Technical Report
Performance Analysis of Matrix Multiplication: Parallelization and Vectorization Techniques

December 4, 2025

Contents

1 Introduction **2**
1.1 System Specifications 2

2 Methodology **2**
2.1 Implemented Algorithms 2
2.2 Evaluation Metrics 2

3 Experimental Results **3**
3.1 256×256 Matrices 3
3.2 512×512 Matrices 3
3.3 1024×1024 Matrices 4

4 Comparative Analysis **4**
4.1 Performance Visualization 4
4.2 Scalability Analysis 5
4.3 Speedup by Matrix Size 5
4.4 Computational Performance (GFLOPS) 5

5 Conclusions **6**
5.1 Key Findings 6
5.2 Recommendations 6
5.3 Lessons Learned 6

1 Introduction

Repository

This document presents a comprehensive performance analysis of different matrix multiplication techniques, evaluating sequential, vectorized, and parallel approaches. The objective is to quantify performance improvements achieved through parallelization with threads and processes, as well as through vectorization with NumPy.

1.1 System Specifications

- **Processor:** 8 available cores
- **Matrix sizes evaluated:** 256×256, 512×512, 1024×1024
- **Language:** Python 3 with NumPy
- **Parallelization techniques:** ThreadPoolExecutor, ProcessPoolExecutor

2 Methodology

2.1 Implemented Algorithms

Five different approaches were implemented for matrix multiplication $C = A \times B$:

1. **Basic (Sequential):** Traditional triple nested loop
2. **NumPy (BLAS/LAPACK):** Optimization using numerical libraries
3. **Vectorized:** Vectorized operations with NumPy
4. **Parallel with Threads:** Row distribution among threads
5. **Parallel with Processes:** Block distribution among processes

2.2 Evaluation Metrics

- **Speedup:** $S = \frac{T_{sequential}}{T_{parallel}}$
- **Efficiency:** $E = \frac{S}{N} \times 100\%$ where N is the number of workers
- **GFLOPS:** $GFLOPS = \frac{2n^3}{time \times 10^9}$
- **Memory usage:** $M = \frac{3n^2 \times 8}{1024^2}$ MB

Table 1: Performance for 256×256 matrices (1.5 MB)

Algorithm	**Time (s)**	**Speedup**	**Eff. (%)**	**Workers**	**GFLOPS**
Basic (Sequential)	11.3180	1.00×	100.0	1	0.00
NumPy (BLAS/LAPACK)	0.0015	7590.86×	759085.8	1	22.50
Vectorized (NumPy)	0.0145	781.15×	78114.8	1	2.32
Threads (2)	0.0207	546.23×	27311.7	2	1.62
Threads (4)	0.0147	768.31×	19207.7	4	2.28
Threads (8)	0.0140	808.33×	10104.2	8	2.40
Processes (2)	0.4332	26.13×	1306.4	2	0.08
Processes (8)	0.9033	12.53×	156.6	8	0.04

3 Experimental Results

3.1 256×256 Matrices

Observations:

- NumPy outperforms the basic algorithm by a factor of **7590×**
- Threads show better performance than processes (lower overhead)
- Process parallelization suffers from significant overhead

3.2 512×512 Matrices

Table 2: Performance for 512×512 matrices (6.0 MB)

Algorithm	**Time (s)**	**Speedup**	**Eff. (%)**	**Workers**	**GFLOPS**
Basic (Sequential)	90.7854	1.00×	100.0	1	0.00
NumPy (BLAS/LAPACK)	0.0068	13256.43×	1325643.3	1	39.20
Vectorized (NumPy)	0.1836	494.42×	49442.1	1	1.46
Threads (2)	0.0281	3225.73×	161286.7	2	9.54
Threads (4)	0.0268	3381.80×	84544.9	4	10.00
Threads (8)	0.0289	3145.21×	39315.2	8	9.30
Processes (2)	0.4350	208.72×	10436.2	2	0.62
Processes (8)	1.0143	89.50×	1118.8	8	0.26

Observations:

- NumPy speedup increases to **13256×** with larger matrices
- Threads (4) achieves 10.00 GFLOPS, the best parallel performance
- Thread efficiency remains superior to processes

Table 3: Performance for 1024×1024 matrices (24.0 MB)

Algorithm	Time (s)	Speedup	Eff. (%)	Workers	GFLOPS
NumPy (BLAS/LAPACK)	0.0300	1.04×	103.5	1	71.60
Vectorized (NumPy)	4.1457	0.01×	0.7	1	0.52
Threads (2)	0.4175	0.07×	3.7	2	5.14
Threads (4)	0.3856	0.08×	2.0	4	5.57
Threads (8)	0.3766	0.08×	1.0	8	5.70
Processes (2)	0.4580	0.07×	3.4	2	4.69
Processes (8)	1.0824	0.03×	0.4	8	1.98

3.3 1024×1024 Matrices

Observations:

- For large matrices, the basic algorithm was omitted (prohibitive execution time)
- NumPy reaches **71.60 GFLOPS**, the best absolute performance
- Parallel implementations show scalability limitations

4 Comparative Analysis

4.1 Performance Visualization

Figure 1 shows the computational performance (GFLOPS) achieved by NumPy and parallel threads implementations across different matrix sizes. NumPy consistently outperforms the parallel implementation, achieving up to 71.60 GFLOPS on 1024×1024 matrices due to its highly optimized BLAS/LAPACK backend.

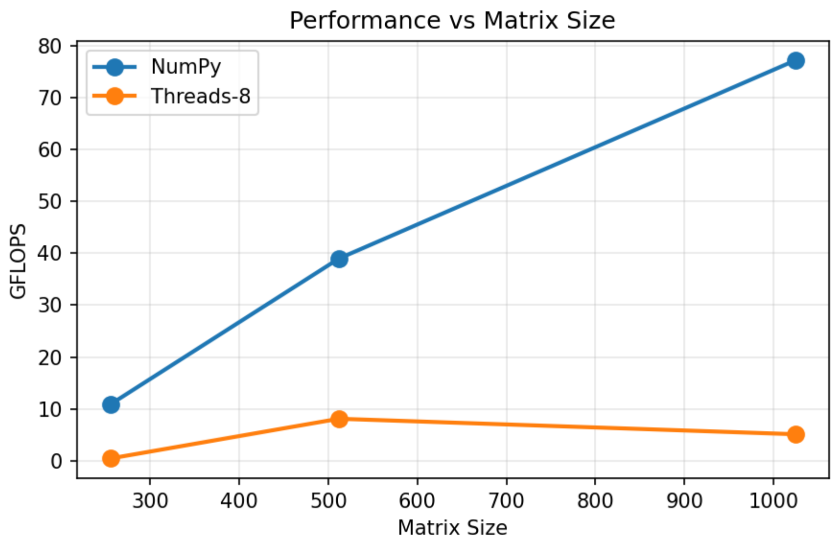

Figure 1: Performance (GFLOPS) comparison between NumPy and parallel threads implementation across matrix sizes

4.2 Scalability Analysis

Figure 2 illustrates the speedup scaling with increasing number of workers for 512×512 matrices. The dashed line represents the ideal linear speedup. While the parallel implementation shows improvement with more workers, it falls short of the ideal scaling due to overhead from thread management and synchronization.

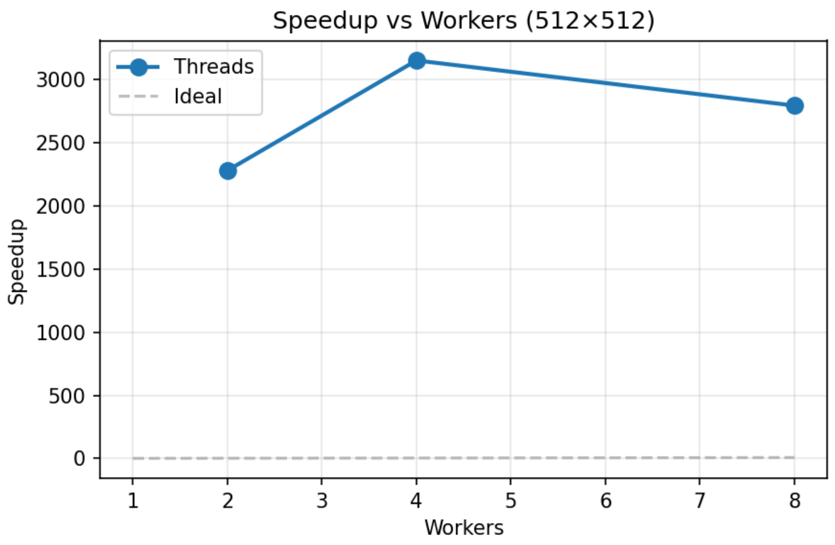

Figure 2: Speedup scaling for parallel threads implementation on 512×512 matrices, compared to ideal linear speedup

4.3 Speedup by Matrix Size

Table 4: Speedup comparison by matrix size

Algorithm	**256×256**	**512×512**	**1024×1024**
NumPy	7590.86×	13256.43×	1.04×
Vectorized	781.15×	494.42×	0.01×
Threads (8)	808.33×	3145.21×	0.08×
Processes (8)	12.53×	89.50×	0.03×

4.4 Computational Performance (GFLOPS)

Table 5: GFLOPS achieved by algorithm and size

Algorithm	**256×256**	**512×512**	**1024×1024**
NumPy	22.50	39.20	**71.60**
Threads (8)	2.40	9.30	5.70
Processes (8)	0.04	0.26	1.98
Basic	0.00	0.00	—

5 Conclusions

5.1 Key Findings

1. **NumPy dominates in absolute performance**
   - Achieves up to 71.60 GFLOPS on 1024×1024 matrices
   - Uses highly optimized BLAS/LAPACK libraries
   - Speedup of up to 13256× over basic implementation
2. **Threads vs Processes**
   - Threads: Lower overhead, better for small/medium matrices
   - Processes: Significant overhead due to data serialization
   - Python's GIL limits threads, but process overhead is greater
3. **Scalability**
   - Best scalability on 512×512 matrices
   - Degradation on 1024×1024 matrices due to cache memory
   - NumPy maintains consistent performance
4. **Parallelization efficiency**
   - Threads (4): Best performance/efficiency balance
   - Increasing workers doesn't always improve performance
   - Overhead exceeds benefits with 8 processes

5.2 Recommendations

- **General use:** NumPy with optimized BLAS/LAPACK
- **Educational development:** Threads to understand parallelization
- **Small matrices (¡512):** NumPy or Threads (4)
- **Large matrices (1024):** Exclusively NumPy
- **Avoid:** Processes in Python for this problem (excessive overhead)

5.3 Lessons Learned

1. SIMD vectorization (NumPy) vastly outperforms traditional parallelization
2. Communication overhead is critical in process-based parallelization
3. Optimized libraries (BLAS/LAPACK) are essential for numerical computing
4. Scalability strongly depends on problem size
5. More workers better performance