

Optimizing Square Matrix Symmetry Verification and Transposition Using Implicit Parallelization Techniques and OpenMP

Mosè Arcaro, ID:226616

*Department of Engineering and Computer Science
Faculty of Engineering, Università degli studi di Trento
Trento, Italy
mose.arcaro@studenti.unitn.it*

Abstract—This paper presents a comparative analysis of three implementations of square matrix operations (transposition and symmetry check functions): sequential, optimized, and parallel using OpenMP. The study quantifies the performance improvements achieved through optimization techniques and parallelization, demonstrating significant speedups and efficiency gains in both implementations compared to the sequential version, except for very small matrices and cases with excessive thread usage where performance deteriorates. FOR ALL DATA AND CODES VISIT THE LINKED GITHUB REPOSITORY

I. INTRODUCTION

Matrix operations are fundamental in numerous scientific computing applications, from image processing to numerical simulations. The performance of these operations can significantly impact overall system efficiency.

The primary goal of this project is to compare three distinct implementations of square matrix operations:

- Basic sequential implementation
- Compiler-optimized sequential version
- Parallel implementation using OpenMP

The comparison focuses on execution time to determine the effectiveness of each optimization approach.

II. STATE OF THE ART

Matrix transposition optimization has been extensively studied in literature [15]. There are numerous techniques to optimize this process (like blocking technique [4] [8] or vectorization [3]). While extensive research exists on matrix transposition optimization, several gaps remain:

- Lack of systematic comparison between basic, optimized, and parallel implementations
- Need for practical guidelines based on matrix size and hardware capabilities
- Study of thread count impact in relation to matrix size

Our work aims to evaluate and find the best combination of techniques to optimize the time performance of matrix transposition and symmetry check to show a complete view of various possibilities.

III. CONTRIBUTION AND METHODOLOGY

This project addresses these gaps by: Providing systematic performance comparison; Quantifying combined optimization effects and delivering practical implementation insights.

The sequential version consists of two simple for loops that traverse all matrix values (both for the symmetry check function and transpose calculation). We chose to traverse all elements to make the results independent of the matrix content. The optimized version involves modifying the for loops to optimize memory access and using various pragma instructions and compilation flags [2] to find the optimal version. The parallel version uses combined OpenMP techniques [10] and memory access optimizations. All versions traverse all matrix elements to highlight the improvement. All versions were studied with float data type. The study was conducted on a system with 64-byte aligned memory [6] and implemented square cache blocking methodology with size 64 [5]: the power of two lower than the calculation:

$$\sqrt{\frac{\text{cache cluster}}{\text{dimension of Float}}} = 90 \quad (1)$$

to ensure the block fits entirely in L1 cache (32k), size 64 is also confirmed by system specifications. An iterative process was conducted testing different implementations and comparing results to identify the best version.

IV. EXPERIMENTS AND SYSTEM DESCRIPTION

A. Hardware

UniTn Cluster with: Ram: DDR SRAM; [13] Processors: Intel® Xeon® Gold 6252N Processor 35.75M Cache, 2.30 GHz; [12]

B. Software

Compiler: GCC 9.1; Libraries: stdio.h, stdlib.h, time.h, immintrin.h, sys/time.h, omp.h, getTimeOfDay [14]; OpenMP version 3.1; File PBS; File CSV; Python 3.7.2 with pandas, matplotlib.pyplot, sys, os;

C. Sequential Version

pragma option override(checkSym, "opt(level, 0)") was used to remove all possible compiler optimizations

| Dim | Check Sym | Transposition |
|------|-----------|---------------|
| 4 | 0.000001 | 0.0000004 |
| 8 | 0.0000016 | 0.0000006 |
| 16 | 0.0000032 | 0.0000016 |
| 32 | 0.0000096 | 0.000007 |
| 64 | 0.00004 | 0.0000384 |
| 128 | 0.0001404 | 0.000163 |
| 256 | 0.0006166 | 0.0014622 |
| 512 | 0.0030484 | 0.0047096 |
| 1024 | 0.0053626 | 0.0205132 |
| 2048 | 0.0260366 | 0.0898084 |
| 4096 | 0.2214532 | 0.37326 |

TABLE I
RESULTS OF THE SEQUENTIAL CODE

D. Optimized Version

For the optimized sequential version, I tested these flags and pragmas [2] [11]:

- -O2
- -ftree-loop-distribution: This flag attempts to split a single loop into multiple separate loops
- -floop-interchange: This optimization allows changing the order of nested loops
- -floop-unroll-and-jam: combines loop unrolling with jamming for nested loops
- -falign-loops: tells the compiler to align loop starts to specific memory boundaries for cache optimization
- pragma isolated call(): is a compiler extension indicating that a function call has no side effects or dependencies with other code parts
- pragma unroll: is an instruction for the compiler suggesting to unroll a loop, replicating the loop body to reduce iteration overhead
- pragma execution frequency(very high): is a hint to the compiler indicating that a specific code block will be executed very frequently during program execution
- pragma inline: suggests to the compiler to inline a function, replacing the call with the function body itself
- builtin prefetch: is a GCC compiler intrinsic function that suggests the processor to preload data in cache [1]

Additionally, I implemented matrix blocking with 64-sized blocks, the best size for optimizing memory access

| AllFlagsNoO2 | OnlyO2 | PragmaO2 | Best | |
|--------------|-----------|-----------|-----------|----------------|
| 0.3717912 | 0.1627616 | 0.157442 | 0.0848064 | 4096 transpose |
| 0.0001922 | 0.0001562 | 0.0001608 | 0.0001406 | 128 transpose |
| 0.0191116 | 0.0080258 | 0.0073168 | 0.0068766 | 1024 transpose |
| 0.2211534 | 0.1310578 | 0.1303696 | 0.0538312 | 4096 symmetry |
| 0.0001562 | 0.0000582 | 0.0000542 | 0.0000788 | 128 symmetry |
| 0.007266 | 0.0071528 | 0.0073014 | 0.0045826 | 1024 symmetry |

TABLE II
RESULTS OF THE DIFFERENT VERSIONS WITH 4096 AND 128 DIM

Therefore, the best version is the "Best" version from Table II, which implements matrix blocking with the addition of: -O2 -floop-interchange -floop-unroll-and-jam and the pragmas mentioned above. Note however that for symmetry checking, where fewer memory accesses are required, these aggressive optimizations interfere with each other for small matrices when used together, leading to efficiency loss while still remaining better than the sequential version.

E. OpenMP

First, a version was implemented that aimed to avoid implementing blocking, using only pragma instructions like parallel for, collapse and simd [7], and then another using blocking together with other pragmas [10] to improve locality such as proc bind(close), prefetch, aligned; this last combination was crucial for the final result showing great performance improvements. Note how performance significantly improved implementing Blocking techniques (Fig. 1) vs (Fig. 2) Also note how using static scheduling shows negative performance peaks (Fig. 3) and how they disappeared using dynamic scheduling (Fig. 2) The best version can be seen in Fig. 4 and 5, in logarithmic scale for better view

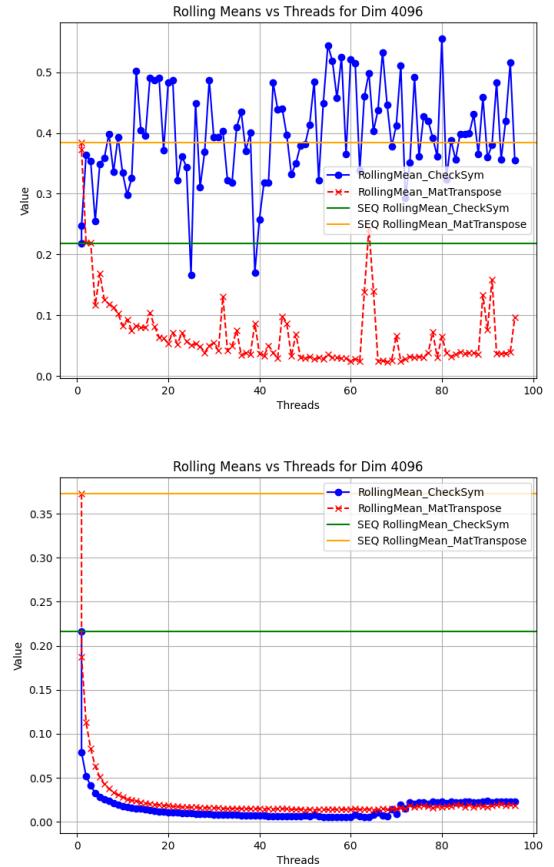


Fig. 2. Performances With Blocking and Scheduling Dynamic

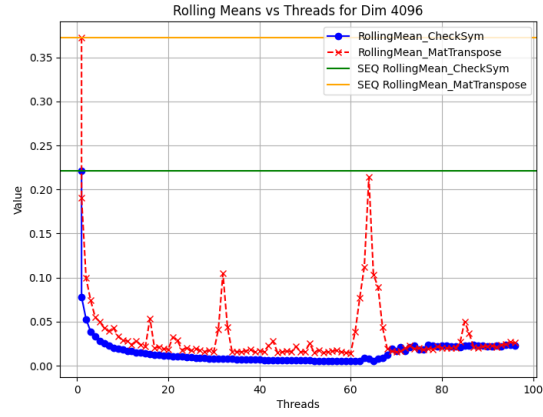


Fig. 3. Scheduling Static

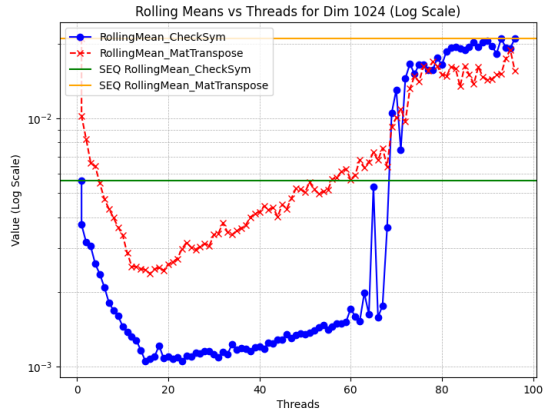


Fig. 4. Time for Threads, Dim = 1024

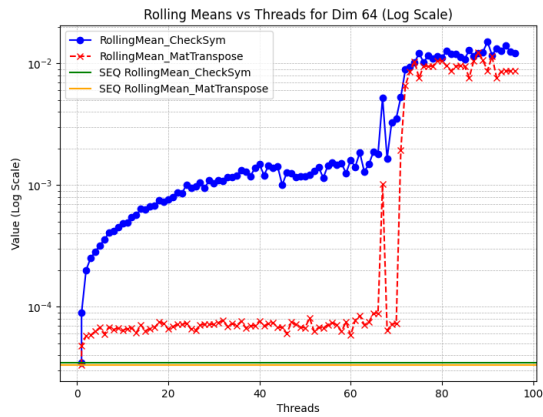


Fig. 5. Time for Threads, Dim = 64

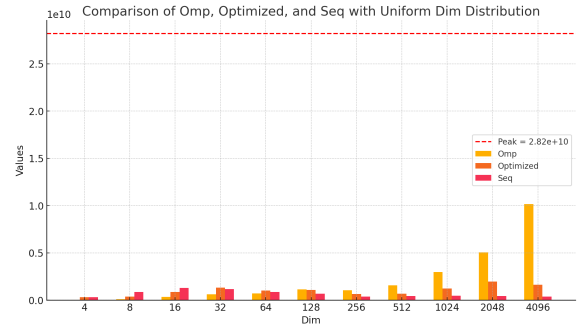


Fig. 6. Bandwidth Used

V. RESULTS AND DISCUSSION

Let's now analyze speedup(2) and efficiency(3) for better understanding of results

$$\frac{\text{Time On Sequential Version}}{\text{Time On N processors}} = \text{speedup} \quad (2)$$

$$\left(\frac{\text{speedup}}{\text{number Of Processors}} \right) \cdot 100 = \text{efficiency} \quad (3)$$

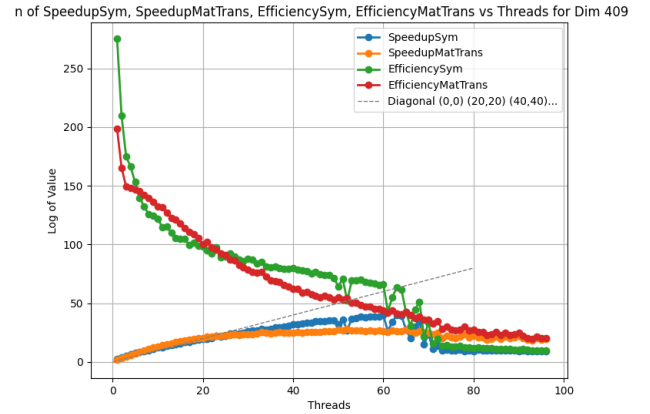


Fig. 7. Efficiency and Speedup for Dim = 2048

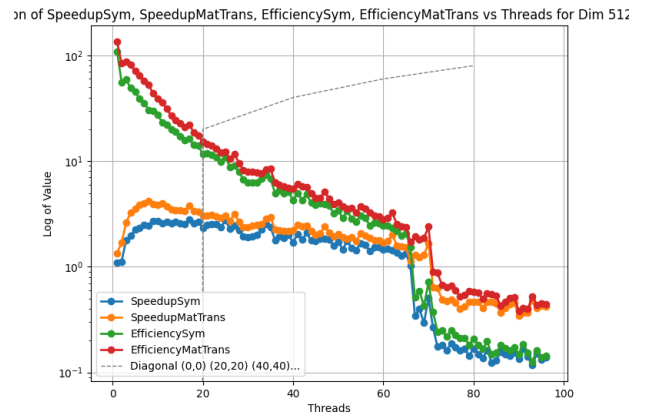


Fig. 8. Speedup and Efficiency for Dim = 512 (log scale)

As can be seen, there is a significant improvement for large matrices, deviating slightly from ideal speedup (gray line)(Fig. 7), while for smaller matrices there is actually a deterioration caused by thread overhead related to thread creation and management(Fig.8). For the same reason, beyond a certain number of threads, performance begins to decrease(Fig. 8). Furthermore, as shown by the graphs, the efficiency of each thread tends to decrease as their number increases, for the same reason. As time increases, efficiency decreases, as explained by Amdahl's Law [9]: which illustrates how sequential parts of a program become the bottleneck as threads increase, reducing efficiency(Fig. 7) It can also be noted that the transposition operation requires more time compared to symmetry checking, due to the sparse read and write operations it must perform compared to the simple data checking of symmetry control.

A. Memory Bandwidth of Transposition

The theoretical Bandwidth of the system was calculated:

$$\text{MemoryClock} \cdot \text{TransferRate} \cdot \text{BusWidth} \cdot \text{MemChannelPerSocket} = \text{PeakBandwidth} \quad (4)$$

therefore:

$$2.933 \times 10^6 \cdot 2 \cdot 8 \cdot 6 = 28200000000 \quad (5)$$

And the Bandwidth of various code versions:

$$\frac{\text{Dim} \cdot \text{Dim} \cdot \text{FloatDimension} \cdot \text{NumberDataTransfers}}{\text{timeTaken}} = \text{BW} \quad (6)$$

| Dim | Omp | Optimized | Seq | Peak |
|------|-------------|-------------|-------------|-------------|
| 4 | 21333333.33 | 320000000 | 320000000 | 28200000000 |
| 8 | 85333333.33 | 365714285.7 | 853333333.3 | |
| 16 | 341333333.3 | 853333333.3 | 1280000000 | |
| 32 | 630153846.2 | 1321290323 | 1170285714 | |
| 64 | 694237288.1 | 1011358025 | 853333333.3 | |
| 128 | 1147740806 | 1083239669 | 689127234.5 | |
| 256 | 1045231260 | 644880688.8 | 358561072.4 | |
| 512 | 1572785361 | 665635751.9 | 445293018.5 | |
| 1024 | 2965220219 | 1238317144 | 453114966.6 | |
| 2048 | 5050335942 | 1955432061 | 434651909.9 | |
| 4096 | 10153855837 | 1618434305 | 369690608.2 | |

TABLE III
BANDWIDTH COMPARISON

For each dimension, we used the version that required the least time, regardless of the number of threads used. The bandwidth graph can be observed in Fig. 6.

As can be seen in Table III, the program's bandwidth increases with matrix size, and the best version is the OpenMP version which, for dimension 4096, is about one-third of the system's peak bandwidth.

B. Problems Encountered

Problems encountered: The OpenMP version initially had very slow symmetry check function due to atomic instruction of all threads in all threads, this was resolved using nowait (doesn't cause problems in this case) and writing to the

common variable only at the end of cycles and if necessary. It would have been useful to try and test the behavior of omp cancel [16] but it wasn't possible because only OpenMP version 3.1 was present in the system. Additionally, implementing matrix blocking was crucial for noting enormous performance improvements. Negative performance peaks were noticed when the number of threads approached numbers 16, 32, 64 because using 16, 32, or 64 threads, it's possible to align with cache-line size. This can lead to cache thrashing problems [17] or cache overload, as multiple threads try to access memory positions residing in the same cache-line simultaneously, causing conflicts; using schedule(dynamic) instead of schedule(static) distributes work uniformly allowing threads that finish their block to take another's block, thus avoiding these conflict-related peaks.

CONCLUSIONS

We have shown 3 versions for performing matrix transposition operation and symmetry checking, we have thus seen various optimization methods both at compiler level and by parallelizing using OpenMP, demonstrating how using a multithread approach brings indisputable benefits when working with large matrices, while for small matrices, a sequential approach is preferable. Also confirming how the matrix blocking technique proves to be a great ally in their optimization.

GIT AND INSTRUCTIONS FOR REPRODUCIBILITY

You can find all complete data and codes here, with the instructions for reproducibility <https://github.com/mosearc/ParallelProgramming.git>

REFERENCES

- [1] <https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html> Prefetch
- [2] <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html> flag optimization
- [3] <https://www.sciencedirect.com/topics/computer-science/instruction-level-parallelism> ILP
- [4] <https://ieeexplore.ieee.org/document/6859580> <https://suif.stanford.edu/papers/lam-aspl91.pdf> https://www.researchgate.net/publication/378813407_Matrix_Transposition_Algorithm_Using_Cache_Oblivious <https://stackoverflow.com/questions/8107965/concept-of-block-size-in-a-cache> Blocking
- [5] <https://stackoverflow.com/questions/5401464/finding-the-cache-block-size> find Block size
- [6] <https://stackoverflow.com/questions/794632/programmatically-get-the-cache-line-size> find Alignment
- [7] <https://www.intel.com/content/www/us/en/docs/dpcpp-cpp-compiler/developer-guide-reference/2024-2/prefetch-noprefetch.html> Pragma prefetch
- [8] https://www.cita.utoronto.ca/~merz/intel_c10b/main_cls/mergedProjects/optaps_cls/common/optaps_perf_optstrats.htm optimization strategies
- [9] https://it.wikipedia.org/wiki/Legge_di_Amdahl Amdahl Law
- [10] <https://www.openmp.org/spec-html/5.0/openmpsu42.html#x65-1390002.9.3> <https://www.openmp.org/spec-html/5.0/openmp.html> OpenMP
- [11] https://www.ibm.com/docs/en/zos/3.1.0?topic=category-optimization-tuning#pragmas_optznPragmas
- [12] <https://ark.intel.com/content/www/us/en/ark/products/193951/intel-xeon-gold-6252n-processor-35-75m-cache-2-30-ghz.html> https://docs.google.com/spreadsheets/d/1-n4q-nFSHiUgUh_PSmsFS1VwP_ZUitkCSyDvz2As0E/edit?gid=1254941127#gid=1254941127 <https://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%206252N.html> Info CPU

- [13] https://en.wikipedia.org/wiki/DDR_SDRAM Info Ram
- [14] <https://www.delftstack.com/howto/c/gettimeofday-in-c/> gettimeofday()
- [15] <https://www.intel.com/content/www/us/en/docs/advisor/cookbook/2023-0/optimize-memory-access-patterns.html> optimize memory access pattern
- [16] <http://jakascorner.com/blog/2016/08/omp-cancel.html> omp cancel
- [17] [https://en.wikipedia.org/wiki/Thrashing_\(computer_science\)](https://en.wikipedia.org/wiki/Thrashing_(computer_science)) cache trashing