# Parallel and Vectorized Matrix Multiplication

Individual Assignment 3

Kacper Janiszewski

December 4, 2025

## 1 Introduction

The goal of this assignment is to investigate performance improvements in matrix multiplication using two independent strategies:

- **Parallelisation** using OpenMP multi-threading,
- **Vectorisation (SIMD)** achieved implicitly through data layout optimisation and compiler auto-vectorisation.

All implementations, experiments, and scripts are available in the public repository: `https://github.com/Janisz11/Individual_Assignment_3`

We compare the following approaches:

1. Basic sequential algorithm ($O(n^3)$),
2. Vectorised algorithm (transposed layout of matrix $B$),
3. Parallel algorithm using OpenMP with 1–8 threads.

## 2 Algorithms

### 2.1 Basic Algorithm

The classical triple-nested loop version was implemented as a baseline:

$$C(i,j) = \sum_{k=0}^{n-1} A(i,k) \cdot B(k,j)$$

This implementation has no optimisations and processes elements in row-major order.

### 2.2 Vectorised Algorithm (Optional Part)

Vectorisation was achieved by:

- transposing matrix $B$ into $B^T$,
- ensuring that both input vectors used in the innermost loop are read linearly from memory,
- enabling compiler SIMD through `-O3 -march=native`.

After transposition, the multiplication becomes:

$$C(i,j) = \sum_{k=0}^{n-1} A(i,k) \cdot B^T(j,k)$$

This improves cache locality and allows the compiler to produce AVX/AVX2 instructions.

### 2.3 Parallel Algorithm (OpenMP)

Parallelisation was implemented with a static scheduling policy over the outer loop:

```
#pragma omp parallel for schedule(static) num_threads(t)
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        for (k = 0; k < n; k++)
            C(i,j) += A(i,k) * B(k,j);
```

Speedup and efficiency were calculated as:

$$S = \frac{T_{\text{basic}}}{T_{\text{parallel}}}, \qquad E = \frac{S}{p}$$

where $p$ is the number of threads.

## 3 Experimental Setup

- CPU: multi-core x86_64 (8 logical cores),
- Compiler: g++ with flags `-O3 -march=native -fopenmp`,
- Matrix sizes: $256, 512, 1024$,
- Measurements repeated 3 times, best result used.

The full benchmarking implementation is provided in the file `main.cpp` inside the GitHub repository.

# 4 Results

## 4.1 Execution Times and Speedup

Tables 1, 2, and 3 summarise measured times.

Table 1: Results for size $256 \times 256$

| Variant | Threads | Time [ms] | Speedup | Efficiency |
|---|---|---|---|---|
| Basic | 1 | 22.009 | 1.000 | 1.000 |
| Vectorised | 1 | 6.429 | 3.423 | 3.423 |
| Parallel | 1 | 19.395 | 1.135 | 1.135 |
| Parallel | 2 | 12.596 | 1.747 | 0.874 |
| Parallel | 4 | 8.842 | 2.489 | 0.622 |
| Parallel | 8 | 6.625 | 3.322 | 0.415 |

Table 2: Results for size $512 \times 512$

| Variant | Threads | Time [ms] | Speedup | Efficiency |
|---|---|---|---|---|
| Basic | 1 | 344.962 | 1.000 | 1.000 |
| Vectorised | 1 | 68.652 | 5.025 | 5.025 |
| Parallel | 1 | 338.345 | 1.020 | 1.020 |
| Parallel | 2 | 176.076 | 1.959 | 0.980 |
| Parallel | 4 | 110.931 | 3.110 | 0.777 |
| Parallel | 8 | 70.890 | 4.866 | 0.608 |

Table 3: Results for size $1024 \times 1024$

| Variant | Threads | Time [ms] | Speedup | Efficiency |
|---|---|---|---|---|
| Basic | 1 | 4043.199 | 1.000 | 1.000 |
| Vectorised | 1 | 640.520 | 6.312 | 6.312 |
| Parallel | 1 | 4017.032 | 1.007 | 1.007 |
| Parallel | 2 | 1867.710 | 2.165 | 1.082 |
| Parallel | 4 | 1086.915 | 3.720 | 0.930 |
| Parallel | 8 | 824.128 | 4.906 | 0.613 |

# 5 Plots

To better illustrate the behaviour of the algorithms, the following figures show the speedup of the parallel implementation and the time comparison between the basic, vectorised, and parallel versions.

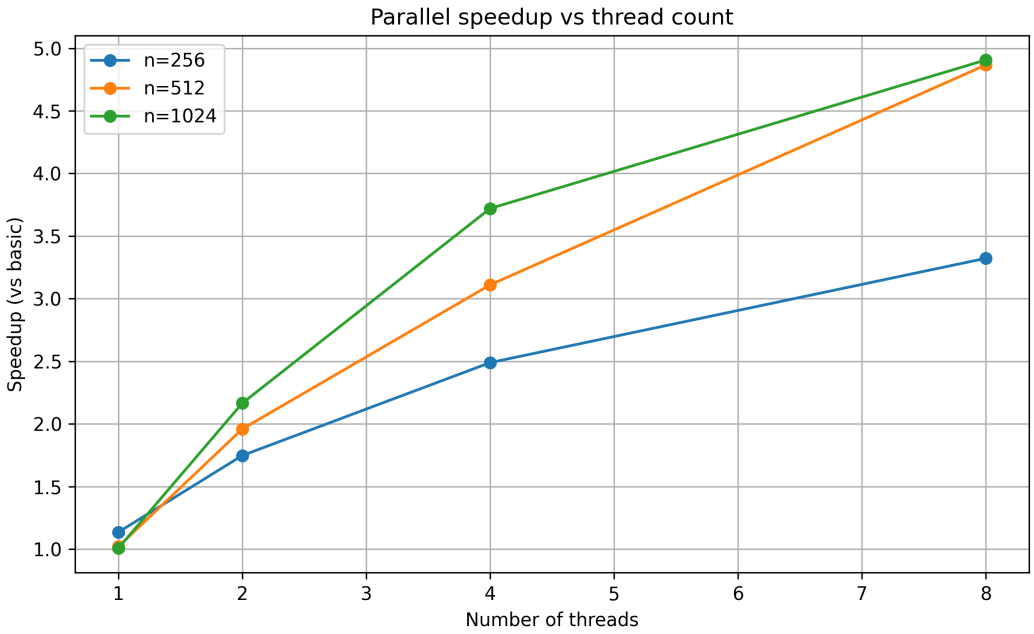


Figure 1: Speedup of the parallel implementation relative to the basic version as a function of the number of threads, for different matrix sizes.

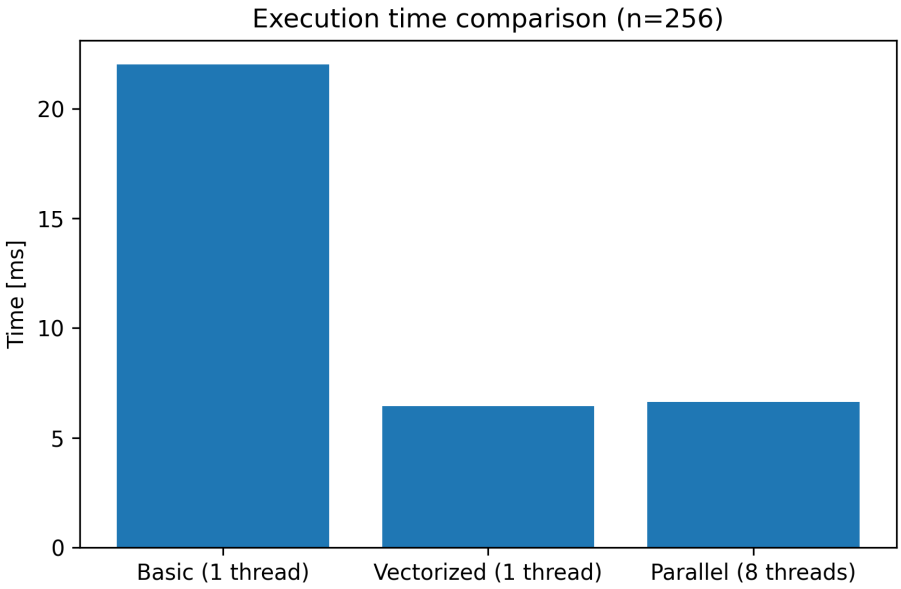


Figure 2: Execution time comparison for matrix size $256 \times 256$: basic, vectorised, and parallel (maximum thread count).

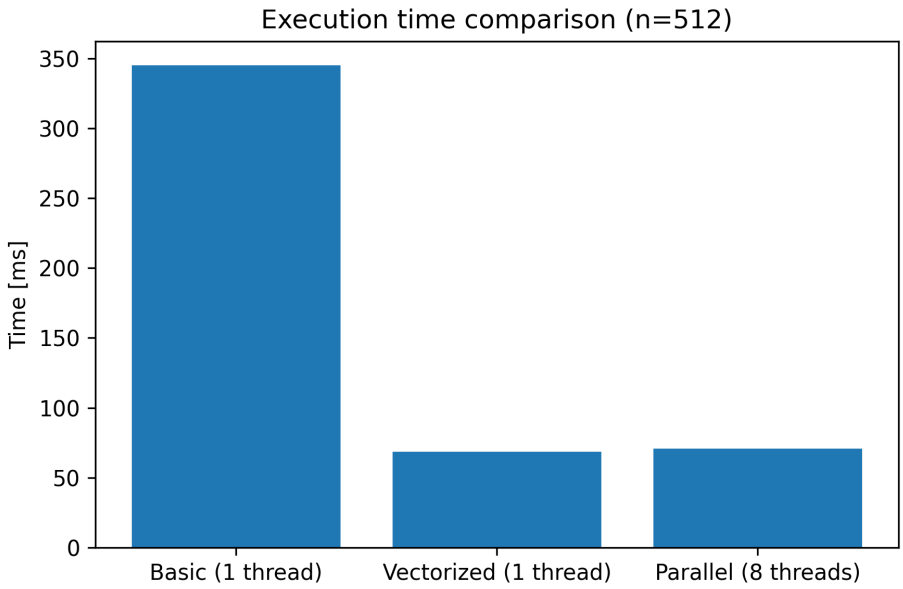


Figure 3: Execution time comparison for matrix size $512 \times 512$: basic, vectorised, and parallel (maximum thread count).

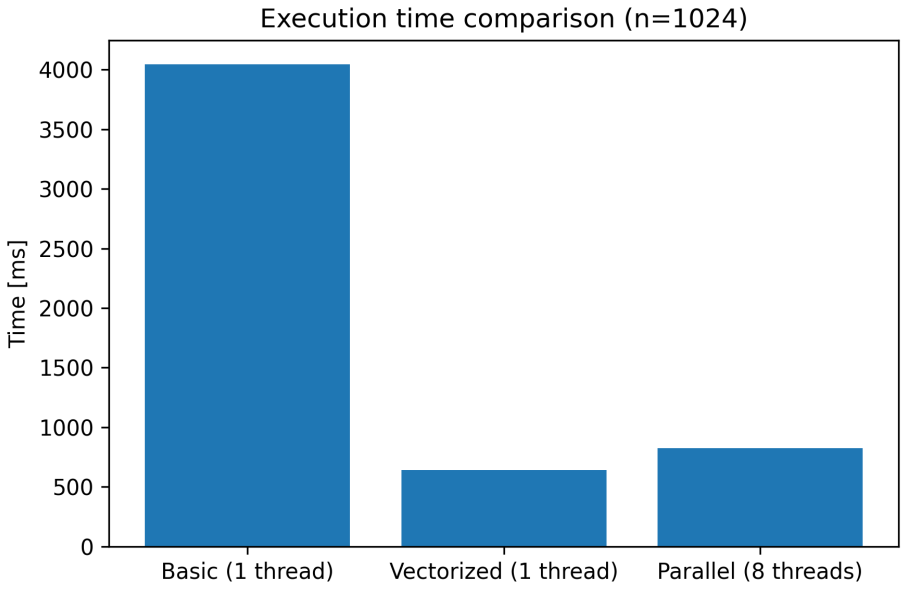


Figure 4: Execution time comparison for matrix size $1024 \times 1024$: basic, vectorised, and parallel (maximum thread count).

# 6 Analysis

## 6.1 Vectorisation Benefits

The vectorised version significantly outperformed the basic algorithm:

- 3.4× faster for size 256,
- 5.0× faster for size 512,
- 6.3× faster for size 1024.

Performance improves with matrix size due to better amortisation of memory access patterns and compiler SIMD utilisation.

## 6.2 Parallelisation Analysis

Parallel speedup scales well:

- up to 4.9× faster with 8 threads for size 1024,
- efficiency close to 1.0 for 2–4 threads,
- expected drop in efficiency for 8 threads ($\approx 0.61$), caused by memory bandwidth limits.

A mild **superlinear speedup** (efficiency $> 1$) appears for 2 threads at size 1024, likely due to improved cache utilisation.

## 6.3 Vectorised vs Parallel

For the largest matrix (1024):

- Vectorised: 640 ms,
- Parallel (8 threads): 824 ms.

Thus the vectorised version is $\approx 1.28\times$ faster than parallel OpenMP with 8 threads. This confirms that data layout and SIMD optimisations can match or exceed multi-threading when memory bandwidth becomes a bottleneck.

# 7 Conclusion

Both optimisation strategies improved performance significantly:

- Vectorisation offered the largest single-thread speedups,
- Parallelisation scaled well up to available CPU cores,

- For large matrices, vectorisation outperformed 8-thread OpenMP,
- Combining both techniques would likely yield the best performance.

The full implementation, datasets, and benchmarking scripts are available at: `https://github.com/Janisz11/Individual_Assignment_3`