Introduction to Parallel Computing

Deliverable 1

Francesco Paissan
ID: 204297
francesco.paissan-1@studenti.unitn.it

***Abstract*—In this report, I present the implementation of two kernels: one for symmetry checking of squared matrices and one for matrix transposition. I analyze the impact of implicit and explicit parallelization on the proposed kernels. I analyze the trade-offs with respect to different optimization flags and number of threads. Finally, I compare the proposed kernel for matrix transposition with the peak performance of the system.**

The code associated to the deliverable is available at https://github.com/fpaissan/d1-parco.

I. Introduction

Matrix transposition and symmetry checks are common numerical kernels in computing linear algebra operations. They play a crucial role for many computational scenarios such as computer graphics and artifical intelligence. In this project, I will present the C implementations of the kernels and their optimizations.

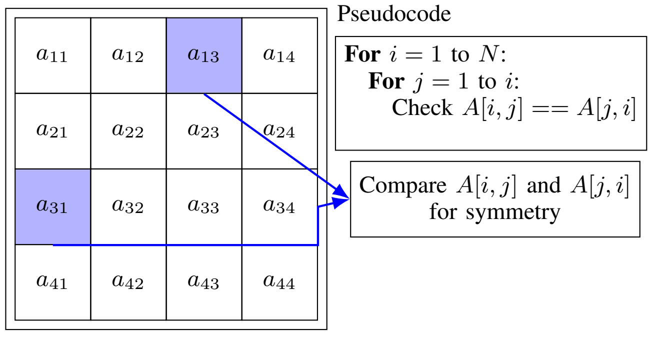
In Section **??**, I outline the algorithms to perform the matrix transposition and symmetry checks. In Section **??**, I present the experimental setup and computing environment. Finally, Section IV outlines the results obtained with an extensive benchmarking of the kernels.

II. Methods

In this Section, we present the implementation details for the kernels to check whether a matrix is symmetric (Section II-A) and to compute the matrix transpose (Section II-B).

A. Matrix symmetry check

The symmetry check kernel (`checkSym`) takes as input the matrix to check and its size. To reduce the number of comparisons to the minimum, we avoid checking the elements on the diagonal, as they are irrelevant to the successful execution of the kernel. The pseudo-code and visualization of the kernel implementation are presented in Figure 1. I acknowledge that this kernel is entirely memory-bound (as the matrix transpose one), as no arithmetic operations are performed, rendering the optimizations from the SIMD vectorization minimum.


a_{11} a_{12} a_{13} a_{14}
a_{21} a_{22} a_{23} a_{24}
a_{31} a_{32} a_{33} a_{34}
a_{41} a_{42} a_{43} a_{44}
Pseudocode
Compare $A[i,j]$ and $A[j,i]$ for symmetry

```
For $i = 1$ to $N$:
    For $j = 1$ to $i$:
        Check $A[i,j] == A[j,i]$
```

Fig. 1. `checkSym` routine visualization. Note the indices boundaries for the for loops are set to minimize wasteful comparisons.

Implicit parallelization. To optimize the kernel via implicit parallelization, I tested the effectiveness of three different GCC pragmas (`simd`, `ivdep`, `unroll`) independently and combined. These pragmas suggest that the compiler implements vectorization, ignores dependencies that might prevent vectorization and unrolls a specific loop, respectively. Additionally, I compiled the code with the `-ftree-vectorize` and `-funroll-loops` flags. Unluckily, I did not observe any improvement in wall-clock time using these strategies. To investigate the motivations, I experimented by changing the optimization levels. I observed a clear improvement in execution time between `O0` and `O1` (noticeable in the results, see Section IV-C). To dive deeper into the reason why the optimization was speeding up the kernel, I (i) checked the optimization logs obtained via `-fopt-info-optimized` and (ii) independently tested all flags employed in the `O1` optimization level[1] (see `scripts/test_O1_flags.sh`). Remarkably, the logs did not show any significant optimization performed on the kernel (e.g. loop unrolling or vectorization) despite the considerable speedup. Also, independently testing all flags did not prove to be an effective strategy, possibly because these are well-engineered to be working jointly. Finally, I performed manual unrolling of the loops, which proved effective in speeding up the kernel (more on the results in Section IV). Loop unrolling is effective as it reduces the branch overhead by effectively reducing the number of iterations needed to complete the loop. Additional benefits are better CPU pipeline utilization, easier instruction-level parallelism, and reduced function call overhead.

Explicit parallelization. The explicit parallelization of the `checkSym` kernel was performed using OpenMP. Specifically, I distributed the two nested loops on different threads using the following directive: `#pragma omp parallel for collapse(2)`. This directive makes OpenMP treat the two loops as a single loop, thus enabling a better distribution among threads and consequently improving load balancing and parallel execution. I note that since OpenMP does not allow

[1] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

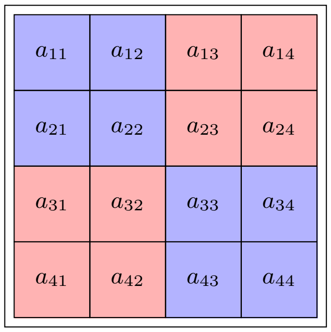


Fig. 2. Block-based matrix transposition. Blue blocks are diagonal blocks, and red blocks are off-diagonal. Each block (diagonal and not) is transposed. Off-diagonal blocks are swapped.

for threads to interrupt the kernel conditionally (e.g. via an if statement), I adapted the kernel to always check the entire matrix, updating a symmetry flag when two entries of the matrix do not match. It should be noted that to avoid issues such as race conditions and inconsistent results, the symmetry flag is updated within a `#pragma omp critical` block.

B. Matrix transpose

The sequential matrix transposition kernel, `matTranpose`, consists of two nested loops that iterate over all the matrix elements and swap the entries. This kernel has a computational complexity that scales quadratically with the size of the matrix. Additionally, it results in (i) suboptimal cache utilization and, thus, (ii) ineffective parallelization of the kernel.

Implicit parallelization. Similarly to what I observed for the `checkSym` kernel in Section II-A, the pragmas directives were not effective, but manually unrolling the loops proved effective also is this case (refer to `matTransposeImp`). I tried to improve the performance also via explicit data prefetching (`__builtin_prefetch`), without any noticeable performance improvement.

Explicit parallelization. For explicitly parallelizing this kernel, I tried various strategies ranging from tiling to parallelization with OpenMP and their combination. With tiling, visually depicted in Figure 2, the kernel is more optimized in terms of memory accesses, resulting in better cache management and utilization. Notably, without using threads, the kernel is more efficient than the sequential one. Using OpenMP, I employed the same directive used for the symmetry check (`#pragma omp parallel for collapse(2)`). However, I should note that in this case, there is no need to create a critical section for modifying variables, as the kernel is coded to avoid any dependency among threads. That is, in no circumstance will two threads be reading or writing to the same memory location. This significantly reduces the control overhead for thread management. Finally, I parallelized the tiled kernel outer loop via `#pragma omp parallel for`. This directive makes OpenMP process each tile in a different thread.

III. Experiments and System Description

In this Section, we describe the experimental setup (Sec. III-A) and the details of the computing systems (Sec. III-B).

A. Experimental design

The experimental setup was designed to give insights into the impact of the employed optimization strategies. To guarantee a fair and statistically significant comparison, I used repeated measurements of the wall-clock time; each kernel with each configuration is measured for 2000 runs. The wall-clock time is used with the `gettimeofday` function of the `sys/time.h` header. I automated the code compilation, run, and the results parsing to minimize human errors when analyzing the data. For each task (`checkSym` and `matTranspose`), I prepared a bash script to

1) compile the kernel with different optimization strategies (from `O0` to `O3`) and `-march=native`;
2) run the kernels on matrices of size ranging from 2^4 to 2^{12};
3) run the kernels with a different number of OpenMP threads (set via the `OMP_NUM_THREADS` environment variable);

Each run, the code outputs on the `stdout` stream the wall-clock time for every iteration, which is piped into a text file for post-processing. To simplify reproducibility, the logs are uploaded on GitHub in the `results` folder.

To ensure the measured wall-clock times are representative of the worst case scenario, we feed symmetric matrices to the kernels, thus executing all the nested iterations.

B. Platform and computing system description

Computing system. All the experiments are performed on a Dell Precision workstation. This has an `Intel(R) Xeon(R) Gold 5215` CPU working at a maximum clock frequency of 2.5 GHz. The memory configuration is 8x16GB DDR4 RDIMECC memory sticks, working at 2933MT/s and 754 GB, amounting to 128GB of RAM. The system employs two memory channels with a theoretical memory bandwidth (B) of

$$\begin{aligned} \text{B} &= \text{TransferRate} \times \text{ChannelWidth} \times \text{Channels} \\ &= 2933 \times 10^6 \times 8 \times 2\,\text{bytes/sec} = 46.928\text{GB}\,\text{s}^{-1}. \end{aligned} \tag{1}$$

The theoretical bandwidth in Equation 1 was compared with the effective bandwidth. The latter was estimated via a copy-paste kernel, presented in the `benchmarking` folder of the repository.

Toolchain and platform. The compiler employed in the experiments is `gcc (Ubuntu 11.4.0-1ubuntu1 22.04) 11.4.0`. I paired the official `gcc` documentation with [1], [2] for analyzing, understanding and comparing the effect of different optimization flags. The OpenMP version used in the experiments is 4.5, released in November 2015. The workstation employed in this experiments runs headless Ubuntu-22.05 and is not shared among other users. Therefore, I did not use any job scheduler such as Slurm or PBS.

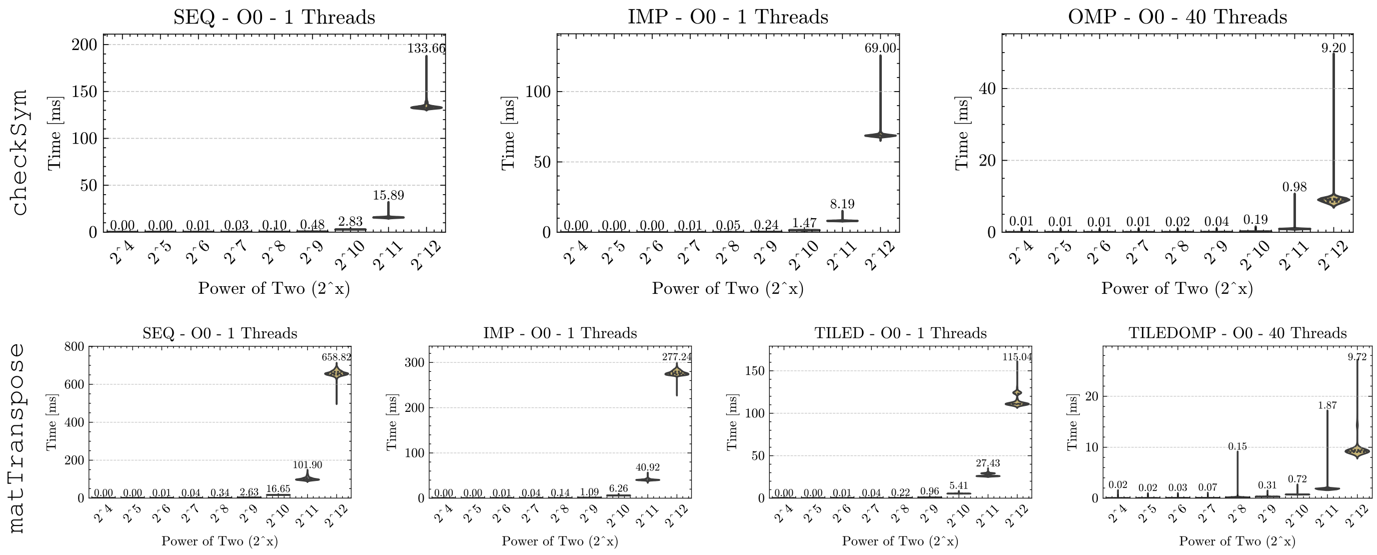


Fig. 3. Wall-clock time measurements for `checkSym` and `matTranspose` are reported in the first and second line, respectively. The violin plots show the distribution of the measurements. The average execution time is reported on top of each violin for easy comparison. `SEQ` refers to the sequential kernel, `IMP` to the kernel with implicit parallelization, `OMP` with the kernel using OpenMP, and `TILED` with the kernel that uses tiling.

IV. Results

For brevity, I report here the results required to address the main requirements of the homework. However, the results reported here are not exhaustively representative of the entire set of experiments I run. For this reason, I uploaded additional numerical results in the Appendix[2].

A. Comparison of sequential, implicit and explicit kernels

In this Section, I report the results needed to compare the wall-clock time of the sequential kernels with respect to the implicitly parallelized and explicitly parallelized kernels. These results are reported in Figure 3 for each matrix size (from 2^4 to 2^{12}) and the two tasks. At this stage, to marginalize the effect of implicit parallelization strategies that might go unnoticed, I report the numbers for the configuration without optimization flags (i.e. `O0`, see Section IV-C for the impact of optimization flags). For the OpenMP versions, I refer to the configuration with 40 threads. An analysis of the influence of the number of threads is reported in Section IV-B.

Symmetry check. For the `checkSym` kernel, I observe that the sequential kernel is the slowest. Doing implicit parallelization with loop unrolling, I observe a consistent speedup of $\sim 2\times$. The explicitly parallelized kernel, instead, achieves a speedup of $\sim 14.5\times$ with respect to the sequential kernel and of $\sim 7.5\times$ with respect to the implicitly optimized. Nonetheless, for small matrices (up to 2^7), there is no noticeable improvement. In fact, there is a performance degradation, possibly caused by the overhead of thread management and syncing, which is not amortized by a significant amount of computation in each thread.

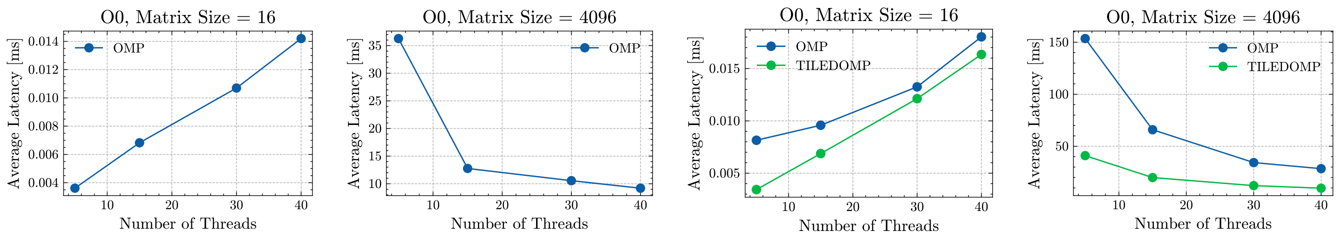
[2]https://github.com/fpaissan/d1-parco/tree/main/appendices

Matrix tranpose. For the matrix transpose kernel, the sequential kernel is the slowest, taking $658.81\,\mathrm{ms}$ for the 2^{12} matrices. With loop unrolling, I observe a speedup of $\sim 2.4\times$ for all matrix sizes. The tiled kernel is comparable to the implicitly parallelized one for matrices up to 2^8, then achieves a speedup of $\sim 2\times$ with respect to what we observe with implicit parallelization. Finally, parallelizing the tiling operations via OpenMP achieves the best results for matrices bigger than 2^8, similarly to what I observed for symmetry checking. Note that I am not reporting the results for OMP on the nested loops for brevity. The plots are presented in the online repository, in the `Appendix` folder. Numerically, parallelizing the two nested loops directly, I achieve a wall-clock time in-between the `TILED` and `TILEDOMP` configurations. Specifically, I observe a speed-up $\sim 5\times$ with respect to the `TILED` kernel, and a slow down of $\sim 3\times$ with respect to the `TILEDOMP` kernel.

In conclusion, I noticed that efficiently managing the memory acceses is crucial to achieve efficient kernels. Using parallelization generally helps when the compute required by the single threads is not negligible with repect to the necessary thread management. Also for parallelization, cache efficiency is key, as highlighted by the speedups between the OpenMP and tiled OpenMP results.

B. Impact of the Number of OpenMP Threads

Intutively, I expect that scaling the number of OpenMP threads increases performance. However, note that this can also lead to a significant overhead for thread creation, synchronization, and management for less demanding applications. To verify this hypothesis, I measured and reported the wall-clock time for the `checkSym` and `matTranspose` routines with respect to the number of threads (Figure 4). For small matrices, I observe that increasing the number of trends decreases the wall-clock time. On the contrary, for big matrices



checkSym matTranspose

Fig. 4. Wall-clock time measurements for `checkSym` and `matTranspose` are reported. The plots showcase the trend with respect to the number of OpenMP threads employed. `OMP` refers to the kernel using OpenMP, and `TILEDOMP` with the kernel that uses tiling and OpenMP.

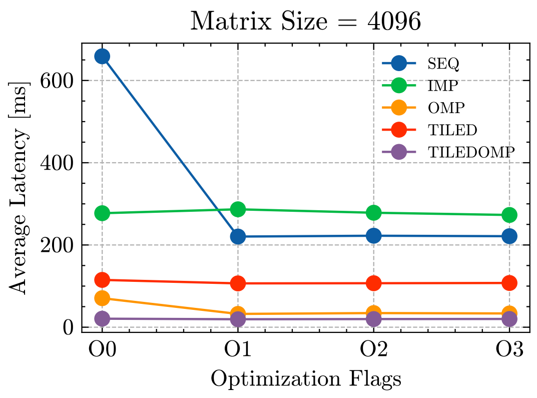


Fig. 5. Wall-clock time measurements for `checkSym` and `matTranspose` are reported. The plots showcase the trend with respect to the optimization flags.

$(2^{12} \times 2^{12})$ I observe a steady decrease in wall-clock time with increasing number of threads. Notably, this happens for symmetry checking and matrix transposition. In particular, for the tiled parallel transpose and also for the "vanilla" OpenMP implementation.

To summarize, I observed that setting the proper number of threads -or whether to use parallelization in general- requires testing with the target application and hardware to ensure that the implemented strategy provides the expected gains. Increasing the number of threads I observe diminishing returns. However, this effect might be mitigated, or not appear, for even bigger matrices.

C. Impact of Compiler Flags

To investigate the impact of compiler flags, I report in Figure 5 the wall-clock time for the kernels compiled with different optimization flags. The corresponding plots for different matrices and the `checkSym` kernel are reported in the Appendix. For both tasks, I observed an improvement in performance between O0 and O1 and a steady wall-clock time afterwards. Note that the logs show that only inline of the debugging functions was performed when O1 is activated, suggesting that some other optimization is going unnoticed. With O2 and O3 loop untolling and vectorization are enabled and activated (as verified in the logs). However, I did not observe any measurable improvement in wall-clock time.

D. Comparing with Peak Memory Bandwidth

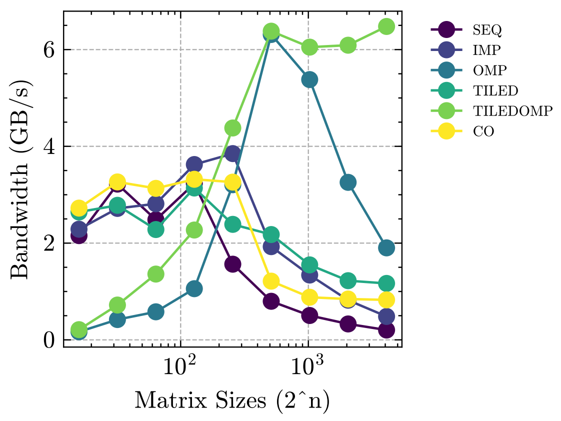


Fig. 6. Transfer efficiency for matrix transposition.

The peak theoretical memory bandwidth of is estimated $49.298\,\mathrm{GB\,s^{-1}}$. I measured the effective bandwidth to be $46.08\,\mathrm{GB\,s^{-1}}$, inline with the prediction. Both kernels should be memory bounded as no arithmetic operations are performed that might limit or bottleneck the transfer of data. To verify this, I compared the wall-clock time measured for the `matTranspose` routine with the peak performance of the system. Specifically, the amount of bytes moved by the matrix transposition is:

$$D_{\text{read}} + D_{\text{written}} = 8 \cdot N^2\text{B}. \tag{2}$$

Combining this with the wall-clock time to perform the kernel, with observe the trends highlighted in Figure 6. None of the implemented kernels is maximally exploiting the memory bandwidth, suggesting that further optimization is possible.

V. Conclusion

To conclude, in this report I showcased the implementation of a matrix transposition and a symmetry checking kernels. I optimized them via implicit and explicit parallelization, analyzing the trade-offs arising from the different implementations. As a result, I showed that the kernels exploiting OpenMP are highly optimized. Nonetheless, the parallelization is effective only for big matrices.

References

[1] W. Von Hagen, *The definitive guide to GCC*. Apress, 2011.
[2] M. T. Jones, "Optimization in gcc," *Linux journal*, vol. 2005, no. 131, p. 11, 2005.