Parallel Computing - Assignment 1

Martin Štrekelj
Hallak mohamadAnas
Rishabh Gupta

Task 1 - Matrix Multiplication using three loops

1. Overview

In this first task, we have implemented a simple three-loop matrix multiplication along with a basic matrix generator that creates a square matrix of a given size and fills it with random integers between 1 and 100. The generator uses a fixed seed to ensure reproducibility of the generated values.

The matrix multiplication is done using a straightforward, classic triple-nested loop approach. The code iterates over each row i of the first matrix and each column j of the second matrix. For each pair (i, j), it performs a dot product between the i-th row of the first matrix and the j-th column of the second matrix by looping through a shared index k. The result of this multiplication is then accumulated and stored in the corresponding element of the target (result) matrix at position [i][j].

The execution time of the multiplication process is measured using std::chrono, and the results including the original input matrices and the result matrix are saved to CSV files. These files are written to a folder named tests so we can verify that the matrix multiplication has been done correctly using numpy.

The code is a functional starting point for working with matrix operations, and it can serve as a baseline for experimenting with more optimized or parallelized versions in future tasks.

2. Implementation

```
for (int i = 0; i < SIZE_OF_MATRIX; i++)
    {
        for (int j = 0; j < SIZE_OF_MATRIX; j++)
        {
            for (int k = 0; k < SIZE_OF_MATRIX; k++)
            {
                targetMatrix[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }
```

Figure 1 : **IKJ Loop Implementation**

3. Benchmarks

#	Size of Matrix	Avg. Kernel Execution Time (ms)
1	128	1
2	256	10
3	512	106
4	1024	960
5	2048	26250
6	3072	79035

We then computed the time complexity, and we followed the log-log analysis approach. We transformed both the input sizes and execution times using $\log_2$ and performed linear regression. The slope of the fitted line represents the exponent b in the empirical time complexity model $T(n) \approx a * n^b$. In this case, the slope was approximately 3.6, indicating that the algorithm's time complexity is $O(n^{3.6})$.

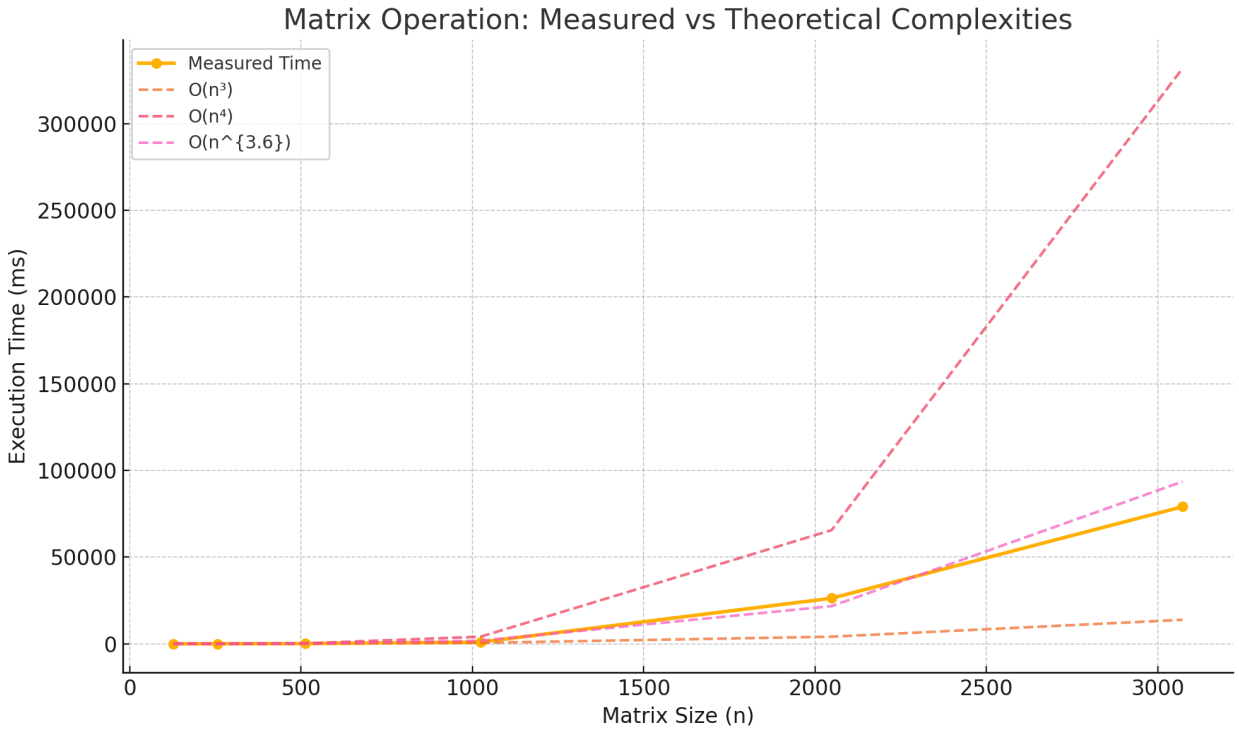

Figure 2: Comparing task1 results with theoretical time complexity

Task 2 - Optimization of Mat-Mul using Transposing and Blocking

1. Overview

For the second task (see Figure 3 for implementation details), we aimed to speed up the matrix multiplication by introducing blocking and a more cache-friendly memory layout. These strategies help reduce cache misses and take better advantage of the CPU's memory hierarchy.

The first key change is storing matrices in a single-dimensional array. Instead of working with a 2D layout, we layout the elements contiguously, which improves data locality. When the CPU fetches a block of memory into its cache, it brings along neighboring elements, so accessing them linearly can dramatically cut down on cache misses.

The second major change is using a recursive divide-and-conquer approach to handle the multiplication in smaller sub-blocks. Once a sub-block's size gets small enough (less than or equal to BLOCK_SIZE), we fall back to a triple-nested loop to finish the multiplication. This blocking ensures that the chunk of data we're multiplying stays in cache for the duration of that local computation, minimizing trips to slower main memory.

Within that triple-nested loop, we also switch from the more common IJK ordering to IKJ ordering. Here's the difference:

IJK:

In this pattern, we update C[i][j] for every k, which can bounce around in memory and cause more cache misses.

IKJ:

By fixing i and k first, we grab a single row element from A (A[i][k]) and reuse it for every j in the same pass, often matching row-major storage more closely. This approach can keep the data you need "hot" in the cache, speeding up the computation.

Altogether, these techniques—blocking, a 1D memory layout, and the IKJ loop ordering—lead to a significant performance boost by making the most of the CPU cache. They also provide a strong foundation for more advanced optimizations in the future, like parallel computation or vectorization.

2. Implementation

```
/**
 * Recursively multiplies two n×n submatrices A and B into submatrix
C.
** A, B, and C each point to the top-left of an n×n region within possibly
 * larger 2D data. offsetA, offsetB, offsetC indicate how many element
s* per row in each matrix (their "leading dimensio
n*/.
void matmulRecHelper(const int *A, const int *B, int *C,
                     int n, int offsetA, int offsetB, int offsetC)
{
    // IKJ version
    if (n ≤ BLOCK_SIZE)
    {
        for (int i = 0; i < n; ++i)
        {
            for (int k = 0; k < n; ++k)
            {
                int r = A[i * offsetA + k];
                for (int j = 0; j < n; ++j)
                {
                    C[i * offsetC + j] += r * B[k * offsetB + j];
                }
            }
        }
    }
    else
    {
        // Divide and conquer: split n×n block into four (n/2)×(n/2) blocks
        int half = n / 2;

        // Offsets in A
        const int *A11 = A;
        const int *A12 = A + half;           // shift right by half
        const int *A21 = A + half * offsetA; // shift down by half
        const int *A22 = A21 + half;         // shift down and righ
                                             t
        // Offsets in B
        const int *B11 = B;
        const int *B12 = B + half;           // shift right by half
        const int *B21 = B + half * offsetB; // shift down by half
        const int *B22 = B21 + half;         // shift down and righ
                                             t
        // Offsets in C
        int *C11 = C;
        int *C12 = C + half;           // shift right
        int *C21 = C + half * offsetC; // shift down
        int *C22 = C21 + half;         // shift down and righ
                                       t
        // C11 = A11*B11 + A12*B21
        matmulRecHelper(A11, B11, C11, half, offsetA, offsetB, offsetC);
        matmulRecHelper(A12, B21, C11, half, offsetA, offsetB, offsetC);

        // C12 = A11*B12 + A12*B22
        matmulRecHelper(A11, B12, C12, half, offsetA, offsetB, offsetC);
        matmulRecHelper(A12, B22, C12, half, offsetA, offsetB, offsetC);

        // C21 = A21*B11 + A22*B21
        matmulRecHelper(A21, B11, C21, half, offsetA, offsetB, offsetC);
        matmulRecHelper(A22, B21, C21, half, offsetA, offsetB, offsetC);

        // C22 = A21*B12 + A22*B22
        matmulRecHelper(A21, B12, C22, half, offsetA, offsetB, offsetC);
        matmulRecHelper(A22, B22, C22, half, offsetA, offsetB, offsetC);
    }
}
```

Figure 3: **IKJ Loop with Recursive Divide-And-Conquer Approach**

3. Benchmarks

#	Size of Matrix	Block Size	Avg. Kernel Execution Time (ms)
1	128	64	0
2	512	64	19
3	1024	64	165
4	2048	64	1376
5	3072	64	5528

We did the same analysis for this part and got that the time complexity is proximity O(n3.15)

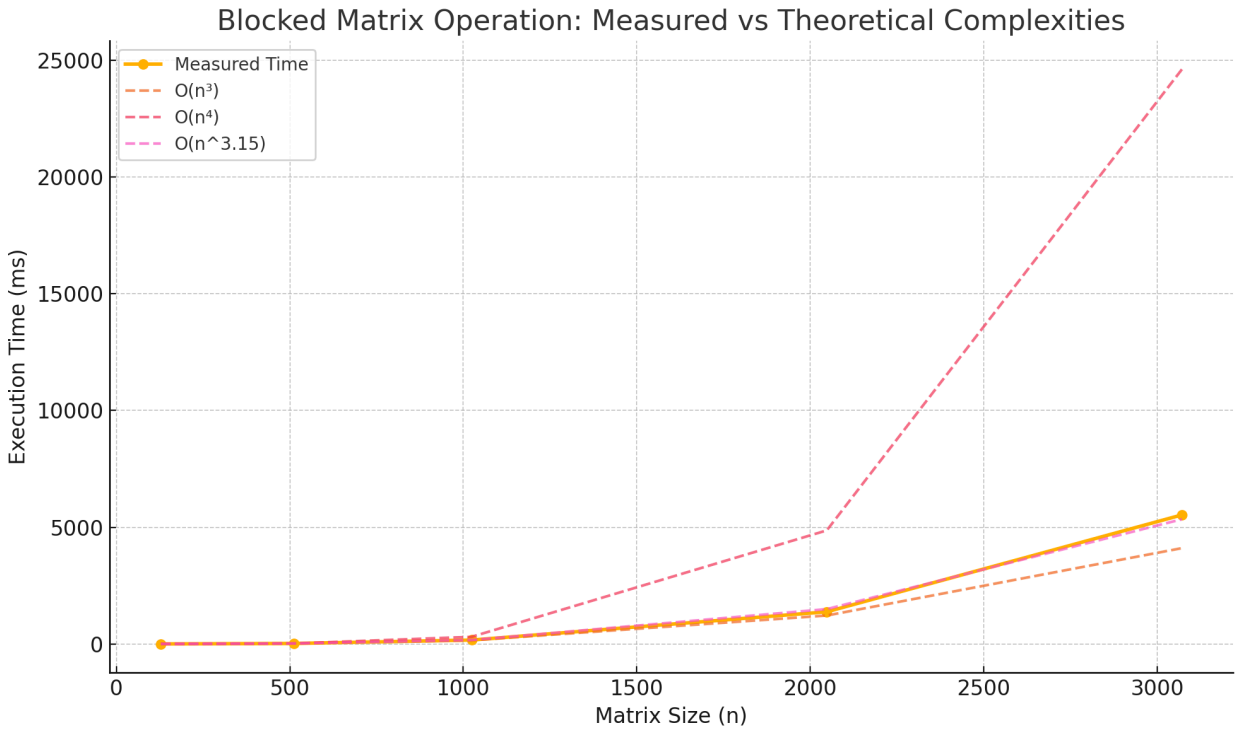

Figure 4: Comparing task 2 results with theoretical time complexity

Task 3 - Parallelization of Mat-Mul using OpenMP

1. Overview

For this third iteration, we extended our block-based, cache-friendly matrix multiplication by adding parallel tasks with OpenMP. We kept the same recursive divide-and-conquer structure but introduced a TASK_THRESHOLD to decide when to spawn parallel tasks. Large submatrices break into smaller blocks that run simultaneously on multiple CPU cores, while smaller submatrices use a direct IKJ triple-nested loop. The IKJ ordering reuses a single row element from the first matrix for all columns in the second matrix, minimizing cache misses.

In our OpenMP implementation (see Figure 6), we rely on #pragma omp task for the larger submatrices, distributing them among available threads. Because each task works on a distinct region of the result matrix, we avoid race conditions—there's no overlap in the sections being written to at the same time. For the final fallback, where blocks are below a certain size, we use #pragma omp parallel for on the innermost loops to further parallelize the multiplications without colliding on shared memory. Altogether, this approach ensures both data safety and maximal CPU utilization.

We also measure execution time using omp_get_wtime(), which gives us a direct look at how effectively we're harnessing multiple threads. By balancing task creation overhead, block sizes, and the recursive structure.

In order to optimize our program, we tested different memory access scheduling strategies for our loop. As displayed in the table below and visible on Figure 5, best performance was achieved by using runtime scheduling and the worst performing was auto.

Size of Matrix	Auto (ms)	Dynamic (ms)	Runtime (ms)	Static (ms)	Guided (ms)
128	0.999928	1.00017	0.999928	2.00009	0.999928
512	23.9999	28.0001	27.0002	30	34.9998
1024	107	63	108	171	104
2048	471	445	413	432	410
3072	2005	1462	1387	1482	1401

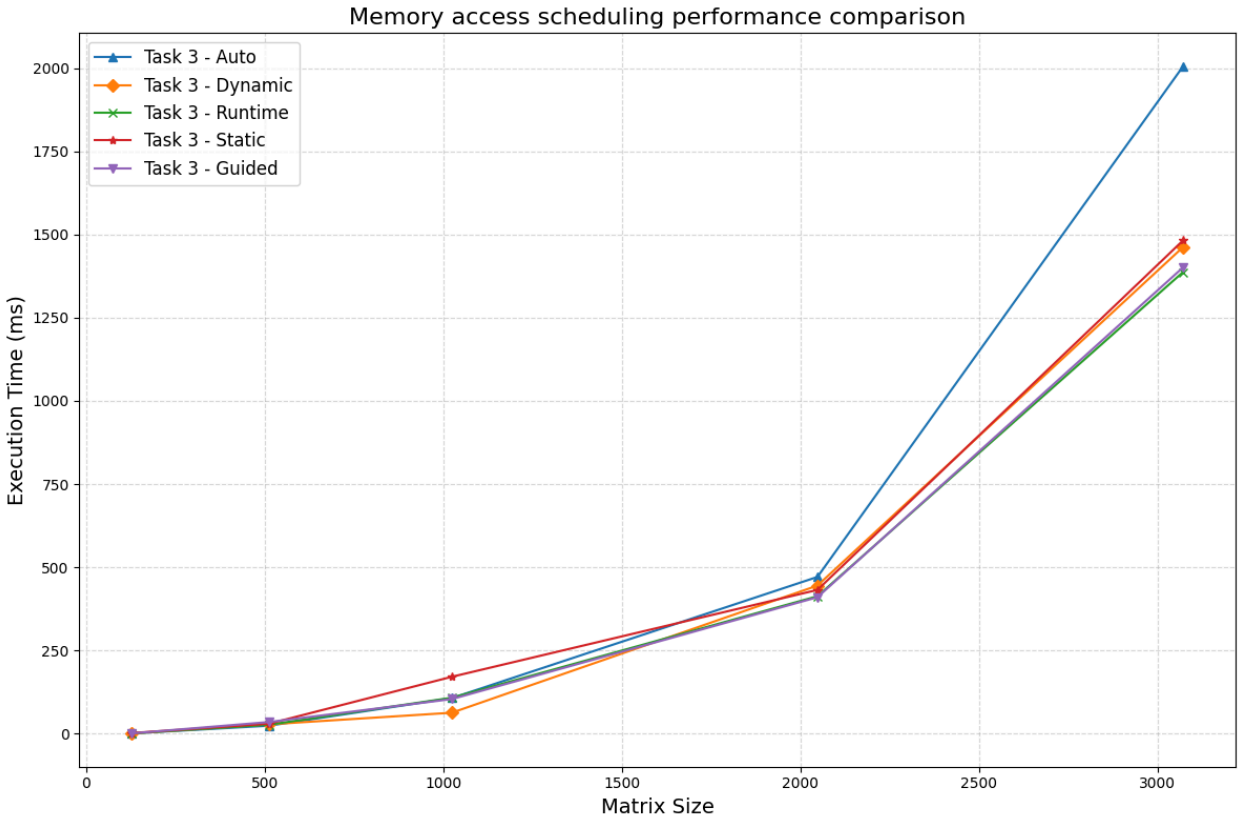

Figure 5: Comparing different memory access scheduling

2. Implementation

```
static const int BLOCK_SIZE = 256;
static const int TASK_THRESHOLD = 512;  // Only spawn tasks for blocks larger than this

void matmulRecHelper(const int* A, const int* B, int* C,
                     int n, int offsetA, int offsetB, int offsetC)
{
    // IKJ version
    if (n ≤ BLOCK_SIZE)
    {
        #pragma omp parallel for schedule(runtime)
        for (int i = 0; i < n; ++i)
        {
            for (int k = 0; k < n; ++k)
            {
                int r = A[i * offsetA + k];
                for (int j = 0; j < n; ++j)
                {
                    C[i * offsetC + j] += r * B[k * offsetB + j];
                }
            }
        }
    }
    else
    {
        // Divide and conquer: split n×n block into four (n/2)×(n/2) blocks
        int half = n / 2;

        // Offsets in A
        const int* A11 = A;
        const int* A12 = A + half;                  // shift right by half
        const int* A21 = A + half * offsetA;        // shift down by half
        const int* A22 = A21 + half;                // shift down and right

        // Offsets in B
        const int* B11 = B;
        const int* B12 = B + half;                  // shift right by half
        const int* B21 = B + half * offsetB;        // shift down by half
        const int* B22 = B21 + half;                // shift down and right

        // Offsets in C
        int* C11 = C;
        int* C12 = C + half;                        // shift right
        int* C21 = C + half * offsetC;              // shift down
        int* C22 = C21 + half;                      // shift down and right

        // Use a taskgroup to synchronize recursive tasks
        #pragma omp taskgroup
        {
            if (n > TASK_THRESHOLD)
            {
                // Spawn tasks for larger subproblems
                #pragma omp task shared(A, B, C) untied
                matmulRecHelper(A11, B11, C11, half, offsetA, offsetB, offsetC);
                #pragma omp task shared(A, B, C) untied
                matmulRecHelper(A12, B21, C11, half, offsetA, offsetB, offsetC);

                #pragma omp task shared(A, B, C) untied
                matmulRecHelper(A11, B12, C12, half, offsetA, offsetB, offsetC);
                #pragma omp task shared(A, B, C) untied
                matmulRecHelper(A12, B22, C12, half, offsetA, offsetB, offsetC);

                #pragma omp task shared(A, B, C) untied
                matmulRecHelper(A21, B11, C21, half, offsetA, offsetB, offsetC);
                #pragma omp task shared(A, B, C) untied
                matmulRecHelper(A22, B21, C21, half, offsetA, offsetB, offsetC);

                #pragma omp task shared(A, B, C) untied
                matmulRecHelper(A21, B12, C22, half, offsetA, offsetB, offsetC);
                #pragma omp task shared(A, B, C) untied
                matmulRecHelper(A22, B22, C22, half, offsetA, offsetB, offsetC);
            }
            else
            {
                // Execute sequentially to avoid task overhead on small subproblems
                matmulRecHelper(A11, B11, C11, half, offsetA, offsetB, offsetC);
                matmulRecHelper(A12, B21, C11, half, offsetA, offsetB, offsetC);

                matmulRecHelper(A11, B12, C12, half, offsetA, offsetB, offsetC);
                matmulRecHelper(A12, B22, C12, half, offsetA, offsetB, offsetC);

                matmulRecHelper(A21, B11, C21, half, offsetA, offsetB, offsetC);
                matmulRecHelper(A22, B21, C21, half, offsetA, offsetB, offsetC);

                matmulRecHelper(A21, B12, C22, half, offsetA, offsetB, offsetC);
                matmulRecHelper(A22, B22, C22, half, offsetA, offsetB, offsetC);
            }
        }
    }
}
```

Figure 6: **Implementation with OMP and multithreading solution**

3. Benchmarks

#	Size of Matrix	Block Size	Avg. Kernel Execution Time (ms)
1	128	256	0.999928
2	512	256	26
3	1024	512	41
4	2048	512	322
5	3072	256	1406

The time complexity for the third task is approximately $O(n^{2.26})$

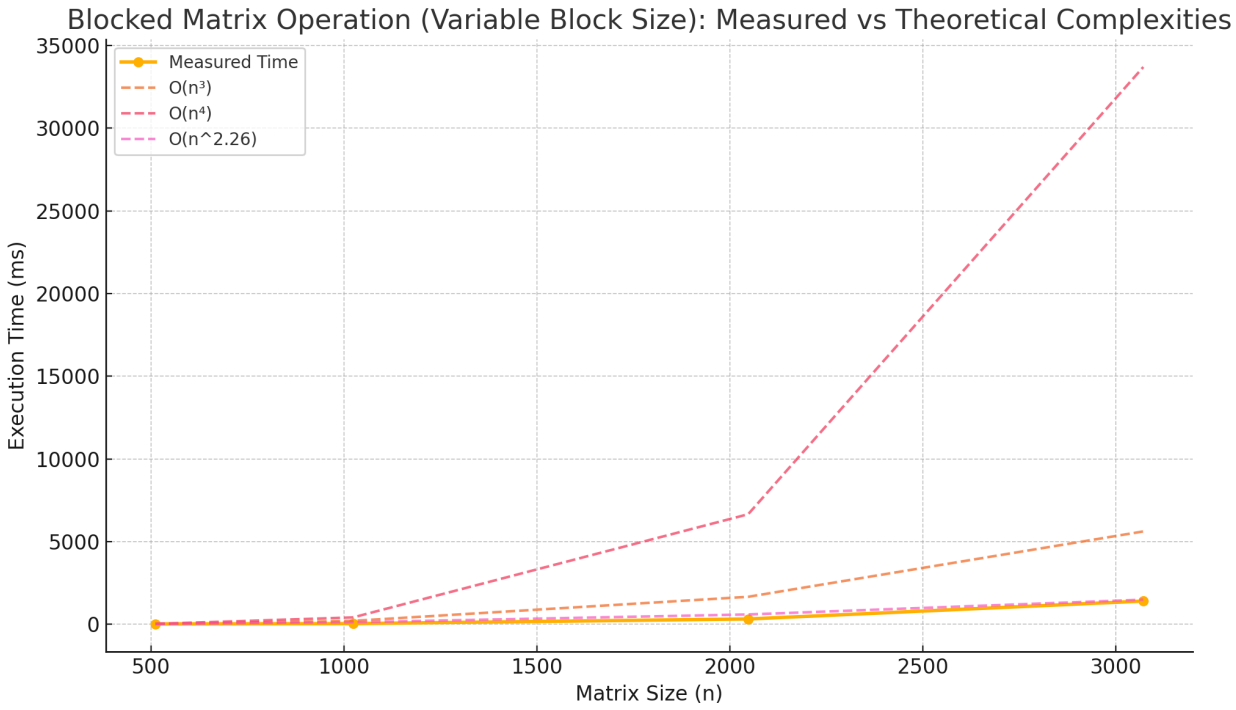

Figure 7: Comparing task 3 results with theoretical time complexity

Comparing execution time between tasks

As we can see on Figure 8, Task 1 (default naive implementation) shows poor scaling, with execution time up to 79,035ms at 3072×3072 size matrix. Task 2's single-threaded optimizations deliver a big (almost 14x) speedup for large matrices through better memory access optimization. Task 3 is built on top of Task 2 optimization and adds multi-threading to get the best results, processing the largest matrices up to 56x faster than baseline and 4x faster than the optimized single-threaded version.

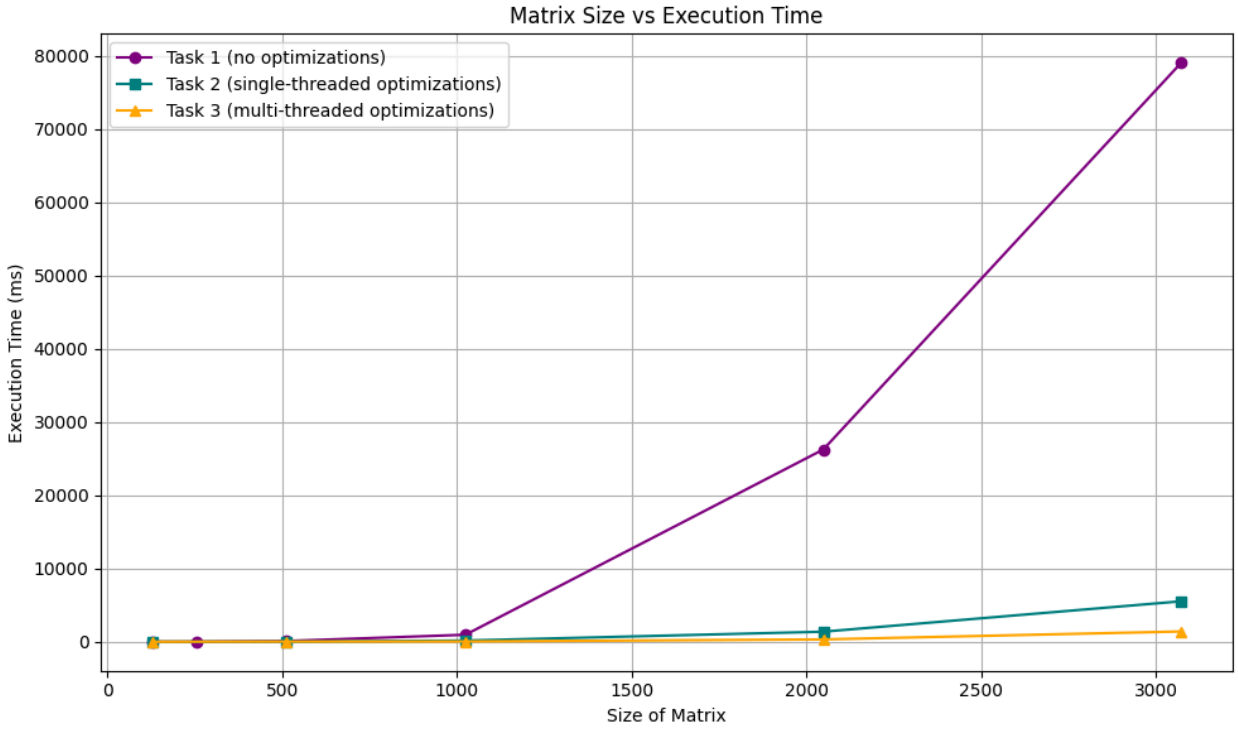

Figure 8: Comparison of execution time between tasks