Parallel (and Vectorized) Matrix Multiplication

Comparative Study

Alberto Rivero Monzón

December 4, 2025

Repository: `https://github.com/albertuti1910/matrix-multiplication-project`

Contents

1 Introduction

In this assignment, we investigate the performance impact of parallel computing and vectorization on Matrix Multiplication ($C = A \times B$). Matrix multiplication is a computationally intensive operation ($O(N^3)$), making it an ideal candidate for high-performance optimization techniques.

We compare three specific implementations:

1. **Basic**: A standard naive implementation using three nested loops.
2. **Parallel**: An implementation using **OpenMP** to parallelize the outer loop across CPU cores.
3. **Vectorized (Optimized)**: An advanced implementation that combines:
   - **Multi-threading** via OpenMP.
   - **SIMD Intrinsics** (AVX2) to process multiple data points per instruction.
   - **Matrix Transposition** to optimize memory access patterns.

2 Theoretical Background

To understand the results presented in this report, it is necessary to define the optimization techniques used.

2.1 OpenMP and Multi-threading

OpenMP is an API that supports multi-platform shared-memory multiprocessing. It allows us to distribute iterations of a loop across multiple threads. In theory, using T threads should provide a speedup close to $T\times$. However, this is often limited by memory bandwidth and the overhead of creating threads.

2.2 SIMD (Single Instruction, Multiple Data)

Modern CPUs, such as the Intel i7-9750H used in this benchmark, support AVX2 (Advanced Vector Extensions). Standard code processes one number at a time (Scalar). SIMD allows the CPU to load 4 `double` precision numbers (256 bits) into a single register and multiply all of them simultaneously.

2.3 Cache Locality and Transposition

Matrices in C++ are stored in row-major order.

- Accessing $A[i][k]$ is fast because we move linearly through memory.
- Accessing $B[k][j]$ in the inner loop is slow because we jump N memory addresses for every step. This causes **Cache Misses**, where the CPU must wait for data to be fetched from RAM.

By transposing Matrix B (B^T), we convert column access into row access, allowing the CPU to read contiguous memory blocks efficiently.

3 Methodology

The benchmarks were executed on an **Intel Core i7-9750H** (6 cores, 12 threads) running Arch Linux.

- **Compiler Flags**: `-O3 -march=native -fopenmp -mfma` (enables Fused Multiply-Add instructions).
- **Tools**: Linux `perf` was used to measure hardware counters, specifically **L3 Cache Misses** and **IPC** (Instructions Per Cycle).

4 Quantitative Results

This section presents the raw data collected during the benchmarks.

4.1 Execution Time

Table 1 shows the execution time in seconds. While the Basic and Parallel versions grow exponentially with N, the Vectorized version remains highly efficient.

Size (N)	Basic (s)	Parallel (s)	Vectorized (s)
128	0.0037	0.0076	**0.0024**
256	0.0268	0.0082	**0.0041**
512	0.1962	0.0404	**0.0142**
1024	2.4993	0.3238	**0.0715**
2048	50.0731	19.8867	**0.5482**

Table 1: *Execution Time Comparison (seconds)*

4.2 Computational Throughput (GFLOPS)

Table 2 measures the raw computing power utilized. The Vectorized approach consistently utilizes more of the CPU's potential.

Size (N)	Basic	Parallel	Vectorized
128	1.13	0.55	**1.73**
512	1.37	6.64	**18.90**
1024	0.86	6.63	**30.02**
2048	0.34	0.86	**31.34**

Table 2: *Throughput Comparison (GFLOPS)*

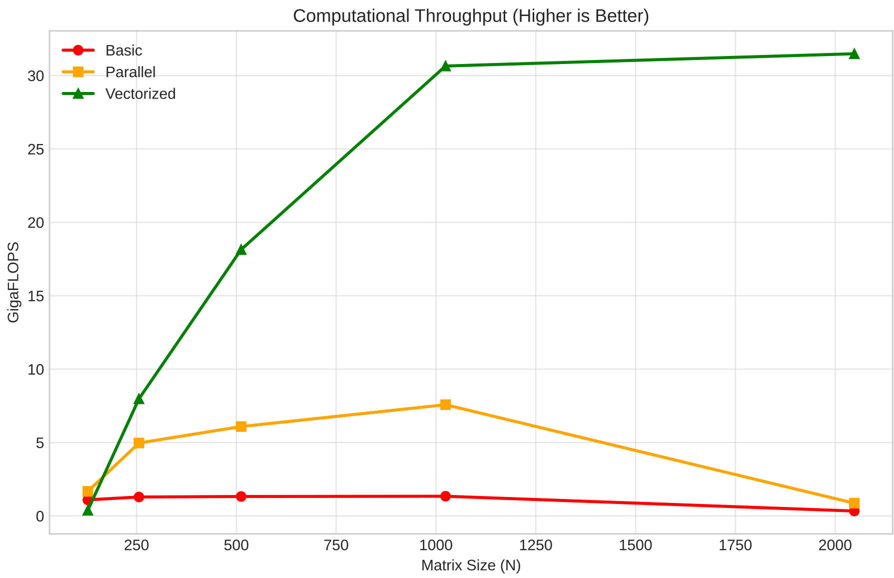

Figure 1: *Computational Throughput (Higher is Better)*

5 Performance Analysis

5.1 Execution Time and Overhead

As illustrated in Figure 2 and supported by Table 1, the performance gap widens significantly as the matrix size (N) increases.

For small matrices ($N = 128$), the **Parallel** implementation ($0.0076s$) is actually slower than the **Basic** implementation ($0.0037s$). This illustrates the *overhead of parallelism*: the time required to spawn OpenMP threads exceeds the time saved by dividing the small workload. However, at $N = 2048$, the Vectorized approach is roughly **91 times faster** than Basic.

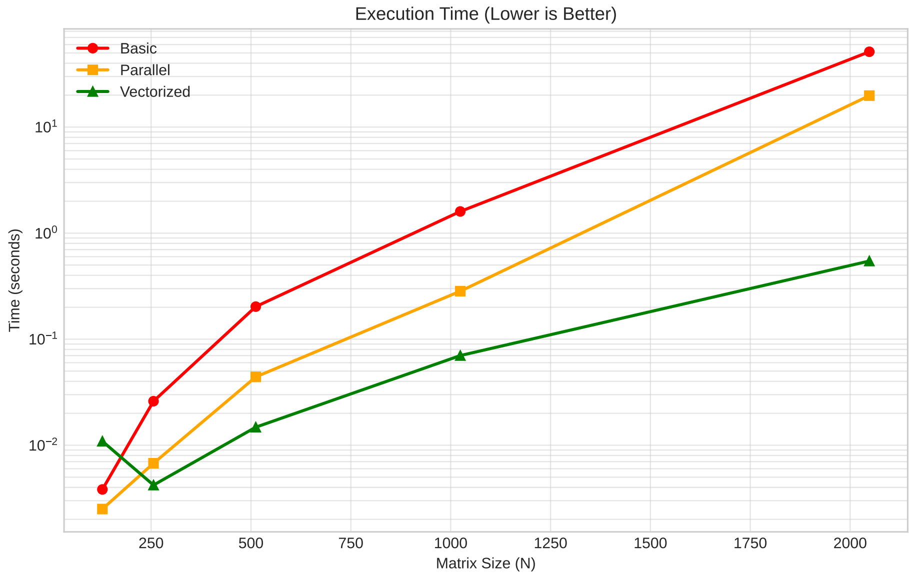

Figure 2: *Execution Time vs Matrix Size (Log Scale)*

5.2 Analysis of the "Parallel Trap"

Ideally, adding more cores should improve performance linearly. However, Figure 3 shows that the Naive Parallel version hits a wall, achieving only a 2.5× speedup at $N = 2048$.

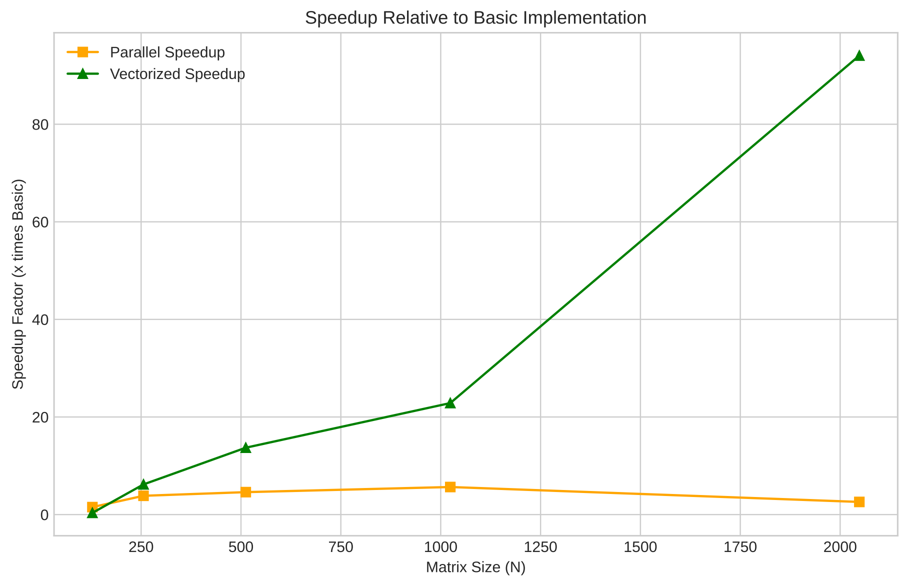

Figure 3: *Speedup Factor Relative to Basic Implementation*

The reason for this poor scaling is found in Table 3. The Parallel version suffers from extreme *Cache Thrashing*. Because threads are reading Matrix B in columns, they load full cache lines but only use a single value before discarding them. At $N = 2048$, this results in nearly **7 Billion cache misses**, forcing the CPU to stall while waiting for RAM.

Table 3: *L3 Cache Misses at varying sizes (Lower is Better)*

Size (N)	Basic	Parallel	Vectorized
512	283,197	320,573	**307,882**
1024	224,839,858	1,891,444	**2,018,394**
2048	7,927,259,288	6,912,059,223	**140,439,039**

As shown in Figure 4, the Vectorized version (which transposes Matrix B) reduces cache misses by a factor of 49 compared to the Parallel version ($140M$ vs $6.9B$).

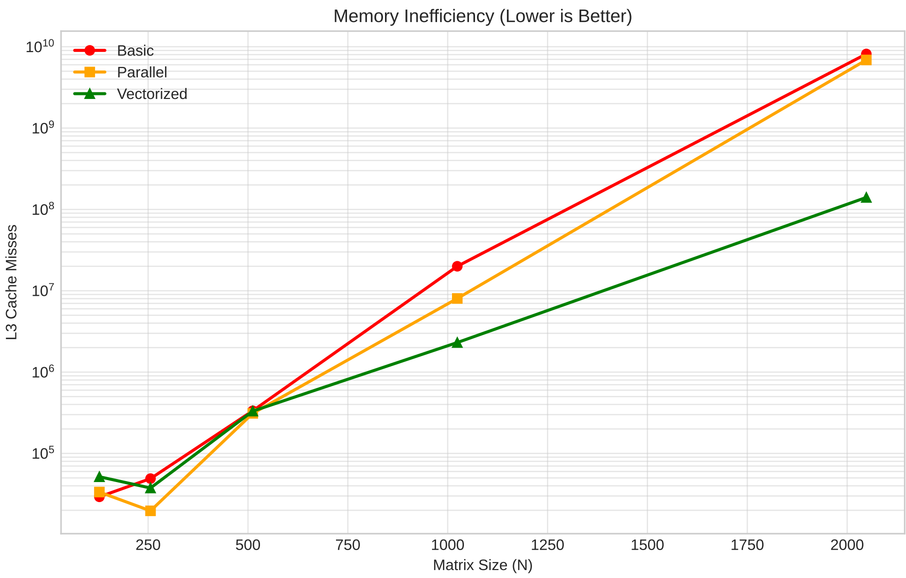

Figure 4: *L3 Cache Misses (Lower is Better)*

6 Conclusion

This study demonstrates that high performance in Big Data applications requires a holistic approach to optimization.

- **Parallelization** alone provided a modest 2.5× gain but was severely bottlenecked by memory latency.
- **Vectorization + Memory Optimization** provided a massive 91× gain.

By aligning the data in memory (Transposition) to match the hardware's access patterns (SIMD/Cache Lines), we achieved a throughput of over **31 GFLOPS**, compared to less than 1 GFLOPS for the naive parallel approach.