Lab01 - Tensor Transposition for Fun and Profit

Team 14: Mark Cunningham, Daniel Yowell and Thi Tran

I. INTRODUCTION

In this lab, our primary objective was to explore the intricate interactions within the memory hierarchy, data layout, loop transformations, and performance optimization techniques. We are provided with an unoptimized baseline implementation. To approach this lab, we began by thoroughly understanding the provided baseline implementations and the accompanying testing frameworks. Our first step was to experiment with various iterations, implementing techniques such as a dispatcher based on input and output matrix sizes and strides, one-level tiling for cache optimization, a blocked variant calling a transposition kernel, integration of SIMD-based kernels, and loop restructuring to minimize memory traffic. Furthermore, we also explored shared memory parallelism by scaling the implementation from 1 to 3 threads.

II. RESULTS AND ANALYSIS

Baseline

The provided baseline is matrix transposition, which means it swaps the rows and columns of a given matrix.

```
void FUN_NAME( int m, int n,
               float *src,
               int rs_s, int cs_s,
               float *dst,
               int rs_d, int cs_d)
{
  for( int i = 0; i < m; ++i )
    for( int j = 0; j < n; ++j )
      {
        dst[ j*rs_d + i*cs_d ] =
          src[ i*rs_s + j*cs_s ];
      }
}
```

Fig. 1. Matrix transposition

The function uses two nested loops to iterate through the rows and columns of the input matrix. For each element at position (i, j) in the input matrix (src), it calculates the corresponding position in the output matrix (dst) using the provided row and column strides. The element is then copied from the source matrix to the destination matrix, effectively transposing the matrix.

Variant 1

In the first variant, we applied SIMD (Single Instruction, Multiple Data) which is a class of parallel processing archi-

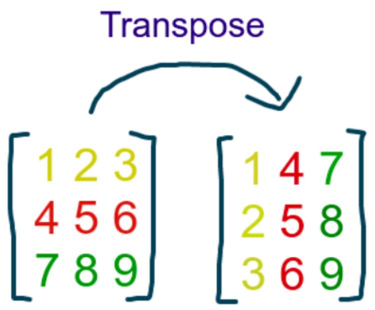

Fig. 2. Matrix transposition

tectures that allow multiple data elements to be processed in a single instruction to optimize matrix transposition.

We improved the code by dividing the task into smaller 8x8 blocks and using AVX instructions like unpacking, interleaving, shuffling, and permuting, which allow for efficient data movement and transposition operations. Our program also handles different combinations of row and column strides for both source and destination matrices, ensuring flexibility in data layout. Moreover, AVX instructions are designed for parallelism, making this code potentially faster on systems that support these instructions. All the specified test cases resulted in PASS, indicating the successful transposition of matrices according to the specified strides.

Variant 2

In variant 2, we applied different strategies. It allows to choose the transposition function based on input and output strides. Different transposition functions handle specific combinations of row and column strides. It dynamically selects the appropriate function at runtime, enabling flexibility for different memory layouts. While in variant 1, it does not use dynamic function selection based on strides. Instead, it directly checks for specific combinations of input and output strides and handles them within the transposeLargeMat function.

Variant 2 has thoroughly passed all tests for different matrix sizes and various combinations of input and output strides. It demonstrates consistent and correct behavior for transposing matrices under different conditions.

Variant 3 and Variant 4

In these variants, we also applied OpenMP for parallelization to achieve high-performance matrix transposition. OpenMP allows us to parallelize loops by splitting the work among multiple threads; we used 2 threads for variant 3 and 3 threads for variant 4. The outer loops that iterate over the blocks allow multiple blocks to be transposed concurrently on multi-core processors. These variants also passed all tests for different matrix sizes and various combinations of input and output strides. It demonstrates consistent and correct behavior for transposing matrices under different conditions.

Variant 5

In this variant, the loadMat64 and storeMat64 functions unroll the loops. We applied loop unrolling which is a compiler optimization technique where the loop body is duplicated multiple times to reduce the overhead of loop control and branching. So it can lead to more efficient SIMD code generation.

Variant 6

It combines multiple transpose functions optimized for different stride patterns and dynamically controls the number of threads used for parallel processing. We expected the code to provide a flexible and efficient matrix transposition implementation that can handle various matrix sizes and stride patterns. It utilizes different optimized functions to enhance performance when applicable.

Graph Analysis

The following graphs describe the performances of all variants which based on different matrix sizes and layouts.

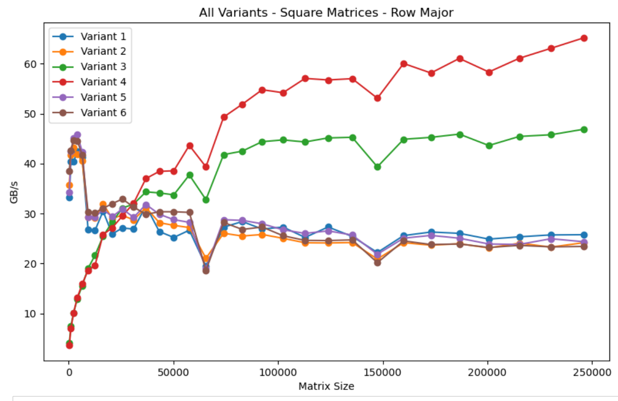

Fig. 3. Square Matrices - Row Major

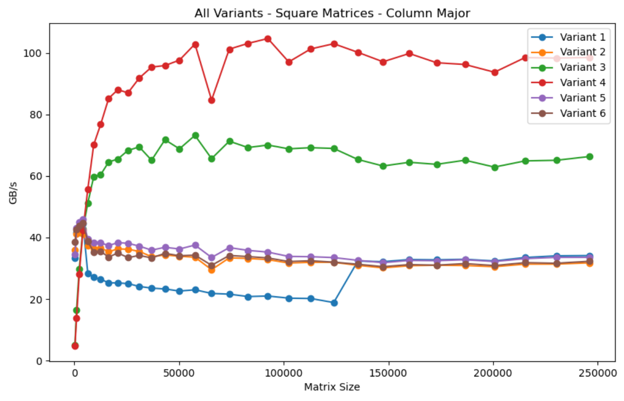

Fig. 4. Square Matrices - Column Major

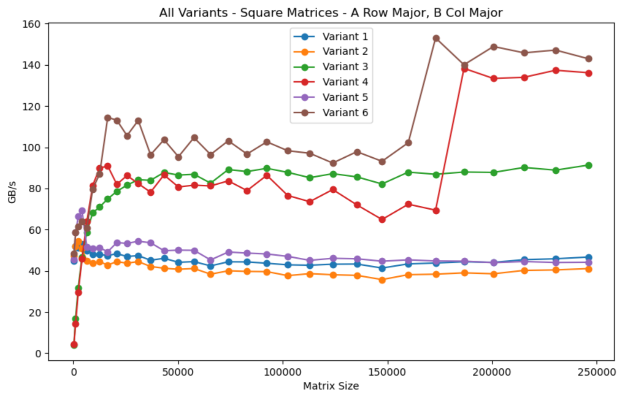

Fig. 5. Square Matrices - A Row Major, B Column Major

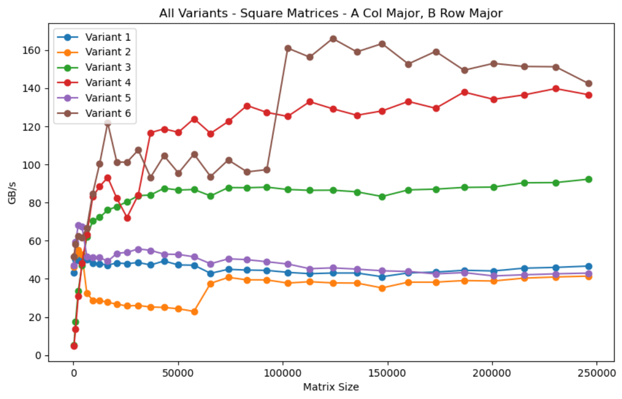

Fig. 6. Square Matrices - A Column Major, B Row Major

From the graphs, we see that variant 4 outperformed the other variants in all cases. Moreover, variant 4, with 3 threads, outperformed variant 3, with 2 threads, indicating that more threads led to better performance. Therefore, variant 4 is the most optimized solution, performing consistently well across different matrix sizes and layouts.

The variants 1, 2, and 5 have similar trends; they had consistent performance in most cases and were comparable to each other.

The variant 6 outperformed other variants in specific layout configurations (e.g., Square Matrix: A Row Major, B Column Major). Showed similar trends to variants 1, 2, and 5 in other layouts, indicating good overall performance but excelling in certain scenarios.

III. Conclusion

In summary, this lab was all about improving matrix transposition. We started by understanding the existing code and then made it better by trying out different techniques we learned in class. We created different variants of the program, each with different approach. By testing and comparing these variants, we learned what worked best and how the program could run faster. We also explored making the program work

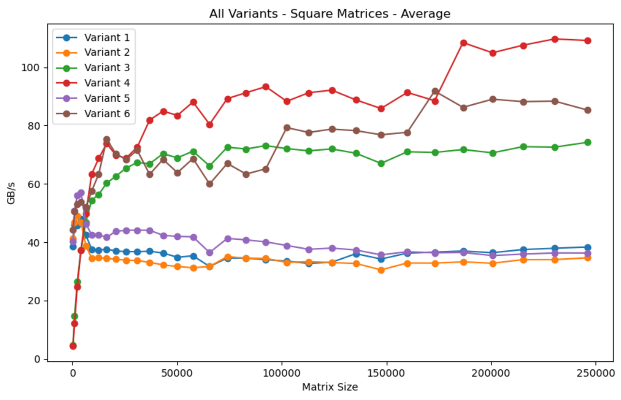

Fig. 7. Square Matrices - Average

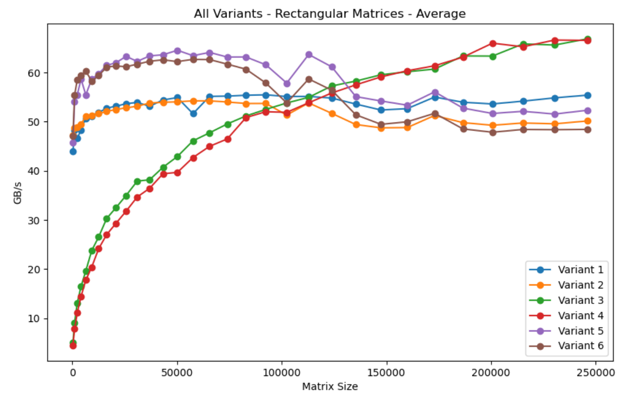

Fig. 8. Rectangular Matrices - Average

on multiple parts of the computer at the same time. This experience taught us a lot about how to make computer programs faster and more efficient, which will be really helpful in our future work.

References

[1] P. Pacheco and M. Malensek, An Introduction to Parallel Programming. Morgan Kaufmann, 2021.

[2] R. Veras, CS4473/5473: Parallel, Distributed, and Network Programming (PDN) Lectures, Fall 2022.