Optimization and OpenMP Parallelization of Matrix Addition and Product

Noa Collet and Eliott Oster

1 Introduction

This practical work aims to optimize and parallelize two fundamental operations on dense matrices:

- the **addition** of two matrices $A + B = C$,
- the **product** $C = A \times B$.

Several implementations and optimization levels were tested:

1. In-house naive and optimized implementations for better memory access,
2. Comparison with **BLAS** levels 1, 2, and 3,
3. Optimization via **cache blocking**,
4. **OpenMP parallelization**, with an analysis of scheduling and chunk size.

Performance was measured in terms of **execution time (s)** and **computational throughput (GFlops/s)** as a function of the matrix size N. All results were visualized through Python-generated plots.

2 Matrix Addition Optimization

2.1 Description

Two implementations were compared:

- `row_add`: traverses the matrix row by row (row-major order),
- `column_add`: traverses the matrix column by column (column-major order, consistent with BLAS).

2.2 Overview

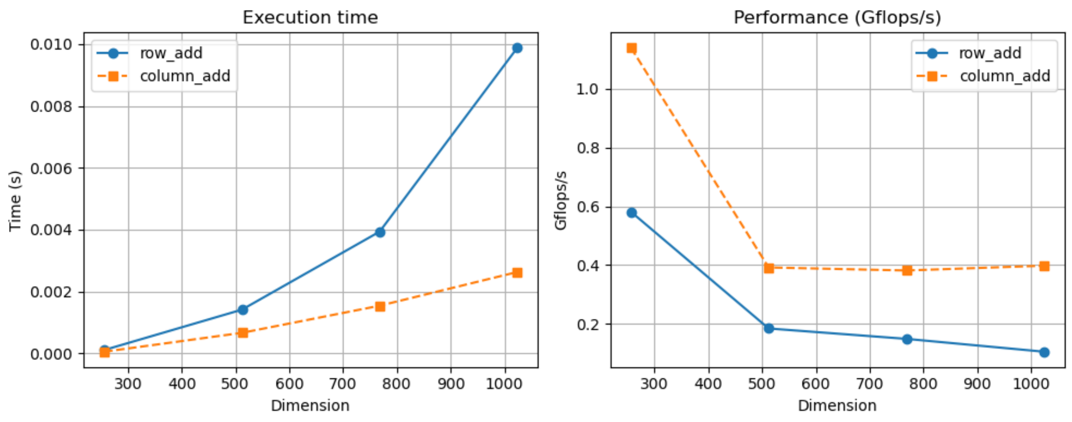

Figure 1: Comparison of row-wise and column-wise addition.

Analysis. The `row_add` version is significantly slower, especially for large matrices, while `column_add` remains roughly four times faster for $N > 1000$. The column-wise version achieves a stable performance around ≈ 0.4 GFlops/s.

Explanation. Matrices are stored in *column-major order*. Row-by-row traversal leads to non-contiguous memory accesses, producing many cache misses. Column-wise traversal ensures spatial locality, with consecutive elements stored contiguously in memory, resulting in much better cache utilization.

3 BLAS Comparison

3.1 Description

The three BLAS levels (Basic Linear Algebra Subprograms) were tested:

- **BLAS Level 1**: optimized for vector–vector operations,
- **BLAS Level 2**: optimized for matrix–vector operations,
- **BLAS Level 3**: optimized for matrix–matrix operations.

3.2 Overview

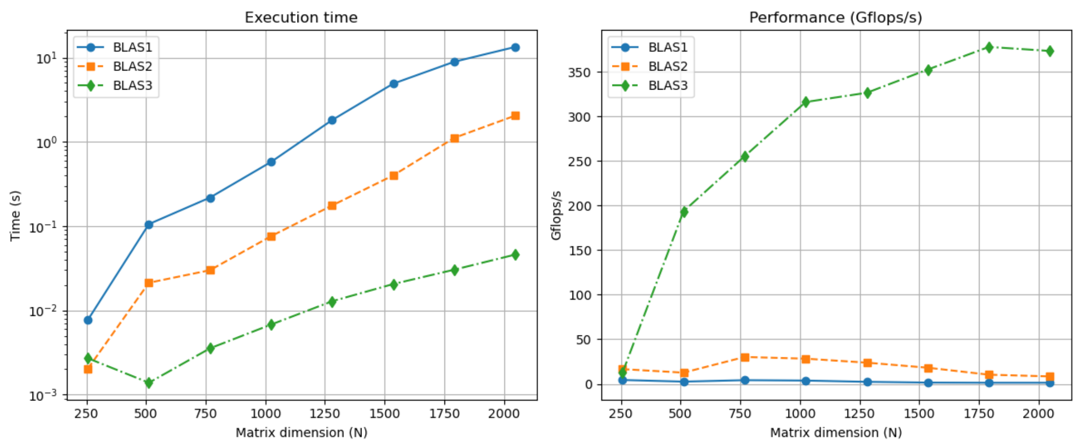

Figure 2: Performance comparison between BLAS1, BLAS2, and BLAS3.

Analysis. The execution times clearly differ as we change the level. For a matrix of dimension 2048, BLAS1 is the slowest, BLAS2 is 6.5 times faster, and BLAS3 is 13 times faster. In terms of throughput, BLAS1 reaches only a few GFlops/s and BLAS2 reaches tens, while BLAS3 exceeds **350 GFlops/s**, close to the machine's peak performance: all processor cores were operating at 100% capacity.

Explanation. BLAS3 routines maximize both computational intensity and memory locality because BLAS Level 3 is specifically optimized for the type of operations we are performing (matrix-matrix). Nevertheless, if we were performing other types of operations (e.g., matrix-vector or vector-vector), another BLAS level would be appropriate. They reuse loaded data in the cache and take advantage of vectorization and internal parallelism, leading to near-optimal throughput.

4 OpenMP Parallelization

The `inhouse_dot` function was parallelized using OpenMP to distribute the workload across the available CPU cores. The outer loop of the matrix multiplication was parallelized with the following directive:

```
#pragma omp parallel for schedule(dynamic, CHUNK_SIZE)
```

Listing 1: OpenMP Directive

This approach allows control of the scheduling policy and chunk size through the environment variable `OMP_SCHEDULE`. For instance, running the program with:

```
export OMP_SCHEDULE=dynamic,8
```

Listing 2: Environment Variable Example

means that OpenMP dynamically assigns chunks of eight iterations to each thread, redistributing work as soon as threads become idle.

4.1 Overview: Parallel vs. Sequential Performance

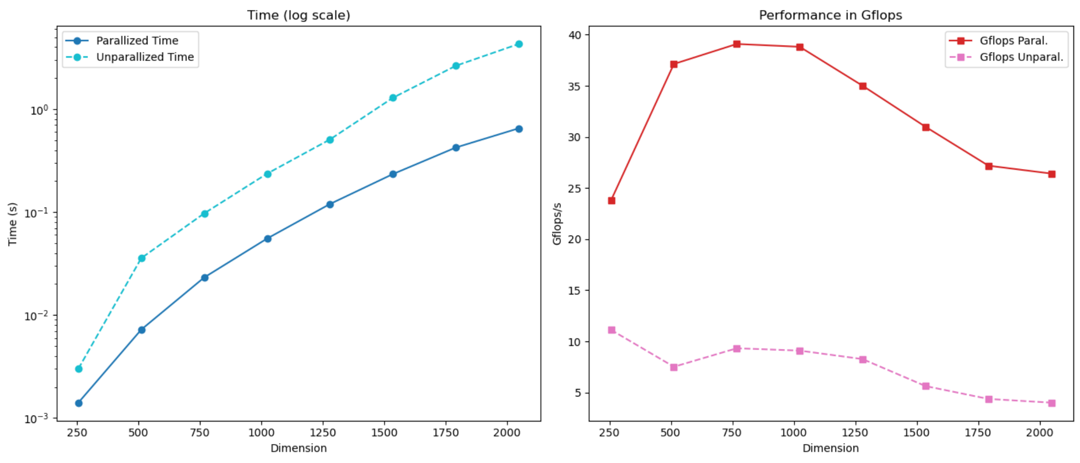

Figure 3: Comparison between parallelized and non-parallelized execution: time (left, log scale) and performance (right).

Analysis. The parallelized implementation achieves a dramatic improvement in both execution time and GFlops/s. The execution time (left) decreases by nearly an order of magnitude across all problem sizes. In terms of performance (right), the parallel version sustains between **35 and 40 GFlops/s**, while the sequential version remains below **10 GFlops/s**.

Explanation. For small matrices ($N < 500$), parallelization overhead limits speedup because thread creation and synchronization dominate. However, as matrix size increases, the computational workload per thread becomes large enough to fully amortize these costs. The scaling remains efficient up to large dimensions, showing that the implementation makes good use of available cores and memory bandwidth.

4.2 Overview: Influence of the Chunk Size

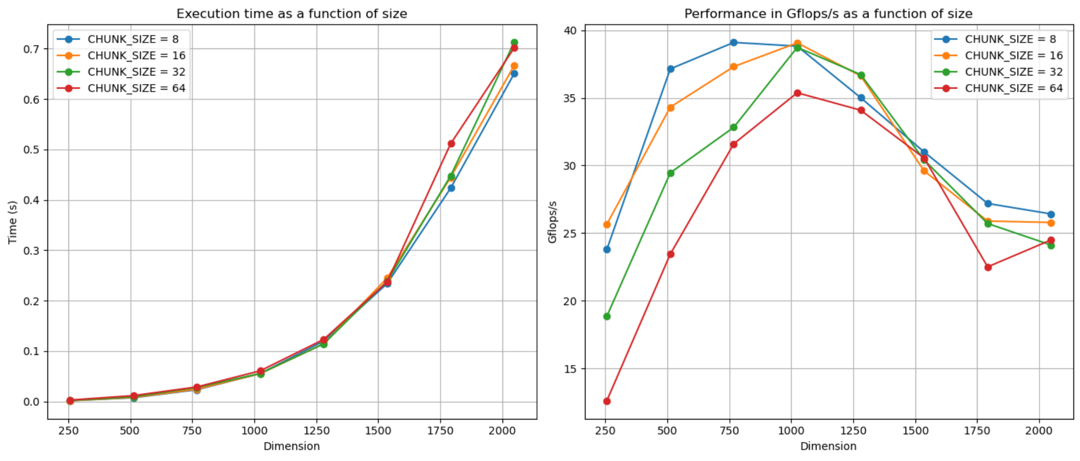

Figure 4: Impact of `CHUNK_SIZE` on execution time (left) and performance (right).

Analysis. The execution time (left) increases with the matrix size, as expected, but we observe that smaller chunk sizes (`CHUNK_SIZE = 8, 16`) consistently achieve slightly better performance than larger ones. On the right, the performance in GFlops/s peaks around `CHUNK_SIZE`

`= 8-16`, reaching close to **40 GFlops/s**. For larger chunk sizes (32 and 64), the performance drops noticeably.

Explanation. Each iteration of the outer loop performs a large amount of work, since it involves several nested loops for matrix multiplication. Because each iteration is already very heavy, the extra cost of dynamic scheduling is very small compared to the total computation time. Using a smaller `CHUNK_SIZE` helps the threads share the work more evenly: when one thread finishes its part, it can immediately take another one. This prevents some threads from being idle while others are still running. As a result, smaller chunk sizes generally lead to better performance and faster overall execution. In our case, values between **1 and 8** gave the best results.

5 Cache Blocking Optimization

5.1 Implementation Principle

The final optimization step consists in applying the **cache blocking** technique to the in-house matrix multiplication routine. The idea is to improve data locality by dividing the large matrices into smaller submatrices (blocks) that fit into the CPU cache. This way, the same data elements are reused several times while they remain in the fast cache memory, reducing the number of slow main-memory accesses.

```
#pragma omp parallel for schedule(runtime) default(none) shared(A, B, C, N,
     ld)
for (int j = 0; j < N; j += BLOCK)
  for (int k = 0; k < N; k += BLOCK)
    for (int i = 0; i < N; i += BLOCK)
      for (int jj = 0; jj < BLOCK; jj++)
        for (int kk = 0; kk < BLOCK; kk++)
          for (int ii = 0; ii < BLOCK; ii++)
            C[(j + jj)*ld + i + ii] += A[(k + kk)*ld + i + ii] * B[(j + jj)
    *ld + k + kk];
```

Listing 3: Cache Blocking Implementation

Each triple-nested block loop (ii, jj, kk) operates on a small sub-block of the matrices, improving temporal and spatial locality. This blocking pattern ensures that the working set of A, B, and C fits into cache, which is the key to the performance gain.

5.2 Overview

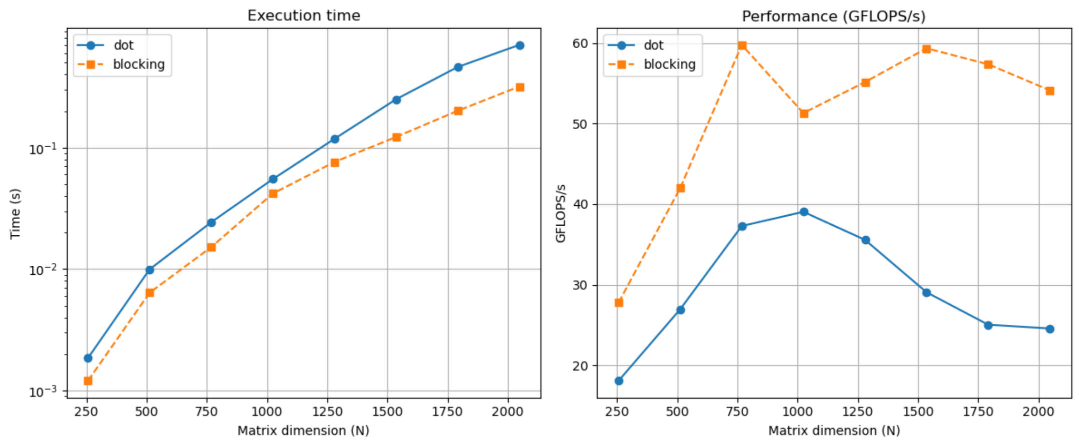

Figure 5: Comparison between the naive matrix multiplication (`dot`) and the cache-blocked version (`blocking`).

Analysis. The execution time increases with the matrix dimension as expected. However, the blocked version is consistently faster across all sizes, often by nearly one order of magnitude for large matrices ($N > 1000$). In terms of computational throughput, the naive version (`dot`) reaches at most **40 GFlops/s**, while the blocked version sustains up to **60 GFlops/s**.

Explanation. The gain arises from the improved **cache reuse**: once a sub-block of matrix A or B is loaded into cache, it is reused for multiple computations before being replaced. This drastically reduces the number of memory transfers and cache misses. As matrix sizes grow, this optimization becomes increasingly beneficial since memory bandwidth becomes the bottleneck.

6 Conclusion

This study highlights the importance of efficient memory usage and parallel computation in achieving high performance for dense matrix operations. Throughout the experiments, several key aspects have been analyzed and optimized: memory access patterns, cache locality, blocking techniques, and parallel execution with OpenMP.

The obtained results demonstrate significant improvements in computational efficiency. The optimized implementations achieve speedups ranging from approximately **4 to 10 times** compared to the naive versions, reaching peak performances of about **40 to 60 GFlops/s**. Nevertheless, the performance achieved with our in-house implementations remains significantly lower than that obtained with the highly optimized **BLAS Level 3** routines, which reach performances of about **350 GFlops/s**.