Optimization using Implicit and Explicit Parallelism with OpenMP

Nivriti Raisingh
Dept. of Information Engineering and Computer Science
Università di Trento
Trento, Italy
nivriti.raisingh@studenti.unitn.it

***Abstract*—This study investigates optimization techniques for matrix transposition using implicit and explicit parallelization methods. It explores low-level optimizations and OpenMP parallelization strategies, evaluating their efficiency, scalability, and computational efficiency against the baseline through rigorous performance metrics.**

I. Introduction

Matrix transposition, a fundamental operation in computational mathematics, involves flipping a matrix along its diagonal to interchange rows and columns. This operation is essential in various fields, including numerical simulations and graphics processing [8]. It is applied in wavelet transforms for large-scale image processing, linear regression [9], and multidimensional Fast Fourier Transforms (FFTs) [5] [6].

In this report, we identify the bottlenecks in matrix transposition, propose optimized and parallelized approaches, and compare them to the sequential baseline. These approaches uses techniques like implicit and explicit parallelisation.

The report is structured as follows: Section II discusses the state-of-the-art, solutions, and their limitations. Section III presents the contributions and methodologies. Section IV describes the system and experimental setup. Section V presents the results and discussions of the experiments. Finally, Section VI provides a summary.

II. State-of-the-art

Recent studies have emphasized optimizing memory hierarchies, maximizing workload distribution, and leveraging modern parallel frameworks to enhance efficiency and scalability. Matrix transposition algorithms are hindered by inefficient memory access patterns [13], cache miss penalties [10], and slow I/O operations [12].

Existing solutions, such as the **cycles method** for in-place matrix transposition [10], offer efficient memory access and minimal memory overhead. However, this method is not scalable for large matrices and is purely sequential, making it unsuitable for parallelization. Its execution time suffers due to non-contiguous memory accesses, which is severely inefficient for the cache [13] [7].

Cache-oblivious algorithms, like the **Recursive MaxSquare** [13], aim to reduce cache misses by being inherently cache-optimal. This algorithm uses recursive block-partitioning for in-situ matrix transposition, enhancing performance by minimizing cache misses. However, it faces challenges such as limited memory-to-cache associativity and inefficiencies with irregular matrix sizes. Furthermore, low-level performance tuning techniques, like register tiling and array alignment, can distort the algorithm's benefits.

The **Basic Linear Algebra Subprograms (BLAS)** [1] provides robust and efficient support for matrix operations, leveraging hardware-specific optimizations like vectorization, pipelining, and memory prefetching [2]. These routines are scalable and memory-efficient, reducing unnecessary overhead. However, they are pre-optimized and depend on the underlying hardware.

III. Contribution and Methodology

A comprehensive methodology for optimizing matrix transposition, comprising three distinct approaches: **sequential implementation**, **implicit parallelization**, and **explicit parallelization**. The sequential implementation serves as a baseline for performance comparisons. Implicit parallelization techniques optimize memory access patterns, reducing memory latency and improving arithmetic intensity. Parallel implementations utilizes threads through OpenMP [3] [4] to enhance performance. Performance metrics, including speedup (Eq.1), scalability (Eq.2), effective memory bandwidth (Eq.3) and the Roofline Model, are employed to compare each algorithm's performance [14] [15]. Python is used to generate plots of the data and visualize the results.

$$Speedup = \frac{T_{seq}}{T_{par}} \tag{1}$$

$$Scalability = \frac{Speedup}{Num_{threads}} \tag{2}$$

$$Memory\ Bandwidth = \frac{2 \cdot N \cdot N \cdot sizeof(float)}{time} \tag{3}$$

This project handled several challenges, including false sharing, thread contention, and load imbalance, which are mitigated through fine-tuning the number of threads, optimization flags, and compiler directives. Additional challenges, such as minimizing cache misses, reducing memory overhead,

and improving cache efficiency, are addressed using cache-oblivious algorithms. The project also considers hardware dependency, tailoring solutions for general architectures by performing tests on multiple systems to identify the most efficient solution.

IV. Experiment and System Description

The experiments were conducted on two distinct computing systems:

- Dell Inspiron laptop with an AMD Ryzen 5500U CPU, 6 cores, 12 threads, 8GB RAM, and 3 levels of cache, running Windows 11, theoretical peak performance of 384 GFLOPS4. No other applications were running during the experiments. [1]
- Unitn High-Performance Cluster with an Intel Xeon Gold 5218 CPU, 64 cores, 128 threads, running Linux CentOS 7, theoretical peak performance of 355.09 TFLOPS.

The experiments utilized the GNU Compiler Collection (GCC) with versions 9.3.0 and 9.1.0 on the laptop and cluster, respectively. Matrices of sizes 2^4-2^{12} filled with random floating-point numbers were used as input. The libraries employed were: *chrono,cstdlib, fstream, iostream, algorithm* and *omp.h*. Experiments were run 10 times on each approach, and the average values was recorded and analysed.

*1) **Sequential Implementation**:* **checkSym** exhibits a time complexity of $O(\frac{(n^2)}{2})$. Although more efficient algorithms exist, this report focuses on measuring the performance of the worst-case scenario.

matTranspose exhibits a time complexity of $O(n^2)$). To determine the most efficient memory access pattern, an experimental investigation was conducted, implementing two functions: (1) reading matrix **M** in row-major order and writing the transposed matrix **T** in column-major order, and (2) reading matrix **M** in column-major order and writing the transposed matrix **T** in row-major order. The optimal access pattern was selected as the baseline for the analysis of parallelized implementation.

*2) **Implicit Parallelization**:* Four different approaches were implemented for each function, utilizing compiler directives to reduce memory latency and improve arithmetic intensity. Concepts such as blocking, vectorization and unrolling were tested. An experimental investigation was conducted to determine the most efficient optimization flags.

It is important to note that the **O3** flag is not mentioned due to its aggressiveness, which may lead to unpredictable behaviour. The **Os** flag is not mentioned as well since it is used to optimize the code size and not the performance.

*3) **Explicit Parallelization**:* OpenMP directives are employed to divide the workload across threads. Work-sharing strategies, such as block-based partitioning, are tested to minimize thread contention and maximize parallel efficiency. Further experiments were carried out to determine the optimal number of threads and understand its scalability.

The following hypotheses were formulated for this report:

- Implicit parallelization will offer speed up but will hit a threshold or bottleneck at a certain point.
- Explicit parallelization will not be effective for small matrices but offer significant speedup for larger matrices.
- The speedup will be proportional to the number of threads used but it will not be linear due to overheads.

V. Results and Discussion

*1) **Best access pattern for the naive method**:* The results, illustrated in Fig.1, demonstrate that the row-order reading approach is approximately twice as fast, primarily due to reduced cache misses. Although writing to contiguous memory locations offers significant benefits, the overhead of non-contiguous reads outweighs these advantages. Consequently, an efficient access pattern becomes increasingly crucial as the matrix size increases. This algorithm is thus used as the baseline for calculating speedup, scalability, and effective bandwidth.

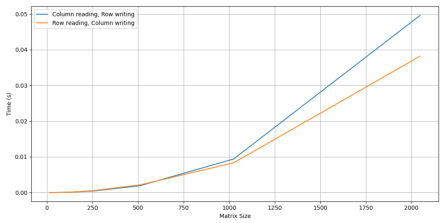


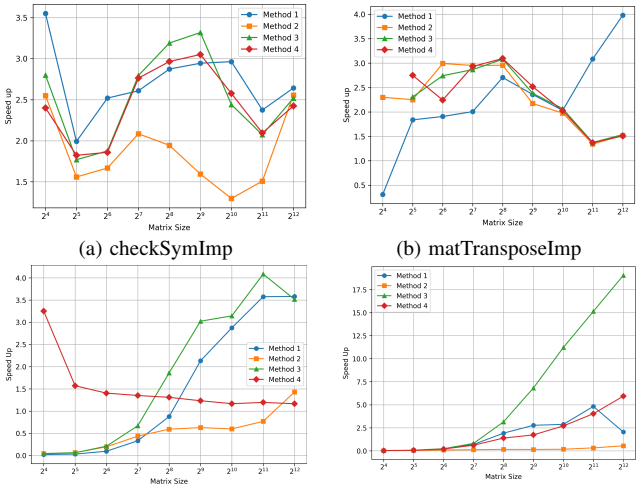
Fig. 1. Time vs. Matrix size : Access pattern comparison for Column-major reading and Row-major reading

*2) **Implicit Parallelization**:* Fig.I[(a) **checkSymImp**] indicate that method 1 achieves the highest overall speedup, ranging from 2x to 3.5x. The *ivdep* directive enables aggressive optimizations by instructing the compiler to assume no dependencies within the loop. This improvement is likely due to better utilization of CPU vector units and cache-friendliness. Method 3 and Method 4 exhibit similar speedups, ranging from 1.7x to 2.7x. Both methods employ the *unroll* directive. The relatively lower performance could be attributed to increased code and instruction size pressure, potentially leading to cache misses. Additionally, unroll does not perform well with branch checks, leading to moderate speedup. Method 2 shows the least speedup, ranging from 1.2x to 2x. This is likely due increase in the number of loops and inefficient optimization of cache handling by the blocking approach. Additionally, inefficient branch prediction leads to pipeline stalls.

Fig.I[(b) **matTransposeImp**] indicate that the speedup of method 1 increases significantly with matrix size, ranging from 0.2x to 4x. This method utilizes blocking with a block size of 16 elements. Given that the L1 cache is 32KB, both sub-matrices can fit into the cache, providing better spatial locality. The L2/L3 cache further reduces the pressure, enhancing locality. The other methods exhibit relatively consistent performance, with speedups ranging from 1.3x to 3x.

[1]

$$\text{Peak Performance} = FLOPS \times num_{cores} \times \text{max clock rate} = 16 \times 6 \times 4 \times 10^9 = 385 GFLOPS \quad (4)$$



(a) checkSymImp (b) matTransposeImp

(c) checkSymOMP (d) matTransposeOMP

TABLE I
SPEEDUP VS MATRIX SIZE : DEMONSTRATES THE PERFORMANCE OF THE IMPLICIT AND EXPLICIT APPROACHES

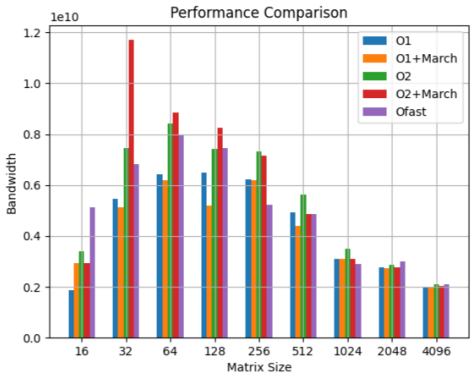


Fig. 2. Bandwidth vs. Matrix size : Effect of Optimization Flags on matTransposeImp

The substantial difference in performance of blocking on *checkSymImp* and *matTransposeImp* is due to the memory-bound nature of latter.

The **O2+March** optimization is the best performing, as shown in Fig.2. It provides a good balance between optimization and compilation time, enabling a wide range of optimizations that improve performance. It makes faster and enhanced decisions about unroll factors while preserving the memory access patterns needed for unrolling. However, memory bandwidth and cache efficiency become bottlenecks as matrix size increases, and the bandwidth performance of all the flags converges.

*3) **Explicit Parallelization**:* Fig.I[(c) **checkSymOMP**] indicate that method 1 achieves the highest overall speedup, ranging from 0.2x to 4.1x. Speedup increases significantly with size. This method utilizes *reduction*, which minimizes synchronization overhead by combining results at the end. Method 2 has moderate performance. Although safe, the use of atomic operations introduces overhead, which does not scale well for larger sizes. Each thread in Method 3 declares a new

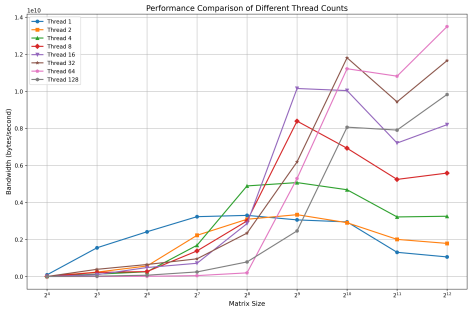


Fig. 3. Bandwidth vs. Matrix size: Showing Effect of Threads

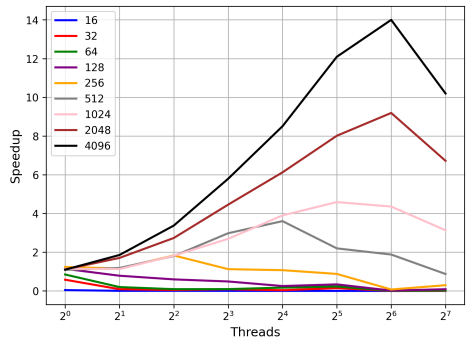
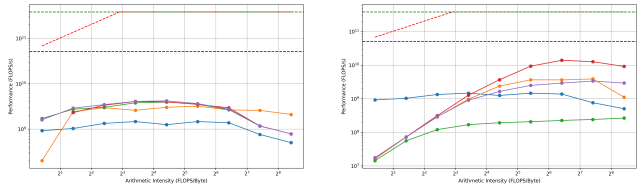


Fig. 4. Speedup vs. Number of Threads : Demonstrates Scalability for Different Matrix Sizes

variable at the start of the parallel region and combines the result at the end. This minimizes the synchronization overhead, proving very efficient for large sizes. Method 4 employs *simd* to vectorize execution. This approach shows high performance for small matrices but does not scale well for larger sizes due to inefficient handling of the shared variable.

Fig.I[(d) **matTransposeOMP**] illustrates that method 3 achieves the greatest speedup as matrix size increases, reaching up to 17.8x. This approach manually distributes the workload across threads, effectively reducing overhead and maximizing parallel efficiency. In contrast, method 4 exhibits a speedup between 1.3x and 3.5x by employing a blocking. This approach partitions the matrix into sub-matrices and assigns them statically to threads, enhancing cache locality by transforming non-contiguous memory access into localized operations, thereby reducing TLB misses and page faults.

Method 1 achieves moderate speedups, ranging from 0.4x to 5.1x. It statically distributes the workload evenly among threads, proving effective for balanced workloads but lacking adaptability to dynamic variations. On the other hand, method 2 achieves speedups of 1.2x to 3.5x by dynamically assigning sub-iterations to threads. This approach better handles imbalanced workloads but incurs additional overhead due to the dynamic task assignment process. Fig.3 further demonstrates that increasing the number of threads leads to an exponential increase in bandwidth for larger matrices, highlighting the scalability potential of these parallelization strategies. 4 illustrates the speedup achieved for a fixed matrix size as



(a) Implicit parallelization (b) Explicit parallelization

TABLE II
ROOFLINE MODEL FOR IMPLICIT AND EXPLICIT PARALLELIZATION. THE DOTTED LINES INDICATE THE PEAK COMPUTATIONAL PERFORMANCE, ROOFLINE, AND PEAK MEMORY BANDWIDTH

the number of threads increases. For smaller problem sizes, the speedup either plateaus or declines with increasing thread count, indicating suboptimal strong scaling. This behaviour is likely attributed to communication overhead or the problem size becoming insufficiently large to justify parallelization. The **4096 curve** demonstrates robust scaling initially, but the performance begins to degrade beyond 64 threads. This degradation is likely a result of thread-allocation overheads surpassing the computational gains, thereby limiting the efficiency of parallel execution.

The **Roofline Model**, as shown in Fig.II, provides a visual representation of the theoretical performance limits of the implicit and explicit implementations. The blue line with circles represents the performance of the sequential implementation. It starts at a lower performance level and increases with arithmetic intensity but does not reach the roofline, indicating limited memory or computational efficiency. All implicit and explicit implementations have improved performance as arithmetic intensity increases. However, after a certain peak, the performance decreases. None of the implementations reach the peaks or roofline, likely due to memory bandwidth limitations, cache inefficiencies or other system bottlenecks. Therefore the algorithm is memory-bound. The explicit implementations, shown in fig.II(b), performs poorly for small matrix sizes, but improve exponentially as the size increase.

VI. CONCLUSION

This report explored optimization techniques for matrix transpose employing techniques like implicit and explicit parallelism. Experimental results demonstrated that accessing matrices in row-major order significantly reduced cache misses.

Implicit techniques proved effective in significantly enhancing performance by optimizing memory access patterns, reducing latency and increasing arithmetic intensity. A balanced application of optimization flags i.e., O2+March, achieved better performance by maintaining the trade-off between memory access efficiency and computational load.

Explicit parallelization exhibited remarkable efficiency and scalability for large matrices. Careful management of thread allocation and workload balancing, improved speedup and effective bandwidth exponentially.

The Roofline Model highlighted that the matrix transposition is inherently memory-bound. Both implicit and explicit implementations demonstrated performance gains with increasing arithmetic intensity. Implicit implementations generally outperform the baseline across all arithmetic intensities, while explicit implementations struggled with small matrices but improve significantly as size and hence, arithmetic intensity increased. Theoretical peak performance, however, is never achieved due to bottlenecks such as memory bandwidth constraints and cache inefficiencies. These results emphasize the importance of combining robust efficient parallelization strategies with hardware-specific optimizations to maximize computational efficiency and push performance to another level.

REFERENCES

[1] Gianluca Frison, Dimitris Kouzoupis, Tommaso Sartor, Andrea Zanelli, and Moritz Diehl. 2018. BLASFEO: Basic Linear Algebra Subroutines for Embedded Optimization. ACM Trans. Math. Softw. 44, 4, Article 42 (December 2018), 30 pages. https://doi.org/10.1145/3210754

[2] I-Jui Sung, Juan Gómez-Luna, José María González-Linares, Nicolás Guil, and Wen-Mei W. Hwu. 2014. In-place transposition of rectangular matrices on accelerators. SIGPLAN Not. 49, 8 (August 2014), 207–218. https://doi.org/10.1145/2692916.2555266

[3] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. "Parallel Programming in OpenMP". Morgan Kaufmann Publishers, San Francisco, CA, 2001.

[4] Bucker, H. Martin and Rasch, Arno and Wolf, Andreas. "A class of OpenMP applications involving nested parallelism". In: Proceedings of the 2004 ACM/IEEE conference on Supercomputing. 2004.

[5] G. L. Anderson, "A stepwise approach to computing the multidimensional fast Fourier transform of large arrays", IEEE Transactions on Acoustics and Speech Signal Processing, vol. 28, no. 3, pp. 280-284, 1980.

[6] D. H. Bailey, "FFTs in external or hierarchical memory", Journal of Supercomputing, vol. 4, no. 1, pp. 23-35, 1990.

[7] Gustavson, F.G., Walker, D.W. (2014). Algorithms for In-Place Matrix Transposition. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds) Parallel Processing and Applied Mathematics. PPAM 2013.

[8] D. Chaver, M. Prieto, L. Pinuel and F. Tirado, "Parallel wavelet transform for large scale image processing", Proceedings 16th International Parallel and Distributed Processing Symposium, pp. 4-9, April 2002.

[9] Scott H. Brown, "Multiple linear regression analysis: a matrix approach with MATLAB", Alabama Journal of Mathematics, vol. 34, 2009.

[10] S. Chatterjee and S. Sen, "Cache-Efficient Matrix Transposition", Proceedings Sixth International Symposium on High-Performance Computer Architecture, pp. 195-205, January 2000.

[11] P. F. Windley, "Transposing matrices in a digital computer", The Computer Journal, vol. 2, no. 1, pp. 47-48, January 1959.

[12] Krisnamoorthy, Baumgartner, Cociorva, Chi-Chung Lam and Sadyappan, "Efficient parallel out-of-core matrix transposition," 2003 Proceedings IEEE International Conference on Cluster Computing, Hong Kong, China, 2003

[13] C. A. Parra, T. Yu, K. S. Yum, A. Garza and I. D. Scherson, "Recursive MaxSquare: Cache-friendly, Parallel, Scalable in situ Rectangular Matrix Transposition," 2020 International Conference on Computational Science and Computational Intelligence (CSCI), Las Vegas, NV, USA, 2020.

[14] Lee, Kyu-Yean, Cho, Seong-Jin, Yoon, Seung-Hyun, Jeon, Jae-Wook. (2015). Roofline measurement application. ACM IMCOM 2015 - Proceedings. 10.1145/2701126.2701150.

[15] N. Ding and S. Williams, "An Instruction Roofline Model for GPUs," 2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), Denver, CO, USA, 2019, pp. 7-18, doi: 10.1109/PMBS49563.2019.00007.