Comparing different approachs for matrix multiplication omptimization

Eduardo López Fortes

December 1, 2024

Github repo: Third-MatrixMultiplication

Abstract

Throughout this memory, some algorithms for matrix multiplication algorithms will be discussed. To implement these algorithms, we will use the Java programming language because it is not only widely used for software development but also because the previous stage showed us that it is the most efficient language for this problem, at least for middle-sized matrices, i.e., matrices with dimensions up to 1024x1024. The algorithms that will be discussed are the following: the use of threads to improve the performance of the operation by adding parallelization to the implementation (the number of threads will be both prefixed and nonprefixed), the use of Java Streams alongside with the parallel method and finally a vectorization approach. The goal of this memory is to analyze the performance of these algorithms and to determine which one is the most efficient for different matrix sizes. We should expect the fixed thread implemention to be the best one but we will find out that the Java Streams approach will turn out to be the best one. Once all the algorithms are implemented, they will be tested with matrices of different sizes and the results will be analyzed using Java Microbenchmark Harness (JMH), which is a Java library that allows us to measure the performance of the algorithms. Finally, the results will be discussed, and the most efficient algorithm for each matrix size will be determined by analyzing the performance of the algorithms using graphs and tables.

1 Introduction

In the previous stage, we implemented some algortithms in order to improve the performance of the naïve algorithms. There were discussed several algorithms such as the Strassen algorithm, block matrix multiplication, and the column-major order algorithm. It was also discussed the performance of these algorithms for different matrix sizes and block sizes as well as how to optimize the process of matrix multiplication for sparse matrices, i.e. when there are many zero elements in the matrices so that the waste of time and resources is minimized by not performing operations with zero elements and storing only the non-zero elements in a more efficient way.

For non-sparse matrices, the best option was the block matrix multiplication algorithm, especially for larger matrix sizes and having a block size of 32. Therefore, in this stage it will be tried to improve the performance of the block matrix multiplication by changing the approach of to the problem of the matrix multiplication: instead of trying to change the way in which the matrix multiplication is done (as it was done in the previous stage by addressing the problem of optimizing the algorithm itself), in this stage it will be tried to divide the problem into smaller subproblems and solve them in parallel, taking advantage of the parallelism of the hardware.

This strategy is known as parallelism and it is a very common strategy in computer science to improve the performance of algorithms. It is based on the divide and conquer strategy, which consists of dividing the problem into smaller subproblems and solving them independently, and then combining the results of the subproblems to obtain the final result. This strategy is very useful when the problem can be divided into smaller subproblems that can be solved independently and in parallel, and when the combination of the results of the subproblems can be done efficiently.

The reason why this strategy is useful is because it takes advantage of the parallelism of the hardware, i.e. the ability of the hardware to perform multiple operations simultaneously. This is especially useful in modern computers, which have multiple cores and multiple threads, which can be

used to perform multiple operations simultaneously. In my case I have a computer with 8 cores, so the maximum number of threads is 2 times the number of cores, i.e. 16 threads. Therefore, I can take advantage of this parallelism to improve the performance of the block matrix multiplication algorithm.

An additional approach to improve the performance of the block matrix multiplication algorithm is to use vectorization, which consists of using the SIMD (Single Instruction, Multiple Data) instructions of the processor to perform multiple operations simultaneously. This is especially useful when the operations can be done in parallel and when the data can be processed in parallel. In the case of matrix multiplication, the operations are independent and can be done in parallel, and the data can be processed in parallel, so vectorization can be used to improve the performance of the block matrix multiplication algorithm.

In this stage, a parallel version of the block matrix multiplication algorithm will be implemented using Java threads and the Fork/Join framework, which is a framework for parallel programming in Java that allows to divide the problem into smaller subproblems and solve them in parallel. It will also be implemented a vectorized version of the block matrix multiplication algorithm using the SIMD instructions of the processor to perform multiple operations simultaneously. The performance of the parallel and vectorized versions of the block matrix multiplication algorithm will be compared with the performance of the sequential version of the block matrix multiplication algorithm and the naïve algorithm for different matrix sizes and block sizes.

Finally, another solution using Java streams will be implemented to compare the performance of the parallel and vectorized versions of the block matrix multiplication algorithm with the performance of the solution using java streams. Java streams is a new feature of Java 8 that allows to perform operations on collections in a functional way, i.e. by applying functions to the elements of the collection. Java streams is a very powerful feature that allows to perform operations on collections in a very efficient way, and it can be used to improve the performance of the block matrix multiplication algorithm.

Java Streams are widely used in Java programming for everyday tasks, not just for matrix multiplication (obviously). They are used to optimize the processing of collections, to filter, map, reduce, and sort elements in a collection, and to perform other operations on collections in a functional way. Java Streams are very powerful and can be used to improve the performance of many algorithms, not just matrix multiplication, but we will take advantage of them to improve the performance of the block matrix multiplication algorithm.

Some issues regarding the implementation of the parallel and vectorized versions of the block matrix multiplication algorithm will be discussed, such as the synchronization of the threads, where the usage of locks, atomic variables, and barriers will be discussed, and the vectorization of the algorithm, where the usage of the SIMD instructions of the processor will be discussed.

The size of the matrices will be up to 1048x1048 unless testing larger matrices will not produce memory errors nor take too long to compute. In order to compare these results with the previous stage, the block size will be 32.

2 Problem Statement

The objective of this stage is to implement a parallel version of the block matrix multiplication algorithm using Java threads and the Fork/Join framework, a vectorized version of the block matrix multiplication algorithm using the SIMD instructions of the processor, and a solution using Java Streams to compare the performance of the parallel and vectorized versions of the block matrix multiplication algorithm with the performance of the solution using Java Streams.

The problem remains the same as in previous stages: multiply two matrices A and B of size N x N and store the result in a matrix C of size N x N. The matrices A and B are randomly generated and the matrix C is initialized to zero. The block matrix multiplication algorithm divides the matrices A and B into blocks of size B x B and multiplies the blocks in parallel using Java threads and the Fork/Join framework. The vectorized version of the block matrix multiplication algorithm uses the SIMD instructions of the processor to perform multiple operations simultaneously. The solution using Java Streams uses the parallel method to perform the matrix multiplication in parallel.

The performance of the parallel and vectorized versions of the block matrix multiplication algorithm will be compared with the performance of the sequential version of the block matrix multiplication algorithm and the naïve algorithm for different matrix sizes and block sizes. The performance of the

solution using Java Streams will also be compared with the performance of the parallel and vectorized versions of the block matrix multiplication algorithm.

This time, we may face a new challenge: the synchronization of the threads. When we use threads to perform operations in parallel, we need to synchronize the threads to ensure that the operations are done correctly and that the results are combined correctly. This can be done using locks, atomic variables, and barriers, which are synchronization mechanisms that allow to control the access to shared resources and ensure that the operations are done correctly.

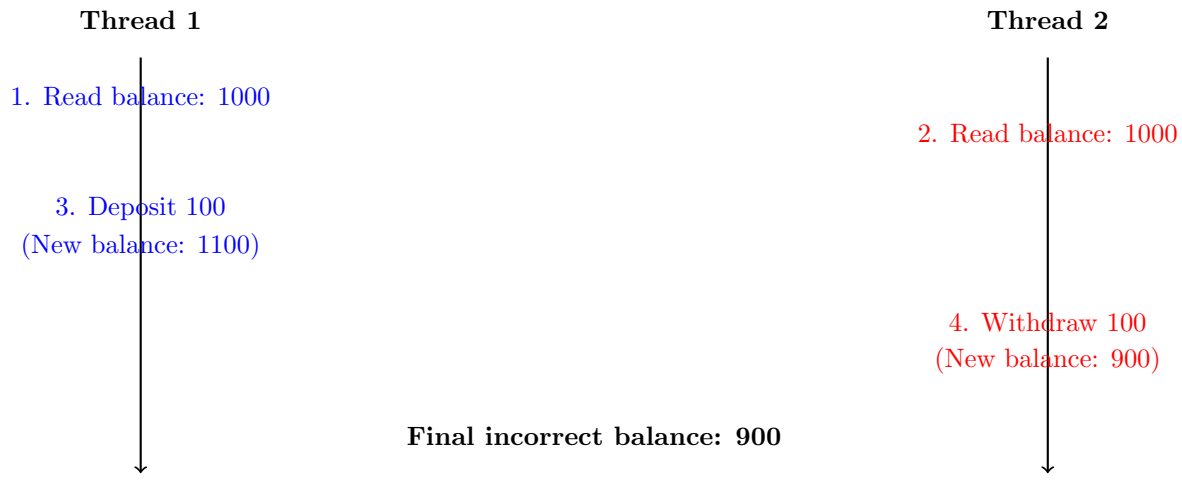
However, in our case we will not need to synchronize the threads because the operations are independent and can be done in parallel. The algorithms are designed in a way so that the results of the operations are stored in different areas of the matrix C, so there is no need to synchronize the threads. That means that only one thread will access a particular position of the matrix C throughout the entire execution of the program, so there is no need to synchronize the threads.

In case it is needed to synchronize the threads, the simplest way to do it is to use locks, which are synchronization mechanisms that allow to control the access to shared resources and ensure that the operations are done correctly. The different threads could first do their operations and when it is time to store the result in the matrix C, they could use a lock to ensure that only one thread can access the matrix C at a time. This way, the threads will not interfere with each other and the operations will be done correctly, and the time a thread has the lock is minimized and therefore the waiting time is minimized. In our case, we will not need to use locks because the operations are independent and can be done in parallel, but it is good to know that locks can be used to synchronize the threads if needed.

Another way to synchronize the threads is to use atomic variables, which are synchronization mechanisms that allow to perform atomic operations on shared variables. Atomic variables are very useful when we need to perform operations on shared variables in a thread-safe way, i.e. when we need to ensure that the operations are done correctly and that the results are combined correctly. In java it is as simple as adding the synchronized keyword to the method that modifies the shared variable.

There are other methods such as semaphores, barriers, and monitors, but they go beyond the scope of this project and are not needed for the implementation of the parallel and vectorized versions of the block matrix multiplication algorithm.

The reason why synchronization is quite important when using threads can be evidenciated by the following example: imagine two threads doing operations over the balance of a bank account. If the operations are not synchronized, it is possible that one thread reads the balance of the account, then the other thread reads the balance of the account, then the first thread adds money to the account, and then the second thread adds money to the account, so the final balance of the account is not correct. This is because the operations are not synchronized and the threads interfere with each other, so the results are not combined correctly. This is why synchronization is important when using threads to perform operations in parallel. Let see this in the following example:



- **Step 1 (Thread 1)**: Thread 1 reads the balance, which is 1000.
- **Step 2 (Thread 2)**: At the same time, Thread 2 also reads the balance, still 1000.

- **Step 3 (Thread 1)**: Thread 1 deposits 100, expecting the balance to be 1100.
- **Step 4 (Thread 2)**: Thread 2 withdraws 100, expecting the balance to be 900.

Because the operations overlap and both threads use the same initial value (1000), the final balance after both operations is **900**, which is incorrect. The correct balance should have been 1000 (after a deposit of 100 and a withdrawal of 100).

This issue occurs because both threads are accessing and modifying the shared balance at the same time without any coordination. To prevent this, synchronization mechanisms are necessary, ensuring that only one thread can read or write to the balance at any given moment.

Ambos hilos realizan operaciones que parecen correctas: depositar 100 y retirar 100 deberían dejar el saldo sin cambios. Sin embargo, cuando estos hilos acceden al balance de la cuenta **al mismo tiempo**, pueden leer y escribir valores incorrectos debido a la falta de coordinación.

Por ejemplo, si el **Hilo 1** está depositando 100 al mismo tiempo que el **Hilo 2** está retirando 100, ambos pueden basarse en un valor incorrecto del balance y, al final, el saldo puede ser diferente al esperado. Esta es la razón por la cual el resultado final del balance puede ser incorrecto e impredecible.

Para evitar estos problemas de concurrencia, se deben usar mecanismos de **sincronización**, que garanticen que solo un hilo pueda acceder a la cuenta bancaria en un momento dado. Esto asegura que las operaciones se realicen secuencialmente y no simultáneamente, evitando interferencias.

3 Methodology

3.1 Baseline

The first step is to implement the algorithms in Java. The implementation of the algorithms will be done using the Java programming language. The algorithms will be implemented as methods that receive two matrices as input and return the result of the multiplication. The matrices will be represented as two-dimensional arrays of integers. Each implementation has been developed in a separate project, so that the code is more organized and easier to understand. This approach also helps to avoid conflicts between the different implementations and makes it easier to test them since each implementation is independent of the others and some of them may require different parameters.

3.1.1 Matrix Generation

Before implementing the algorithms, the method in which the matrices will be generated must be defined. The matrices will be generated randomly, and the elements will be double values between 0 and 1. The size of the matrices will be defined as a parameter, so that the algorithms can be tested with different matrix sizes. The matrices will be generated using the Random class from the java.util package. The matrices will be generated in the main method of the project, and then they will be passed as input to the algorithms. The pseudo-code of the matrix generation is as follows:

Algorithm 1 Matrix Generation

```
1: Input: n                                              ▷ Size of the matrices
2: Initialize matrices a[n][n] and b[n][n]
3: Create random object random
4: for i ← 0 to n − 1 do
5:     for j ← 0 to n − 1 do
6:         a[i][j] ← random.nextDouble()
7:         b[i][j] ← random.nextDouble()
8:     end for
9: end for
```

Now, how all the algorithms work is going to be explained.

Algorithm 2 Matrix Multiplication with Fixed Threads

```
Initialize matrices a, b and c
numThreads ← 16
Create thread pool with numThreads
for each row i in matrix a do
    Submit task to thread pool to multiply row i
end for
Shutdown thread pool
Wait for all tasks to complete or timeout
Output: Matrix c
```

3.2 Parallelism

3.2.1 Fixed Number of Threads

The method for multiplying rows will be the same for all the algorithms unless the contrary is stated. It is as simple as:

Algorithm 3 Row Multiplication

```
procedure multiplyRow(row)
    for each column j in matrix b do
        for each element k in row do
            c[row, j] ← result[row, j] + a[row, k] × b[k, j]
        end for
    end for
end procedure
```

In this example we work with a fixed number of threads, i.e the number of threads is defined before the execution of the program and is independent of the size of the matrix. This is useful when we know the number of threads that we want to use and we want to control the number of threads that are created. In this case, we define the number of threads as 16, which is the maximum number of threads that can be created in my computer. It will be shown on the next case how to create a dynamic number of threads and what happens when the number of threads is greater than the number of cores.

Then, we create an ExecutorService with a fixed number of threads, which is a thread pool that allows to submit tasks to the threads in the pool. We submit each row multiplication task to the ExecutorService, which is a task that multiplies a row of the matrix A by the matrix B and stores the result in the matrix C. We then shut down the ExecutorService once all tasks have been submitted and wait for all tasks to complete.

This code is very simple and easy to understand, but it has some limitations. One limitation is that the number of threads is fixed and does not change depending on the size of the matrix. This means that if the number of threads is greater than the number of cores, the performance of the program may be worse than if the number of threads is equal to the number of cores. This is because the operating system has to switch between threads more frequently, which can cause overhead and reduce the performance of the program.

For the purpose of testing the performance of this approach, "1,2,4,8 and 16" threads will be used. However, we can expect performance to be better when the number of threads is equal to the number of cores, unless the number of rows is smaller than the number of threads, in which case the performance will be worse since the overhead of creating and managing threads will be greater than the benefit of parallelism and there will be idle threads, which will not be used because there are not enough rows to process.

3.2.2 Dynamic Number of threads

In this case the number of threads is not fixed and is equal to the number of rows of the matrix. This is useful when we want to create a thread for each row of the matrix and we want to take advantage of the parallelism of the hardware. In this case, we create a thread for each row of the matrix and launch

Algorithm 4 Matrix Multiplication using Threads

```
Initialize matrices a, b and c
Create an array of threads with size size
for each row i from 0 to size − 1 do
    Create a new thread for multiplying row i
    Start the thread
end for
for each thread i from 0 to size − 1 do
    Wait for thread i to finish
end for
Output: Matrix c
```

the threads in parallel. We then wait for all threads to finish before continuing with the execution of the program.

Here the ExecutorService is not used, instead we create an array of threads and launch the threads in parallel. We then wait for all threads to finish before continuing.

This code is also simple and easy to understand, but it has some limitations. One limitation is that if the matrix size is very large, the number of threads may be greater than the number of cores, which can reduce the performance of the program. This is because the operating system has to switch between threads more frequently, which can cause overhead and reduce the performance of the program.

3.2.3 Java Streams

Algorithm 5 Matrix Multiplication using Parallel Streams

```
Initialize matrices a, b and c
startTime ← current time
Use parallel stream to iterate over rows 0 to size − 1
for each row i in parallel do
    for each column j in matrix b do
        for each element k in row do
            c[i, j] ← result[i, j] + a[i, k] × b[k, j]
        end for
    end for
end for
Output: Matrix c
```

In this case, Java Streams are used to perform the matrix multiplication in parallel. Java Streams are a new feature of Java 8 that allows to perform operations on collections in a functional way, i.e. by applying functions to the elements of the collection. Java Streams are very powerful and can be used to improve the performance of many algorithms, not just matrix multiplication.

The implementation is very simple and easy to understand. We use the IntStream.range method to create a stream of integers from 0 to the size of the matrix (number of rows). We then call the parallel method to create a parallel stream, which allows to perform the operations in parallel. We then call the forEach method to apply a function to each element of the stream, which multiplies a row of the matrix A by the matrix B and stores the result in the matrix C.

We are therefore taking advantage of the parallelism of the hardware to improve the performance of the matrix multiplication algorithm by using Java Streams in conjunction with the parallel method in order to perform the operations in parallel and improve the performance of the algorithm.

3.3 Vectorization

The vectorized version of the block matrix multiplication algorithm is especially useful when the operations can be done in parallel and when the data can be processed in parallel. In the case of matrix multiplication, the operations are independent and can be done in parallel, and the data can

be processed in parallel, so vectorization can be used to improve the performance of the block matrix multiplication algorithm.

Algorithm 6 Matrix Multiplication with Dot Product

```
Initialize matrices a and b with random values
Initialize matrix c of size n × n
for each row i of a do
    for each column j of b do
        c[i][j] ← vectorizedDotProduct(a[i], getColumn(b, j))
    end for
end for
Output: Resulting matrix c
```

Algorithm 7 Vectorized Dot Product

```
Input: Row vector row from matrix a, column vector column from matrix b
Output: Dot product result
sum ← 0
for each element k in row and column do
    sum ← sum + row[k] × column[k]
end for
return sum
```

Algorithm 8 Extract Column from Matrix

```
Input: Matrix matrix, column index colIndex
Output: Column vector of matrix
Initialize column array column of size matrix.length
for each row i in matrix do
    column[i] ← matrix[i][colIndex]
end for
return column
```

The previous code just implements the vectorized version of the block matrix multiplication algorithm, but without using the SIMD instructions of the processor.

The next implementation was an attemtp to use the SIMD instructions of the processor to perform multiple operations simultaneously. This is done by using the intrinsics of the Java Virtual Machine (JVM) to access the SIMD instructions of the processor by using the Vector API, which is a new feature of Java 16 that allows to perform vectorized operations on arrays in a very efficient way. The Vector API provides a set of classes and methods that allow to perform vectorized operations on arrays in a very efficient way, and it can be used to improve the performance of the block matrix multiplication algorithm by using the SIMD instructions of the processor.

However, I could not manage to make the code work since I would get the following error: "(package jdk.incubator.vector is declared in module jdk.incubator.vector, which is not in the module graph)". This is because the Vector API is an incubator module and is not part of the standard Java API, so it is not included in the module graph by default. In order to use the Vector API, it is necessary to add the following VM option to the JVM: "–add-modules jdk.incubator.vector". This option adds the Vector API to the module graph and allows to use the Vector API in the program. I added this option to both the VM options and the pom.xml file, but it did not work. The tests were done with the previous code and that is why the results are very poor for this method.

3.4 Test Developing

Having all the algorithms implemented, the next step is to test them with matrices of different sizes and analyze the performance of the algorithms. The performance of the algorithms will be analyzed by measuring the execution time for different matrix sizes. The execution time will be measured

Algorithm 9 Matrix Multiplication with Vectorization

Input: Two matrices a and b of size $n \times n$
Output: Resulting matrix c
Initialize matrices a and b with random values
Initialize matrix c of size $n \times n$
for each row i of a **do**
    for each column j of b **do**
        Extract column j from matrix b into array $columnB$
        $c[i][j] \leftarrow$ vectorizedDotProduct($a[i]$, $columnB$)
    end for
end for

Algorithm 10 Vectorized Dot Product

Input: Row vector row, column vector $column$
Output: Dot product result
Initialize sum vector $sumVector$ to zero
$length \leftarrow$ size of the vector species (e.g., 256 bits)
$i \leftarrow 0$
while $i < row.length - length$ **do**
    Load a vector from row starting at position i
    Load a vector from $column$ starting at position i
    $sumVector \leftarrow$ Fused Multiply-Add of row and $column$ vectors
    Increment i by $length$
end while
Reduce $sumVector$ by adding its elements together
for remaining elements in row and $column$ **do**
    Add scalar product of $row[i] \times column[i]$ to sum
end for
return sum

using Java Microbenchmark Harness (JMH), which is a Java library that allows us to measure the performance of the algorithms. The results will be analyzed by comparing the execution times of the algorithms for different matrix sizes. The goal is to determine which algorithm is the most efficient for different matrix sizes. The results will be discussed, and the most efficient algorithm for each matrix size will be determined by analyzing the performance of the algorithms using graphs and tables.

The experiment is more precise if we consider the process of creation is inherited from the matrix multiplication. As a result, when benchmarking, it was not treated as setup but as part of the process itself. The process of vectorizing the matrices will be considered as well as part of the process of matrix multiplication.

For this experiment we tested different sizes of matrix: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 (2048 and 4096 were removed from the test set as it takes to much time to compute them).

The multi-threading matrix multiplication needs to be tested with different number of threads to see if the performance is better when the number of threads is equal to the number of cores, and if the performance is worse when the number of threads is greater than the number of cores. The number of threads will be tested with 1, 2, 4, 8, 16 and 32 threads, besides the number of cores in my computer is 8.

For every size the performance would be sized by measuring the time it took to the program to be executed (including the process of creating the matrix) in milliseconds.

In addition, in order to have consistent and reliable results, the following procedure was followed:

1. An arbitrary number of warm-ups rounds would be set to run before testing the program
2. The test would be ran several times (5)
3. This two steps would form a complete round. This process would be completed 3 times before moving to testing the next size (or number of threads if applicable)

3.5 Software used

Intellij Idea along with a maven project were chosen for running and editing Java code, and then Java Microbenchmarking Harness for the testing purposes - added as a dependency in the pom.xml file:

```
- Java 11
- Java Microbenchmarking Harness 1.35
```

3.6 Machine Specifications

Hardware specifications influences notably on the testing results since many factors could change the way in which the program is executed: amount of RAM, type of processor - including number of cores -, graphic card (if applicable), etc.

For that reason, the specifications of the machine in which the test were ran are the following:

```
- Machine: AMD64
- Processor: AMD Ryzen 7 5700U with 8 cores, 1.80 GHz
- RAM: 8GB
- SSD storage: 512GB
- Operating System: Windows 10 Home
- Version: 22H2
```

4 Experiments

4.1 Fixed number of threads

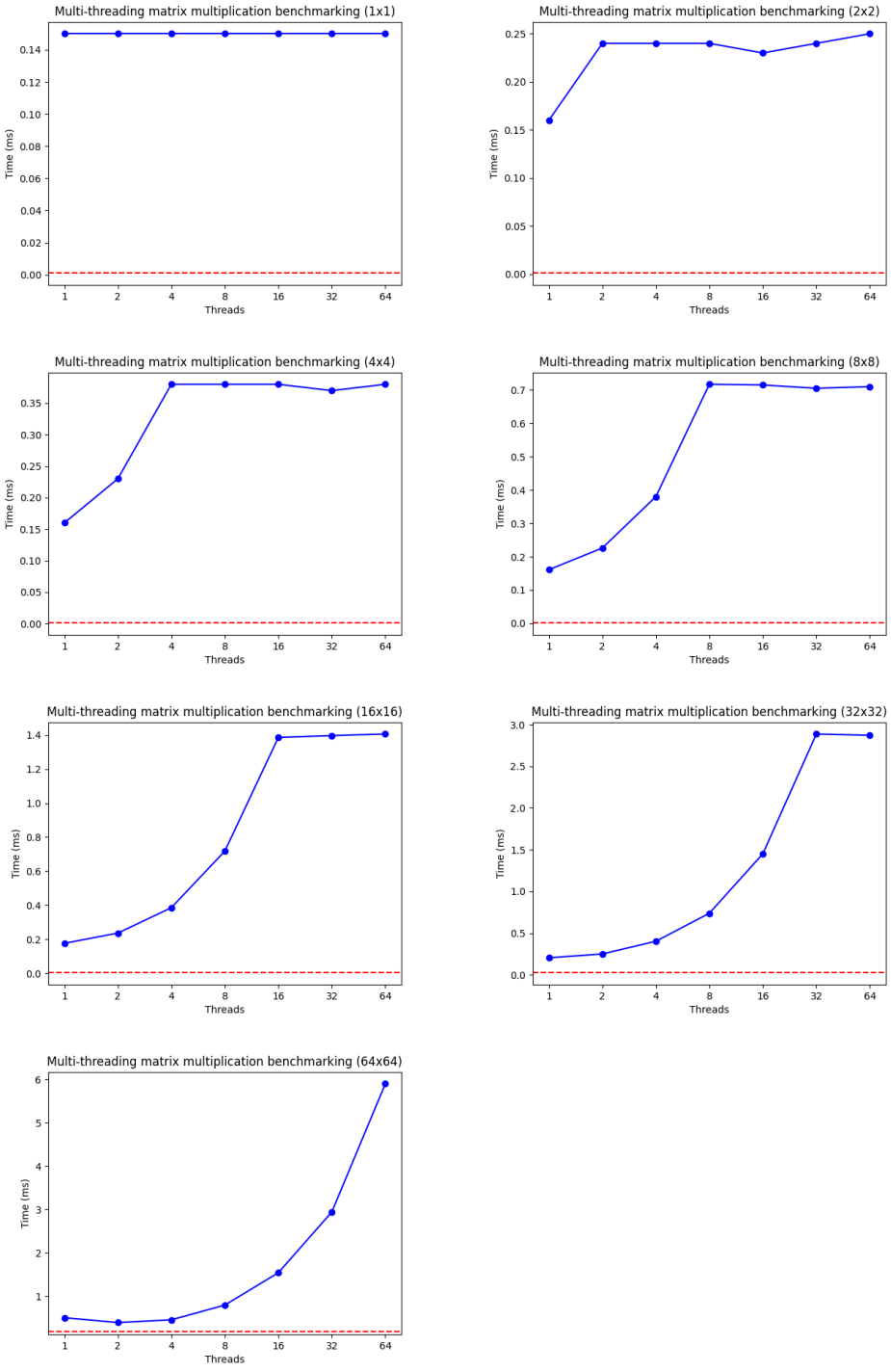


Figure 1: Time of execution for different number of threads for matrix multiplication (sizes 1,2,4,8,16,32 and 64)

We can appreciate in Figure 1 that for matrix sizes up to 64, using threads does not improve the performance of the naïve solution. This is because the overhead of creating and managing threads is greater than the benefit of parallelism. In addition, the optimal number of threads is 1, which is the

equivalent of the naïve solution.

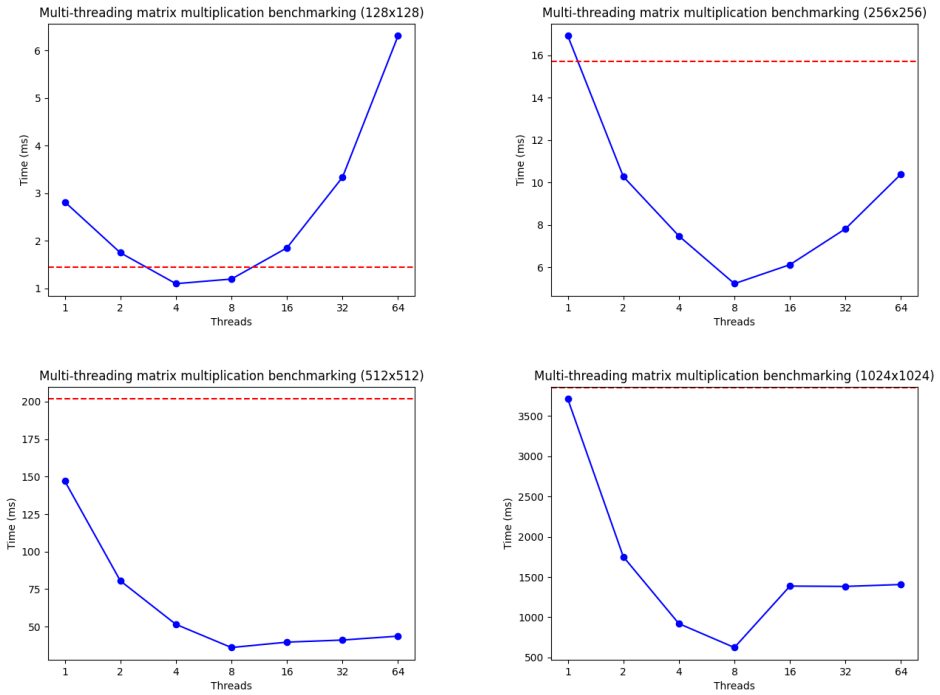


Figure 2: Time of execution for different number of threads for matrix multiplication (sizes 128,256,512 and 1024)

However, figure 2 shows us that for matrix sizes over 128 (included) the performance of the naïve solution is improved by using threads. This is because the overhead of creating and managing threads is less than the benefit of parallelism. In addition, the optimal number of threads is 8, which is the equivalent of the number of cores in the machine. Lowering the number of threads to 4 or 2 will decrease the performance of the solution since the number of threads is less than the number of cores in the machine so we are not taking full advantage of the parallelism; and if the number of threads is increased to 16 or 32, the performance of the solution will decrease since the overhead of creating and managing threads is greater than the benefit of parallelism.

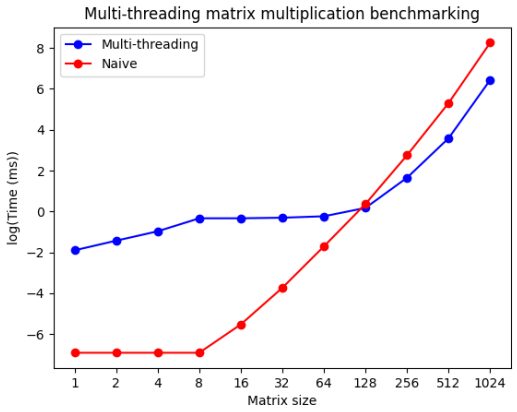


Figure 3: Comparison with the naïve implementation

Therefore, Figure 3 exposes that in general terms, the performance of the naïve solution is improved by using threads for matrix sizes over 128 (included) and the optimal number of threads is 8, which

is the equivalent of the number of cores in the machine. For matrix sizes up to 64, using threads does not improve the performance of the naïve solution and is not worth it.

4.2 Dynamic Number of Threads

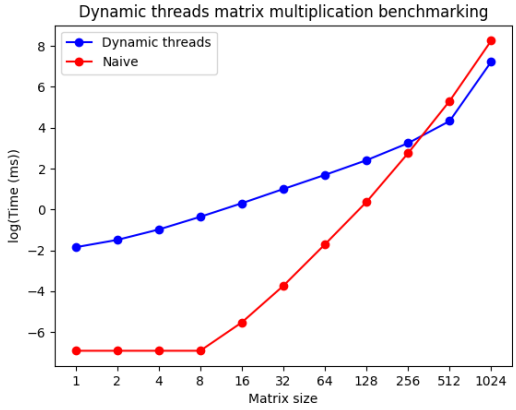


Figure 4: Comparison with the naïve implementation

If the reader take a look on Figure 4, the dynamic threads was the implementation that created the most threads, which is why it performed that bad. For larger matrices the performance of the dynamic threads implementation is better than the naïve implementation, but the difference is not significant. The dynamic threads implementation is not worth it since the overhead of creating and managing threads is greater than the benefit of parallelism and since the number of threads is extremely high (and far from the optimal number of threads, which is 8) the overhead of creating and managing threads is even greater.

4.3 Java Streams

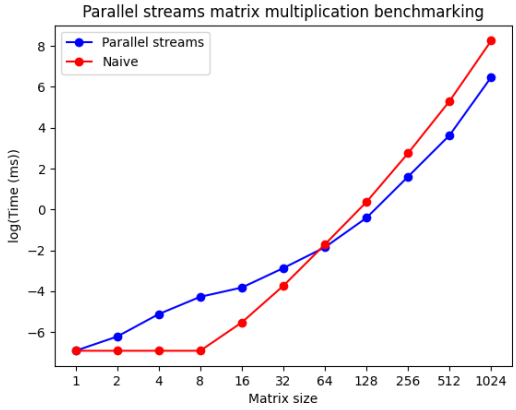


Figure 5: Comparison with the naïve implementation

Figure 5 shows that the Java Streams implementation is faster than the naïve solution, even for mid-sized matrices. For small matrix sizes up to 32, the difference between the two is not significant, but for larger matrix sizes, the Java Streams implementation is significantly faster than the naïve solution. This is because the Java Streams implementation takes advantage of the parallelism of the machine, which the naïve solution does not.

4.4 Vectorization

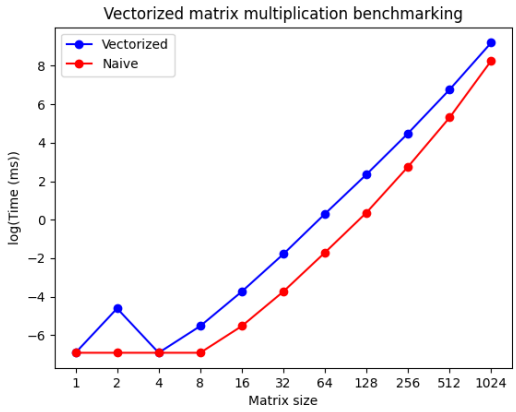


Figure 6: Comparison with the naïve implementation

As it was stated before, the implementation of the vectorized matrix multiplication was incomplete and the results are not accurate. That is why Figure 6 shows that the performance of this implementation is worse than the naïve solution, even for larger matrices where the performance of the naïve solution is improved by using whatever kind of improvement. The implementation using the SIMD would have likely been the fastest implementation, but it was not completed.

5 Conclusions

The vectorized implementation is not represented in Figure 7 since if was worse than the naïve solution for all matrix sizes, so the comparison was made between the naïve solution, the Java Streams implementation, the dynamic threads implementation and the fixed threads implementation. Apart from it, it can be concluded that the best implementation overall was the parallel streams one: it is not only the best in performance but also in easiness of implementation.

For matrix sizes over 256 the results for the optimal thread matrix multiplication (using 8 threads) and the paralell streams multiplication are very similar, but for small matrix sizes, the streams solution is way better. So, if I had to implement one of them for my app and I did not know the sizes of the matrices, I would choose the parallel streams implentation without thinking it: easy to implement and works extremely good for every size of matrix (for matrix sizes between 1-32 in the order of less than one milisecond).

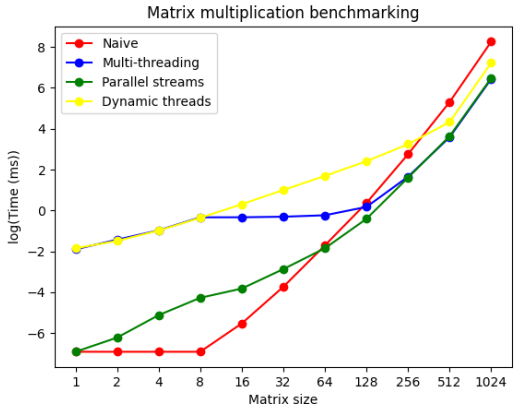


Figure 7: Comparison of all the methods for different matrix sizes. Log scale

Since it would not be a paper made by me without apologizing it the end... I apologize for the vectorized implementation. I took advantage of the code that Marilola kindly provided us and I wanted to go beyond that and use the Java Vector API but after hours of work trying to make it work but in the end if was not possible, so sorry for that.

6 Future Work

In this stage we have implemented an tested several algorithms that could improve the naïve matrix multiplitation algorithm. The streams solution is used in the real world, but threads not that much.

For that reason, this work could be expanded by exploring other options regarding the use of Java Streams suchs as map-reduce.

I will keep trying to make the vectorized implementation work and if it does, I will include it in the final paper.