IntroPARCO 2024 H1

December 1, 2024

Dennis Alberti
238067
dennis.alberti@studenti.unitn.it

***Abstract*—This project explores the implementation and performance analysis of matrix transposition and symmetry using three approaches: sequential, implicit parallelization, and explicit parallelization with OpenMP. The primary objective is to compute the banchmark and analyze the performance, evaluating the efficiency and scalability of each implementation, focusing on matrix sizes ranging from 2^4 to 2^{12}.**
Performance metrics, including execution time, speedup, and efficiency, were analyzed to assess the benefits of parallelization. The results demonstrate significant improvements in execution time for both implicit and OpenMP implementations compared to the sequential baseline, particularly for larger matrices. Key challenges, such as optimizing memory access patterns and minimizing synchronization overhead, were addressed. This report also provides detailed system descriptions, reproducibility instructions and a comparison with theoretical peak performance. These findings contribute to understanding the trade-offs and advantages of various parallelization strategies in high-performance computing.

I. Introduction of the problem and importance

A. Background and Motivation

Efficient matrix operations are fundamental in numerous computational fields, such as scientific computing, machine learning, and graphics processing. Matrix transposition, a simple yet computationally intensive operation for large datasets, provides a foundational benchmark for evaluating algorithmic and system performance. As data sizes grow, optimizing such operations becomes increasingly critical, particularly in the context of parallel and distributed computing.

B. Objectives of the project

This project focuses on the matrix transposition and symmetry problem, leveraging both implicit and explicit parallelization techniques. The primary goals include:

1) Implementing and evaluating a sequential baseline for matrix transposition.
2) Enhancing performance through implicit parallelization techniques such as vectorization and memory access optimizations.
3) Applying explicit parallelization using OpenMP to further optimize execution.
4) Analyzing performance metrics such as execution time, speedup, and efficiency across various matrix sizes and thread counts.
5) Identifying and addressing challenges in parallelization, such as synchronization overhead and memory bandwidth limitations.

By benchmarking and comparing these approaches, this project aims to provide insights into the benefits and limitations of parallelization strategies for a fundamental computational problem.

Identify applicable funding agency here. If none, delete this.

II. State-of-the-Art

A. In-Place Matrix Transposition on GPUs

[**?**] Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks
[1] In-Place Matrix Transposition on GPUs

This paper addresses the challenge of in-place matrix transposition on GPUs, a key operation for many numerical algorithms. While in-place transposition is well-suited for GPUs due to their limited memory and high throughput, traditional CPU algorithms lack the required parallelism and locality. The authors propose a GPU-specific approach using tile-wise transpositions, with low-level optimizations and the identification of the best configurations. They present a heuristic for selecting tile sizes and use minimal padding to address performance limitations, achieving a significant throughput increase. Finally, the proposed method outperforms a recent implementation.

B. Gap Addressed by This Project

This project compare sequential, implicit and explicit parallelization techniques for matrix transposition using CPUs. By benchmarking these implementations across a range of matrix sizes and analyzing key performance metrics, the study aims to identify the trade-offs and best practices for optimizing memory-bound operations. Additionally, the project investigates how system-specific features, such as compiler flags and architecture, influence performance, providing a more comprehensive understanding of parallelization strategies.
CPUs are often more suitable for tasks with smaller data sets where the overhead of data transfer between the CPU and GPU memory can outweigh the parallelization benefits of the GPU and this is our case, a matrix n x n for n = $2^4 to 2^{12}$.

III. Contribution and Methodology

This project investigates the matrix transposition and symmetry check problem through three distinct implementations: sequential, implicit parallelization and explicit parallelization with OpenMP. The function *CheckSym* and *matTranspose* are coded in a way to reduce memory access through $j < i$, in this way only half of the matrix is analyzed for the symmetric control. For the transpose part $matrix[j][i] = matrix[i][j]$ access before the rows and then the columns giving an easier task to elaborate at the CPU. With this choices I will analyze the executions time of the different implementations and compare them in a table, then I analyze the speed up and efficiency of the OpenMP code, putting in relations the symmetry and translate functions for the different threads and the matrix dimension 2^{12}, to get the best out of it. At the end analyze the theoretical bandwidth of my system with the real bandwidth of the system through the different threads and with the matrix size 2^{12}, for the same reason as before. All of that is ran with the best with be best optimizations, obtained after many tests.

1) Baseline Sequential Implementation:

```
int isSym = 1;
for (int i = 0; i < N; i++) {
    for (int j = 0; j < i; j++) {
        if (matrix[i][j] != matrix[j][i]) {
        isSym = 0;
```

```
for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            transpose[j][i] = matrix[i][j];
```

2) Implicit Parallelization:
   - Adding only compiler-specific instructions like:

```
#pragma simd
```

   - This directive is a hint to the compiler to vectorized the loop (or the relevant code), which means it tries to use Single Instruction, Multiple Data (SIMD) instructions

```
#pragma unroll(4)
```

   - This directive is a hint to the compiler to unroll the loop by a factor of 4, meaning the loop is expanded so that four iterations are handled at once within the code.

3) Explicit Parallelization with OpenMP:
   - Development of a thread-level parallel solution using OpenMP, employing techniques such as work-sharing.

```
collapse(n)
```

   - The collapse(n) directive tells the compiler to merge n nested loops into a single loop for parallel execution.

```
shared(variables)
```

   - The shared clause defines variables that are shared among all threads in a parallel region. This means that all threads can access the same memory locations for those variables.

```
num_threads(variable)
```

   - The num_threads directive specifies the number of threads that should be used for a particular parallel region.

A. Comprehensive Performance Analysis:

Ideal Speedup: $S = num\_threads$
Speedup: $S = \frac{T_{sequential}}{T_{parallel}}$
Efficiency: $E = \frac{S(_p)}{num\_threads} * 100$

Some challenge encountered during the analysis are:
Synchronization Overhead solved by the experimentation with thread scheduling policies minimized thread contention. Compiler Limitations solved by iterative refinement of code structure ensured effective utilization of implicit parallelization. Memory Bottlenecks solved by align memory allocation to cache lines to avoid false sharing (multiple threads accessing different data on the same cache line). Most of them are solved in automatic by the correct choice of the $pragmas$ in relation of the code structure needs.

IV. Experiments and System Descriptions

A. Computing Platform and System Description

Processor: Intel(R) Core(TM) i7-8565U CPU 1.80Hz, 4 cores, 8 logical processors.
RAM: 8GB DDR4.
Operating System: Windows 11
Compiler: GCC 13.2.0, G++ 13.2.0 (MinGW Compiler), GCC 9.1.0 on the cluster. Libraries: $< omp.h >$ for OpenMP compilation and the standard C libraries for sequential and implicit implementation.

B. Experiments

The experiments are tested with N ranging from 2^4 to $2^{12}$to evaluate performance scalability and with benchmarked with varying thread counts (e.g. 2, 4, 8, 16, 32, 64.) for the OpenMP implementation.
To obtain the final result I first tested with different optimizations and flags. Then I found the best one for the serial, implicit and explicit parallelization and confront the different time taken from the functions and confront between each other. This pattern was done before in my personal system and than in the cluster. At the end the result saved and confronted are the one obtained form the cluster. The serial code is tested with 3 tipe of optimizations:

```
-O1, -O2, -O3
```

The implicit code is tested with many combination of many vectorization flags:

```
-O1, -O2, -O3, -ftree-vectorize,
-march=native -funroll-loops
-funroll-loops, -fopt-info-vec-all
```

The OpenMP code is tested with many more combinations:

```
-fopenmp
-O1, -O2, -O3, -ftree-vectorize,
-march=native -funroll-loops
-funroll-loops, -fopt-info-vec-all
```

Once found the best of each one we can start analyze it with the different matrix dimension from 2^4 to 2^{12}. Obtaining as result the final data capable to be confronted. All the results are printed in 4 different .csv file where I extact all the data and us them to create the tables and plots.

V. Results and Discussion

Execution Time:

- Table 1 and Table 2 shows the execution times (in seconds) for the sequential, implicit, and OpenMP implementations across different matrix sizes. illustrates the trend of execution times, highlighting the significant reduction achieved with parallel implementations, particularly for larger matrices.

Comparison with Theoretical Peak Performance Bandwidth:

- Table 3 presents the bandwidth task, the achieved performance was compared to the system's theoretical peak memory bandwidth. Figure 1 and 2 shows that while OpenMP approaches the peak for large matrices, certain bottlenecks limit performance for smaller sizes. In conclusion the the actual bandwidth is far from the theoretical one that is 38 GB/s.

Speedup and Efficiency:

- Table 4 presents the speedup S and efficiency E for the OpenMP implementation compared to the sequential baseline (when the thread is $\bar{1}$).
- Figure 3,4,5,6 plots speedup versus thread count for matrix sizes 2^{12}, demonstrating scalability and efficiency for both the functions.

A. Analysis and Interpretation

1) Sequential Baseline:
As expected, execution time increased quadratically with matrix size $(O(n^2))$ due to the nested loop structure. This provided a consistent baseline to measure performance improvements.

2) Implicit Parallelization:
Compiler optimizations such as -funroll-loops and vectorization yielded modest improvements over the sequential implementation from the moment we are analizing only half of the matrix and Compiler-dependent results showed diminishing returns for very large matrices, likely due to memory bandwidth saturation.

3) OpenMP Implementation:
OpenMP achieved substantial speedups, particularly for larger matrices and higher thread counts.
4) Speed up:
Speedup scaled almost linearly with thread count up to the number of physical cores. This evaluate the effectiveness and scalability of parallelization.
5) Efficiency: Efficiency decreased as the number of threads increased, highlighting synchronization overhead and memory contention as limiting factors.
6) Bottlenecks: For small matrices, the parallel overhead (e.g., thread creation and synchronization) outweighed the benefits of parallelization. Cache contention and non-optimal scheduling further impacted efficiency at high thread counts.
7) Impact of Compiler Flags: Aggressive optimization flags (e.g., -O3, -march=native) significantly enhanced the implicit implementation but had limited impact on the OpenMP version, suggesting that explicit parallelism was already well-optimized given that is not a good choice use -O3 optimization in most of the case. Disabling optimizations showed significant performance degradation, underlining their importance.
8) Scalability: While OpenMP scales well with increasing thread count, efficiency drops at higher thread levels due to synchronization and cache contention.

Size	checkSym (s)	matTranspose (s)	checkSym Imp (s)	matTranspose Imp (s)
16	0.000001	0.000001	0.000001	0.000001
32	0.000003	0.000002	0.000002	0.000001
64	0.000009	0.000008	0.000002	0.000003
128	0.000028	0.000049	0.000007	0.000012
256	0.000117	0.000229	0.000028	0.000070
512	0.000343	0.001023	0.000125	0.000436
1024	0.000978	0.002682	0.000740	0.002729
2048	0.007117	0.032369	0.006296	0.030761
4096	0.045685	0.147984	0.044984	0.147136

TABLE I
Execution times of symmetric check and matrix transpose operations with and without improvements for varying matrix sizes.

OpenMP			4096
Threads	Sym_OpenMP (s)	Tran_OpenMP (s)	
1	0.056769	0.156856	
2	0.041765	0.071844	
4	0.028108	0.041034	
8	0.016866	0.021050	
16	0.009878	0.012003	
32	0.005891	0.008522	
64	0.012687	0.015356	

TABLE II
Execution times for Sym_OpenMP and Tran_OpenMP across different thread counts.

Theoretical Peak Bandwidth:
For DDR4-2400:

$$\text{Bandwidth} = 2400\,\text{MT/s} \times 8\,\text{bytes} \times 2\,\text{channels}$$

$$= 38,400\,\text{MB/s} = 38.4\,\text{GB/s}$$

Effective Bandwidth:

$$Bandwidth = \frac{DataTransferred}{TimeTaken}$$

$$Bandwidth = \frac{4096 * 4096 * 4 * 2}{TimeTaken}$$

Bandwidth			**4096**
Threads	**Sym_OpenMP (GB/s)**	**Tran_OpenMP (GB/s)**	
1	2.3643	0.8557	
2	3.2137	1.8682	
4	4.7751	3.2709	
8	7.9581	6.3762	
16	13.5878	11.1818	
32	22.7851	15.7496	
64	10.5793	8.7404	

TABLE III
PERFORMANCE BANDWIDTH COMPARISON OF SYM_OPENMP AND TRAN_OPENMP FOR DIFFERENT THREAD COUNTS.

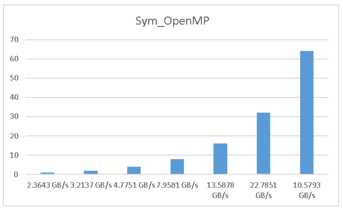

Fig. 1. Performance Bandwidth comparison of Sym_OpenMP

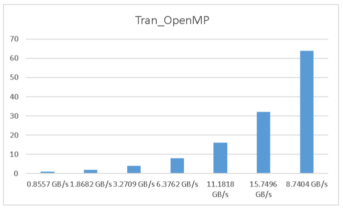

Fig. 2. Performance Bandwidth comparison of Tran_OpenMP

Threads	Sym Speedup	Sym Efficiency (%)	Tran Speedup	Tran Efficiency (%)
1	1.0109	101.0913%	1.0409	104.0990%
2	0.9953	49.7657%	1.0796	53.9803%
4	1.0942	27.3570%	1.1667	29.1681%
8	0.9361	11.7015%	1.1870	14.8380%
16	0.7065	4.4157%	1.5537	9.7111%
32	1.8394	5.7481%	1.6087	5.0274%
64	0.8929	1.3952%	2.3432	3.6612%

TABLE IV
SPEEDUP AND EFFICIENCY COMPARISON FOR SYM_OPENMP AND TRAN_OPENMP ACROSS DIFFERENT THREAD COUNTS.

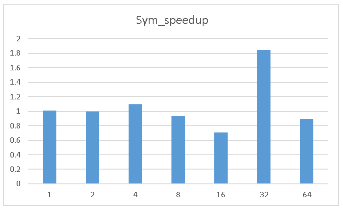

Fig. 3. Symmetry Speedup

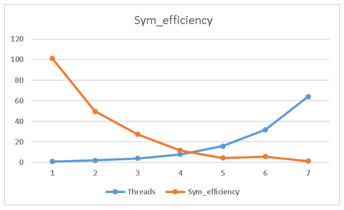

Fig. 4. Symmetry Efficiency

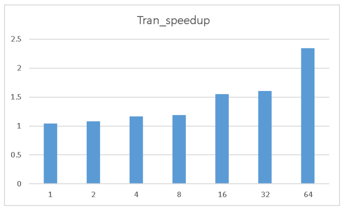

Fig. 5. Transposition Speedup

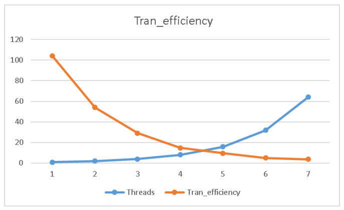

Fig. 6. Transposition Efficiency

B. Comparison with State-of-the-Art

Compared to known solutions in the literature, the results align with expectations, demonstrating that OpenMP provides competitive performance for memory-bound tasks like matrix transposition. Unlike some existing solutions, this project explicitly highlights the trade-offs between implicit and explicit parallelization and provides reproducible benchmarks. Also for larger matrices, optimizing memory access patterns (e.g., through blocking) is critical to achieving near-peak performance.

C. Key Insights

- Parallel Overhead: For small matrices, the overhead of parallelization can negate performance gains, making sequential or implicit approaches more suitable.
- Memory Bottlenecks: For larger matrices, optimizing memory access patterns (e.g., through blocking) is critical to achieving near-peak performance.
- Scalability: While OpenMP scales well with increasing thread count, efficiency drops at higher thread levels due to synchronization and cache contention.

VI. Useful links

Wikipedia Parallel Computing

University of Trento, course Parallel Computing

Intel CPU N°1

Intel CPU N°2

Intel CPU N°3

References

[1] J. Gómez-Luna, I.-J. Sung, L.-W. Chang, J. M. González-Linares, N. Guil, and W.-M. W. Hwu, "In-place matrix transposition on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 3, pp. 776–788, Mar 2016.

References

[1] J. Gómez-Luna, I.-J. Sung, L.-W. Chang, J. M. González-Linares, N. Guil, and W.-M. W. Hwu, "In-place matrix transposition on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 3, pp. 776–788, Mar 2016.