Sequential Solution matrix-vector multiplication slow analysis

```
// Function to do Matrix-Vector multiplication for Sequential solution
void multySequential(const vector<vector<double>>& A, const vector<double>& X, vector<double>& Y1, int
dimension) {

   for (int i = 0; i < dimension; ++i) {
      Y1[i] = 0;
      for (int j = 0; j < dimension; ++j) {
         Y1[i] += A[i][j] * X[j];
      }
   }
}
```

From the above multySequentia() code, my sequential implementation uses a straightforward nested loop to perform matrix-vector multiplication.

The **Strengths** is simplicity, which means the code is straightforward and easy to understand.

Weaknesses is Limited Parallelism, which means the sequential implementation doesn't take advantage of multi-core processors and it runs on a single core, so it won't benefit from parallelism.

2.1.3.2. Threaded Implementation matrix-vector multiplication slow analysis

```
// Function to do Matrix-Vector multiplication for thread solution
void multyThread(int cols, int start, int end, int core) {
   // Set affinity for the current thread
   DWORD_PTR mask = 1ull << core;
   SetThreadAffinityMask(GetCurrentThread(), mask);

   for (int i = start; i < end; ++i) {
      Y1[i] = 0;
      for (int j = 0; j < cols; ++j) {
         Y1[i] += A[i][j] * X[j];

         //If want to see affinitation works, can cout this line
         //std::cout << "Thread " << core << ": Y1[" << i << "] = " << Y1[i] << std::endl;
      }
   }
}
```

```
// Keep same thread number in my 4-core computer or other 8-core computer
int num_threads = 8;
// Declaring threads
vector<thread> threads;
// Declaring core #
// When running other 8-core computer, coreNumber will be 8
int coreNumber = 4;
// Creating threads, each evaluating its own part
int chunk_size = rows / num_threads;
for (int i = 0; i < num_threads; ++i) {
   int start = i * chunk_size;
   int end = (i == num_threads - 1) ? rows : (i + 1) * chunk_size;
   threads.push_back(thread([=]() { multyThread(cols, start, end, coreNumber); }));
   coreNumber++;
}

// Joining and waiting for all threads to complete
for (auto& t : threads) {
   t.join();
}
```

From the above main module code and multyThread () code, my threaded implementation divides the workload across 8 threads using a simple loop, each thread performs matrix-vector multiplication for its assigned range, the affinity is set for each thread using SetThreadAffinityMask. The limite thread number shows why it climbs to some location, then keep stable.

The **Strengths** includes:

A) Parallelism, which means that distributing the workload across multiple threads allows concurrent execution, potentially utilizing multiple cores.

B) Affinity Setting: Setting thread affinity can help bind threads to specific cores, which may optimize cache usage.

The **weaknesses** includes:

A) Mutex Usage: The use of a mutex (std::lock_guard) may introduce contention and limit the performance gain from parallelism, especially for small matrix sizes.

B) Affinity Setting Overhead: Setting thread affinity might have overhead and may not always result in performance improvement, especially for smaller matrix sizes.

2.1.3.3. BLAS Implementation matrix-vector multiplication fast analysis

BLAS uses several main strategies to improve speed and efficiency, includes: A) **Low-Level Optimizations**: BLAS libraries are typically implemented using highly optimized, low-level routines that take advantage of specific processor features, such as SIMD (Single Instruction, Multiple Data) instructions. These instructions enable parallel processing of multiple data elements in a single instruction, improving computational efficiency. B) **Cache Efficiency**: BLAS libraries are designed to optimize memory access patterns, minimizing cache misses and maximizing data reuse. This is crucial for performance, especially when dealing with large matrices. C) **Parallelization**: BLAS libraries often implement parallelization techniques to distribute the workload across multiple cores or processors. This can significantly improve performance, especially on multi-core systems.

2.1.4. Enlargement of Plot of Three Solutions (4-core & g++ complier)

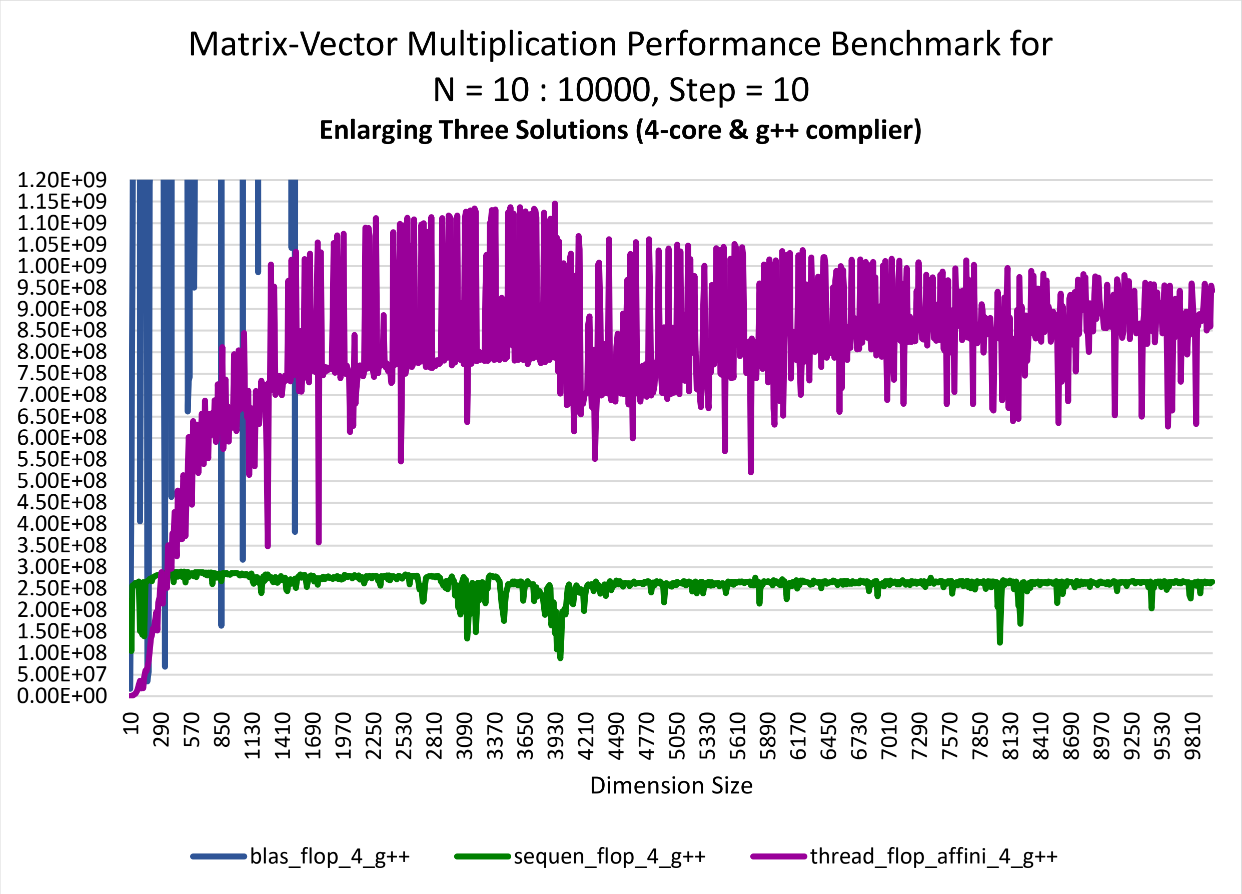


Fig 4, Enlarging Plot of Three Solutions (4-core & g++ complier)

2.1.4.1. Analysis for BLAS jump-climbing at beginning phrase

The slow performance of BLAS at the beginning phase, especially the first call, could be attributed to a few factors:

1) Initialization Overhead: BLAS libraries might have some initialization overhead, especially during the first calls. This could involve setting up internal structures, memory allocation, or other initialization routines.
2) Dynamic Library Loading: If BLAS is dynamically linked, there could be some initial loading and linking overhead when the library is first accessed.
3) Optimization for Larger Workloads: BLAS libraries are often optimized for larger matrix sizes and may have specialized routines or optimizations that are more effective when dealing with larger datasets.
4) Cache Warm-up: The initial calls to BLAS might not benefit from a warm cache. As the library processes more data, it can exploit cache efficiency, leading to improved performance.

2.1.4.2. Analysis for Sequential Solution Keeping Stable Performance

The stability of the sequential solution's performance could be influenced by the following factors:

1) Consistent Workload: The sequential solution has a consistent workload, and its performance is likely not affected by factors such as thread creation overhead or affinity settings. So at the beginning, it doesn't need load too much data, its performance has no sharply climbing.
2) Optimization by Compiler: The compiler might be able to apply optimizations more effectively in the sequential solution, resulting in stable and predictable performance across different matrix sizes.
3) Low Overhead: The absence of parallelization-related overhead in the sequential solution contributes to its stable performance.

2.1.4.3. Analysis for Thread Solution with affinization climbing at beginning phrase

The climbing performance of the thread solution with affinization at the beginning phase could be be influenced by the following factors:

1) Affinization Impact: Setting thread affinity can influence initial performance, especially on systems with multiple cores. Affinization ensures that threads are assigned to specific cores, and the initial phase may involve better cache utilization and reduced contention.
2) Overhead Dominated by Computation: As the matrix size increases, the overhead of thread creation, synchronization, and affinity setting becomes less significant compared to the overall computation. This leads to a performance climb as the benefits of parallelization become more pronounced.
3) Cache Warm-up and Load Balancing: The climbing phase might be associated with cache warm-up effects and better load balancing among threads. As the matrix size increases, the workload becomes more suitable for parallel execution. Then the performance keeps stable.

2.2. Three Solution's Flops Plot and Analysis (g++_VS_VisualComplier on 4core)

2.2.1. Using Raw Data Files

The raw data coming from data folder, includes:

- matrix_vector_blas_g++_4.txt;
- matrix_vector_sequential_g++_4.txt ;
- matrix_vector_thread_affini_g++_4.txt
- matrix_vector_blas_4.txt;
- matrix_vector_sequential_4.txt
- matrix_vector_thread_affini_4.txt

2.2.2. Plot of Three Solutions (g++_VS_VisualComplier_4core)

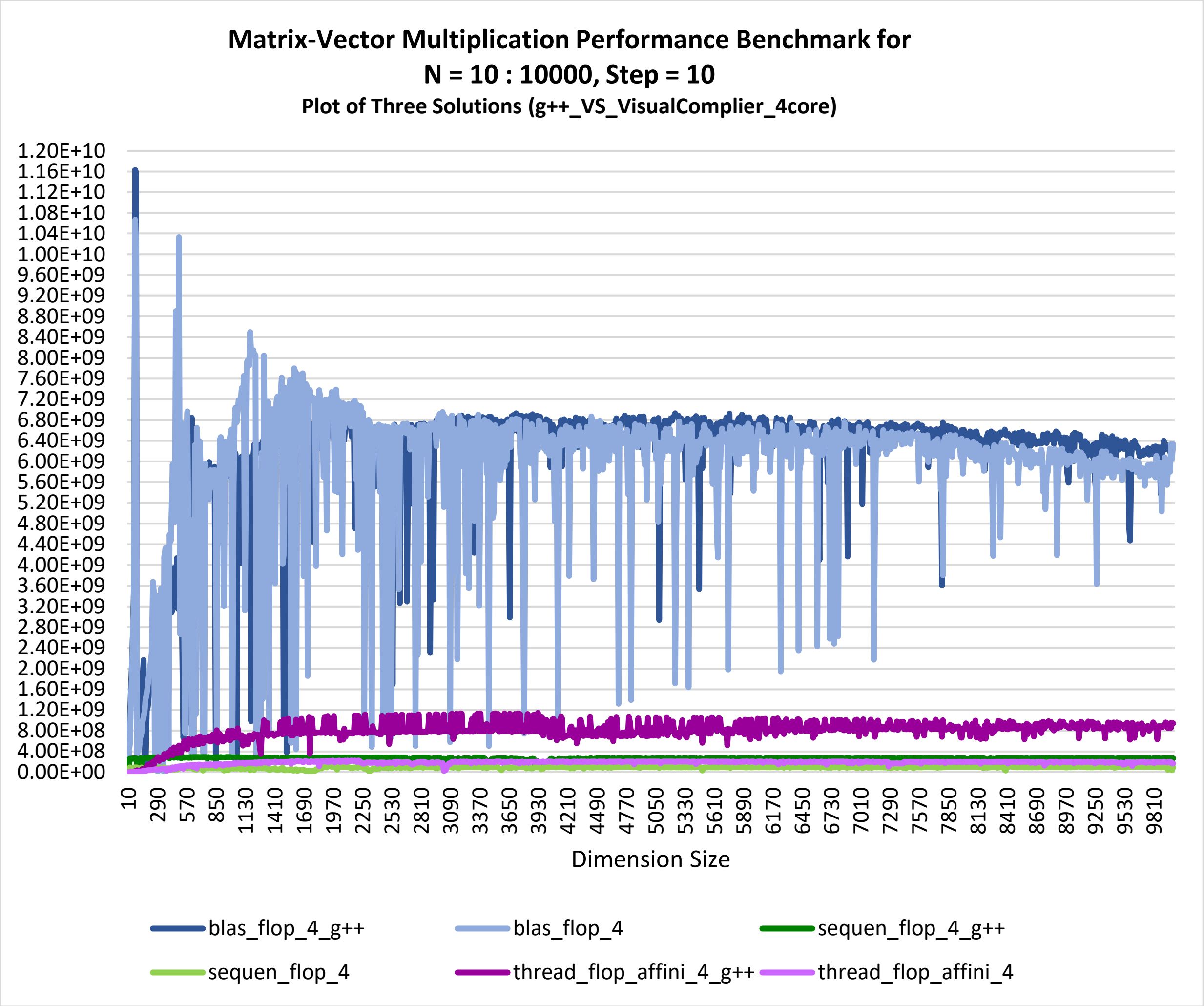


Fig 5, Plot of Three Solutions (g++_VS_VisualComplier_4core).

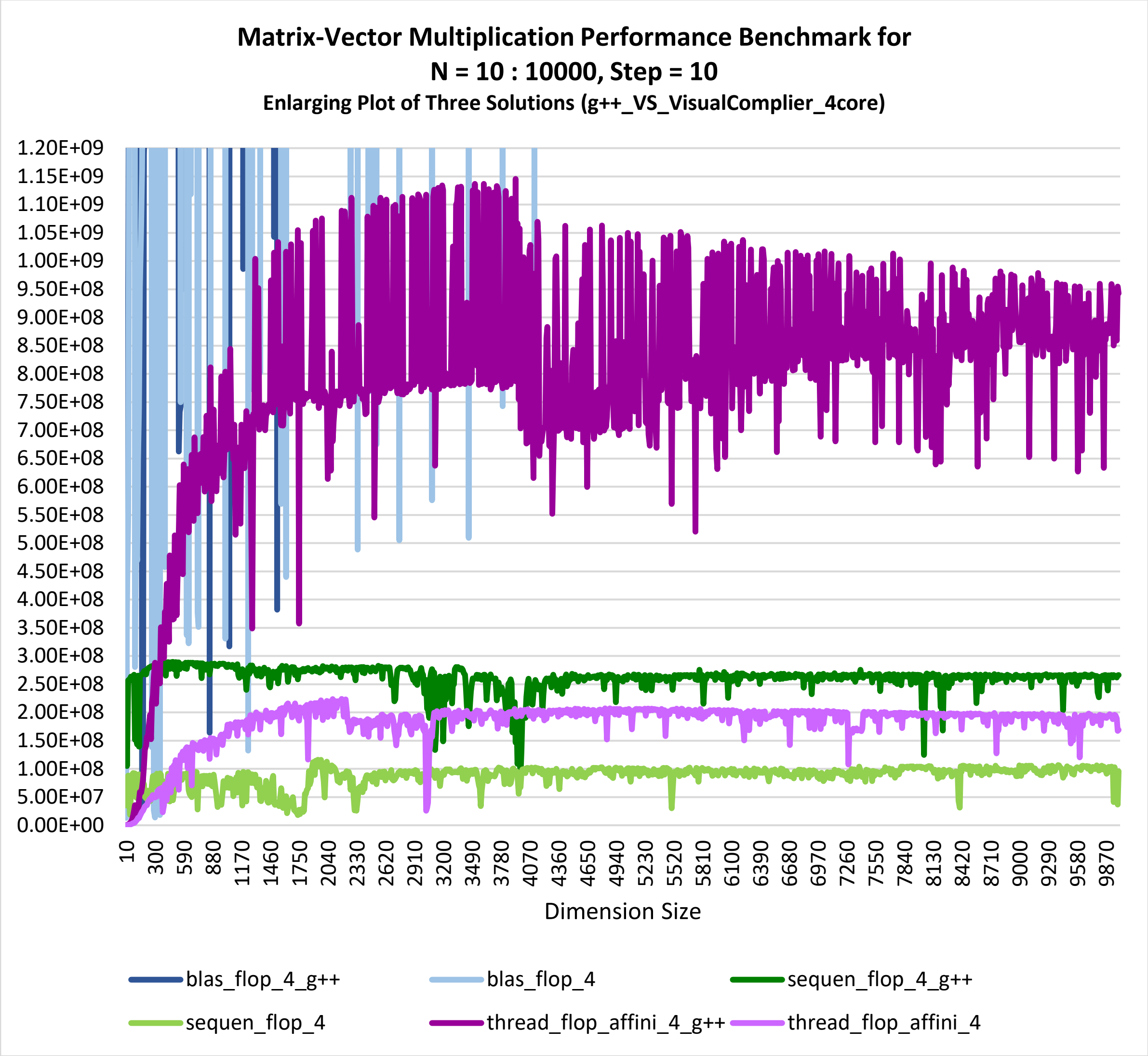


Fig 6, Enlarging Plot of Three Solutions (g++_VS_VisualComplier_4core).

2.2.3. Analysis for Plot of Three Solutions (g++_VS_VisualComplier_4core)

From Fig5 and Fig6, it is easy to observed that running three solutions using g++ complier is faster than running them using Visual Stodio Complier, especially for the Thread Solution. About this situation, there are several factors:

1) It's possible that the compiler used by G++ is more optimized than the one used by Visual Studio IDE.
2) The command line interface is more efficient than the IDE's graphical interface.
3) The way libraries are linked and the dependencies in g++ is faster than in Visual Studio.

3. Running Environment Declaring (CPU, System, IDE)

3.1. 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:
- L1 cache: 256K
- L2 cache: 1.0M
- L3 cache: 8.0M

3.2. Operating System of My Computer

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

3.3. Complier of Program

Use G++ for windows system. Use Visual Studio IDE complier too. Do comparation between them.

3.4. IDE

Use Visual Studio 2022 to compile these three solutions.

4. Calculate Theoretical Peak Performance

4.1. Formula

Theoretical Peak Performance (FLOPS)=Number of Cores * Clock Speed * Instructions per Cycle

1) Number of my computer cores: 4
2) Clock Speed of my computer: 2.11 GHz
3) Instructions per Cycle (IPC)
    a. Modern Intel processors often have a FLOP/Cycle of around 4 for single-precision floating-point operations (float) and 2 for double-precision floating-point operations (double).
    b. So, in this program, it should be 2.

4.2. Theoretical Peak Performance (FLOPS) of my computer

Theoretical Peak Performance (FLOPS) = 4 * 2.11(GHz) * 2 Operations per Cycle

= 16.88 GFLOPs

4.3. Analysis of My Solutions FLOPs (4core & g++) with Peak Performance (FLOPS)

4.3.1. Theoretical Peak Performance

My computer's theoretical peak performance is stated as 16.88 GFLOPs. This is the maximum performance my computer could achieve under ideal conditions.

4.3.2. Actual Performance Results

Sequential Solution: 0.0261 GFLOPs

Thread Solution: 0.0797 GFLOPs

BLAS Solution: 6.14 GFLOPs

4.3.3. Analysis

1) Sequential Solution: My sequential solution's average performance is significantly lower than the theoretical peak performance. It reveals that my sequential implementation is not fully utilizing the computational resources available on my computer.
2) Thread Solution: My threaded solution shows an improvement over the sequential solution but is still relatively low compared to the theoretical peak performance. It indicates that the multithreading is helping to some extent, but there might be room for further optimization.
3) BLAS Solution: The BLAS solution exhibits the highest performance among the tested implementations. BLAS libraries are highly optimized and leverage low-level optimizations, making them efficient for matrix-vector multiplication. The performance of BLAS is closer to the theoretical peak, indicating its effectiveness.

5. Other Analysis

5.1. Performance Analysis on different computers (4core_AMD VS 8core_Intel)

5.1.1. Using Raw Data Files

The raw data coming from data folder, includes:

- matrix_vector_blas_4.txt;
- matrix_vector_sequential_4.txt

- matrix_vector_thread_affini_4.txt
- matrix_vector_blas_8.txt;
- matrix_vector_sequential_8.txt
- matrix_vector_thread_affini_8.txt

5.1.2. Plot of Three Solutions (4core_AMD VS 8core_Intel)

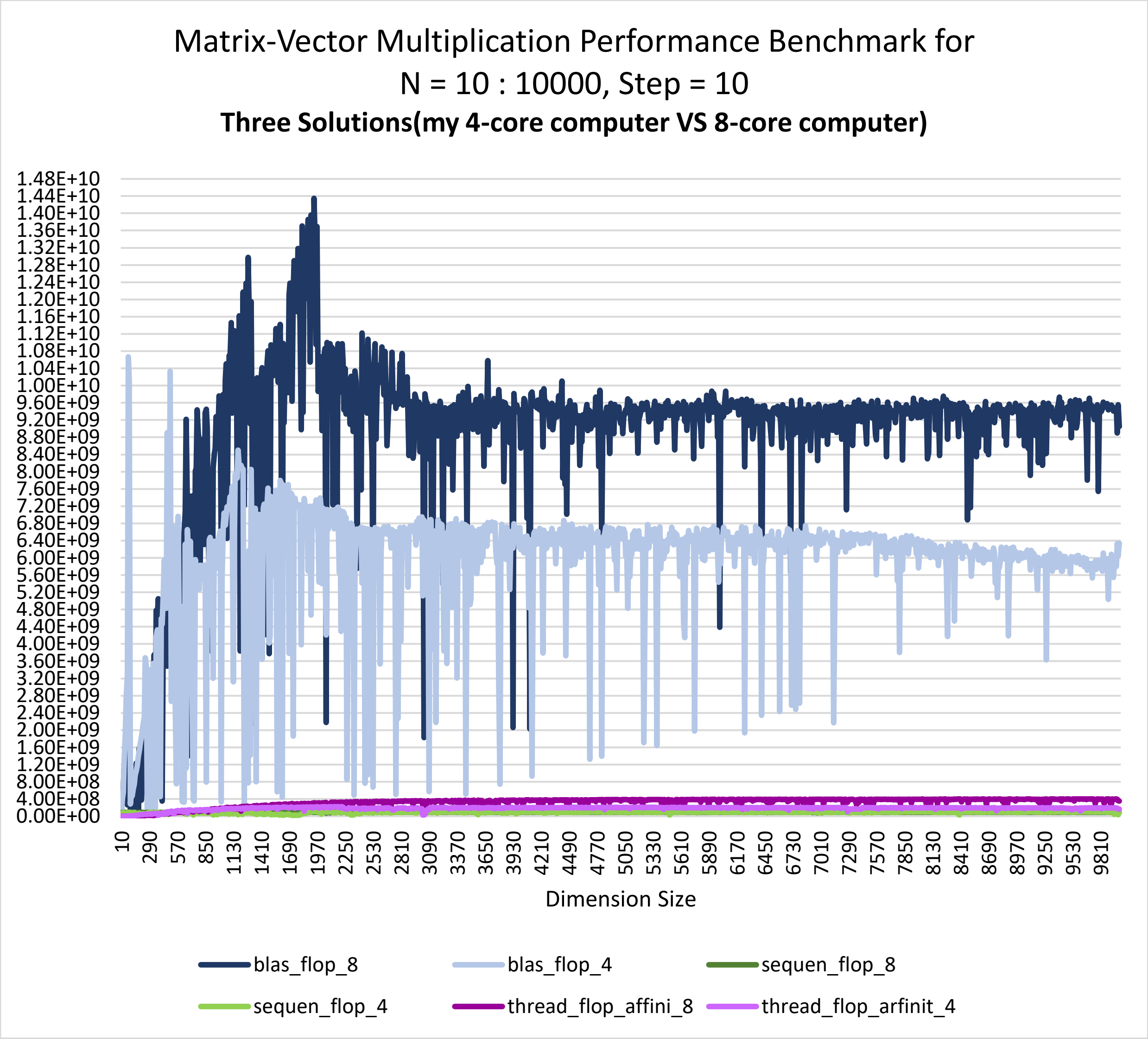


Fig 7, Plot of Three Solutions (4core_AMD VS 8core_Intel).

5.1.3. Enlarging Plot of Three Solutions (4core_AMD VS 8core_Intel)

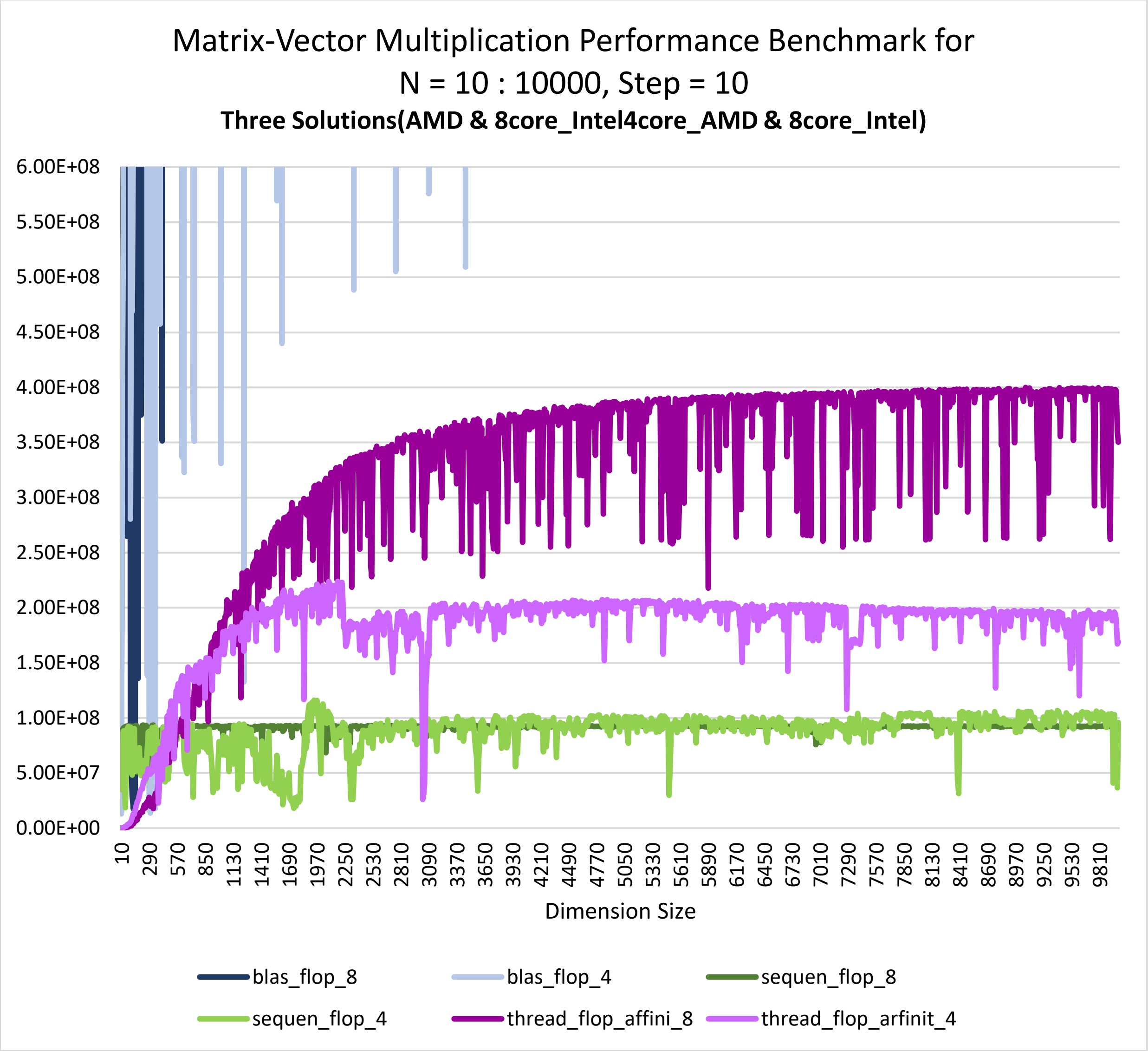


Fig 8, Enlargement Plot of Three Solutions (4core_AMD VS 8core_Intel).

5.1.4. Another computer introduction

- **CPU**

  ------- AMD processor, 2.44GHz

  ------- 8 cores

  ------- 18 logical processors

  ------- L1 cache 512kb

  ------- L2 cache 4.0 MB

  ------- L3 cache 32.0MB

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

5.1.5. Analysis

From Fig7 and Fig8, it is easy to observed that BLAS solution and Thread Solution's FLOPs increase a lot in CPU with 8 cores, and the Sequential Solution in CPU with 8 cores almost has no change (two green lines are almost overlap).

It is mainly because BLAS and my Thread Solution adopt Parallel Optimization.

Parallel Execution in BLAS: BLAS libraries, including OpenBLAS, are optimized to leverage parallelism. The library can distribute the workload across available CPU cores, allowing multiple calculations to be performed simultaneously. This parallelism is particularly beneficial for matrix-vector multiplication, where the workload can be divided into independent tasks.

Thread Solution and Affinity: In my Thread Solution, it implements multi-threading with affinization. Each thread is assigned to a specific core using SetThreadAffinityMask. When running on a CPU with 8 cores, this means that each of the 8 threads can be assigned to a separate core. As a result, the threads can execute in parallel, contributing to the observed increase in FLOPs. But in my 4-core computer, 2 threads have to run on one core, they will compete resource. So, the 8-core have significant performance improvement.

Sequential Solution: The Sequential Solution, as its name suggests, is designed to execute sequentially. When running on a CPU with 8 cores, it does not take advantage of the additional cores for parallel execution. The lack of change in FLOPs for the Sequential Solution on a CPU with 8 cores indicates that it is not benefiting from the parallelism offered by the extra cores. So, the two green lines almost overlap.