Parallel and Vectorized Matrix Multiplication

Víctor Blanco Dávila

December 2025

Project Repository: `https://github.com/Viblancoda/Parallel-and-Vectorized-Matrix-Multiplication.git`

1 Introduction

Matrix multiplication is a fundamental computation in scientific computing, machine learning, computer graphics, and numerical analysis. Its computational complexity makes it an attractive target for performance optimization, especially on modern architectures that offer parallel execution and hardware acceleration through vectorization. Traditional matrix multiplication based on three nested loops exhibits $O(n^3)$ time complexity and rapidly becomes computationally expensive as the matrices grow in size.

This project investigates two approaches to accelerating matrix multiplication: (1) parallel computation using multiple threads, and (2) vectorized computation using highly optimized numerical libraries. The goal is to evaluate performance improvements in terms of execution time, speedup, efficiency, and resource utilization when multiplying large square matrices.

The assignment requires the implementation of a parallel algorithm and, optionally, a vectorized algorithm. In this work, both approaches were implemented and evaluated using Python. The basic scalar algorithm was deliberately omitted, as the focus is exclusively on parallel and vectorized computation. The project also measures CPU utilization and memory consumption to better understand the computational trade-offs of each technique.

2 Methodology

2.1 Parallel Matrix Multiplication

The parallel implementation follows a row-wise decomposition strategy, where each row of the left-hand matrix is processed independently. Each task

computes the dot product between a row of matrix A and matrix B. This approach inherently exposes task-level parallelism, as each row can be computed concurrently. Python's `ThreadPoolExecutor` was used to distribute the work among available threads.

Given matrices $A \in R^{n \times n}$ and $B \in R^{n \times n}$, parallel multiplication computes:

$$C_{i,\cdot} = A_{i,\cdot} \cdot B$$

for each row i.

Although threads enable concurrency, Python's Global Interpreter Lock (GIL) limits potential speedup for CPU-bound numeric workloads. Furthermore, the overhead associated with task scheduling and inter-thread communication may dominate runtime for large matrix sizes.

2.2 Vectorized Matrix Multiplication

Vectorized computation was implemented using NumPy's optimized matrix multiplication operator:

$$C = A@B$$

NumPy internally uses highly optimized BLAS routines with SIMD and cache-aware tiling. These operations execute at C/Fortran level and bypass Python's interpreter overhead.

Vectorization provides implicit parallelism and hardware acceleration, yielding performance comparable to native compiled libraries.

2.3 Resource Monitoring

Execution time, CPU usage, and peak memory usage were recorded during each multiplication. Performance metrics include:

- **Speedup:**

$$S = \frac{T_{vectorized}}{T_{parallel}}$$

- **Efficiency:**

$$E = \frac{S}{p}$$

  where p is the number of available threads.

The experiment was performed on matrices of sizes 500, 1000, 1500, 2000 and 2500, using all logical CPU cores (8 threads).

3 Experimental Results

Table 1 summarizes the results for all matrix sizes. The vectorized approach consistently outperforms the parallel implementation by a large margin. Parallel execution time grows significantly with matrix size, while vectorized execution remains relatively low.

Table 1: Performance Summary

Size	Vectorized (s)	Parallel (s)	Speedup	Efficiency	Par CPU (%)
500	0.056	0.090	0.622	0.078	25.0
1000	0.075	0.610	0.122	0.015	25.0
1500	0.130	1.949	0.067	0.008	13.0
2000	0.273	5.199	0.052	0.007	29.2
2500	0.352	9.821	0.036	0.004	17.9

3.1 Execution Time

The execution time grows cubically with matrix size for both implementations, but the vectorized implementation is dramatically faster. For 2500×2500 matrices, vectorization takes only 0.35 seconds, while the parallel version requires nearly 10 seconds.

Figure 1 shows the time comparison between methods.

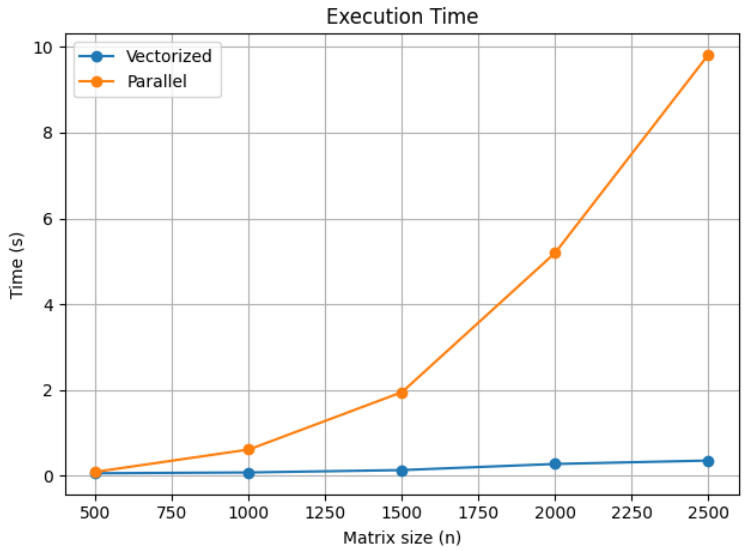

Figure 1: Execution time of vectorized and parallel approaches

3.2 Speedup and Efficiency

Speedup values are consistently below 1, indicating that parallel execution is slower than vectorization. Efficiency is extremely low (below 10%), becoming negligible for larger matrices.

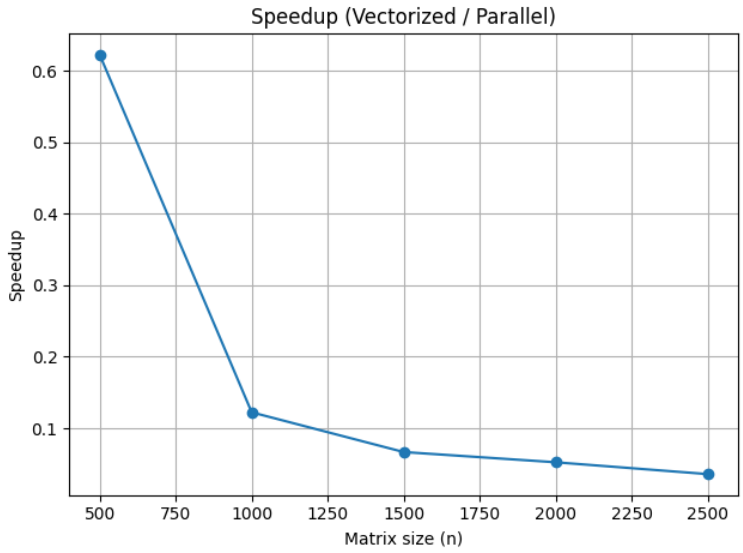

Figure 2: Speedup as matrix size increases

3.3 Resource Usage

CPU utilization was surprisingly low for the parallel implementation, which never exceeded 30%. This suggests severe bottlenecks such as:

- Thread synchronization overhead
- Python interpreter overhead
- Memory bandwidth limitations
- Non-release of GIL during numeric operations

Memory usage increases with matrix size but does not differ significantly between methods.

Figures 3 and 4 show resource usage trends.

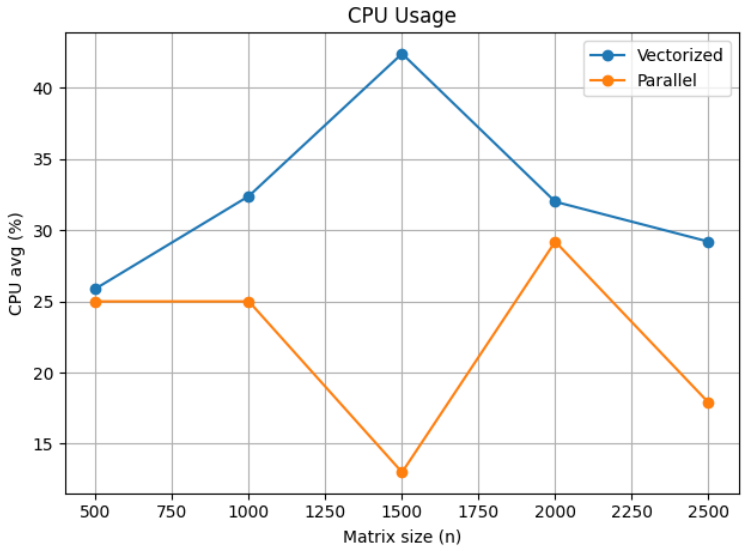

Figure 3: Average CPU utilization

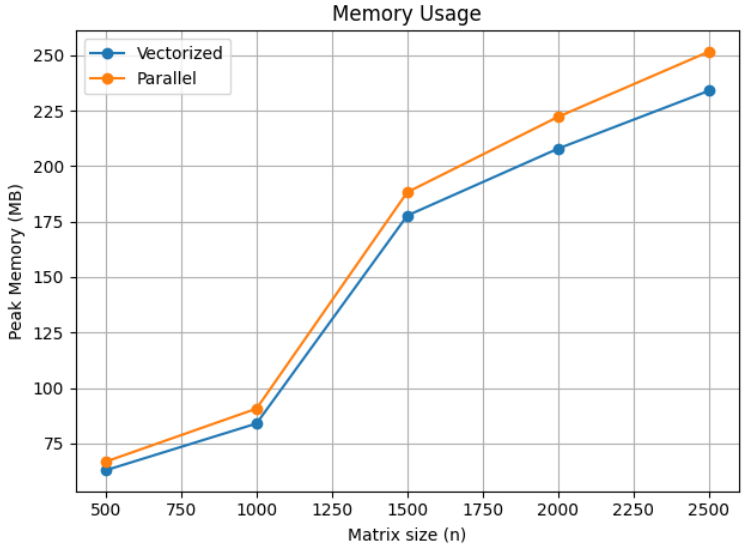

Figure 4: Peak memory usage

4 Discussion

The experimental results show that parallel matrix multiplication implemented in Python using threads performs poorly compared to vectorized execution provided by optimized numerical libraries. Rather than acceler-

ating execution, the parallel approach significantly increases runtime.

These results highlight several important considerations:

1. Python threads are unsuitable for CPU-bound numerical computation due to the GIL.
2. Vectorized libraries exploit highly optimized low-level routines, parallelization, and SIMD instructions.
3. Work distribution overhead can dominate computation time when tasks are small and numerous.
4. Efficiency decreases as matrix size increases because parallel scheduling overhead grows faster than per-task work.

Although parallelization conceptually increases concurrency, effective acceleration requires low-overhead runtimes, shared-memory parallelism, and compiler-level optimizations. Languages such as C, C++, or Fortran with OpenMP would likely yield different results.

5 Conclusions

This project implemented and evaluated parallel and vectorized matrix multiplication. Results show that:

- Vectorized multiplication is extremely fast and scales well with matrix size.
- A Python thread-based parallel approach is significantly slower, even when using all available cores.
- Speedup and efficiency of the parallel implementation decrease with problem size.
- CPU utilization and memory patterns suggest that the performance gap is caused by implementation overhead rather than computational limits.

These findings emphasize the importance of using optimized numerical libraries for high-performance computing tasks in Python. Parallelization at the interpreter level provides limited benefit for CPU-intensive operations and may even degrade performance.

Future improvements could include:

- Using multiprocessing instead of threads
- Implementing OpenMP or GPU acceleration via CUDA
- Exploring optimized block multiplication strategies