CSE 491 - Project 1 - Final Submission Writeup

Team: Paul, Jathin, Divyalakshmi

Overview and Goal of Project

The goal of this project was to implement an efficient algorithm to rotate an image (N×N bit matrix, where N is a multiple of 64) by 90° clockwise using bit manipulation techniques. The focus was on designing a solution and improving our previous solution (the beta submission) that ensures correctness and meets performance requirements while maintaining performance portability across different machine architectures.

Key Features of Our Design

Our project solution (the transpose method) was inspired by the RCR algorithm discussed in recitations. Our beta submission involved the transpose and flip algorithm but upon analysis and debugging, we implemented a series of optimization strategies (discussed below) to improve the performance of our algorithm, while maintaining the correctness aspect at all times.

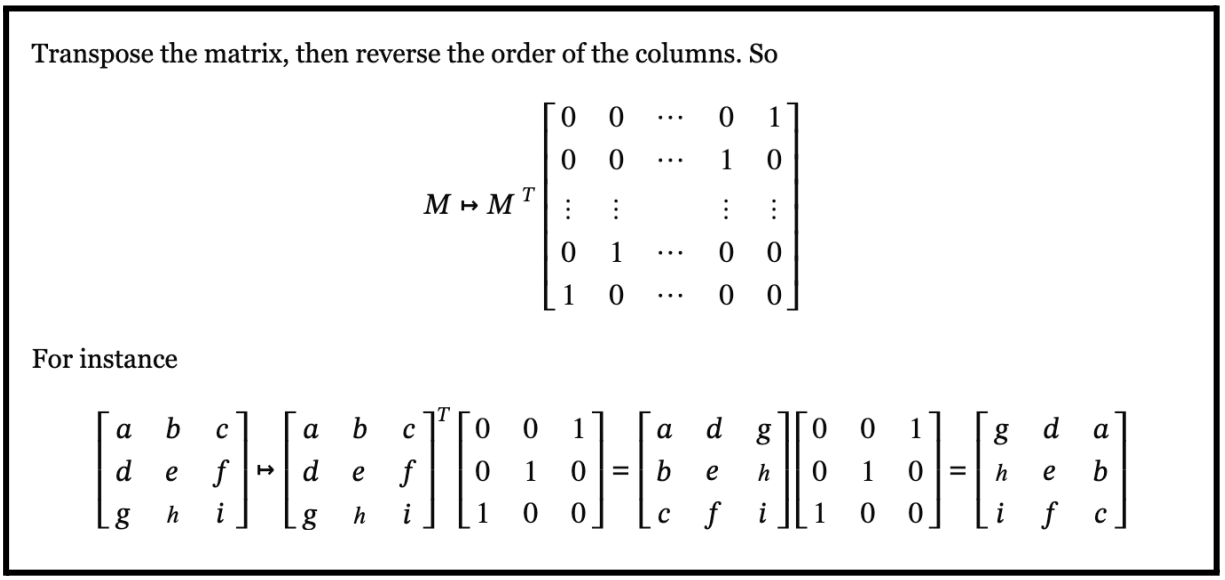

Figure 1: Diagram explaining the Transpose and Flip (reverse columns) algorithm used in our Beta solution. We improved the transpose and eliminated the flip portion to optimize the solution.

1. *Block-Based Transposition and Optimization* ***[Jathin, Divya, Paul]***:
   - The matrix is divided into 64×64 blocks, which are further broken into 8×8 tiles for efficient cache utilization and bitwise operations.
   - Instead of transposing and flipping entire rows separately, we optimized how data is loaded and stored by leveraging AVX2 intrinsics for bit manipulation and efficient swaps.
   - Diagonal blocks are transposed in place, while non-diagonal blocks are transposed and then swapped with the opposite block along the diagonal.

2. *Eliminated Separate Flipping Function After Transposition* ***[Jathin]***:
   - Initially, we performed matrix transposition first and then a separate flipping step to complete the 90° rotation.
   - In the final implementation, we combined transposition and flipping into a single operation by iterating only over one quadrant of the matrix.
   - Using an approach inspired by the bitwise quadrant rotation algorithm described in the course material, we directly mapped (i, j) to their new positions within the quadrant during transposition.
   - Instead of processing the entire matrix twice, we now efficiently swap and transpose in one pass.
   - This method was extended to 64×64 blocks, adapting the bitwise concept to work at the block level instead of individual bits.

3. *Optimized Transposition and Row Reversal* ***[Paul]***:
   - Our final implementation removes unnecessary function calls and directly operates on memory locations. Most importantly, we removed our `load_8x8()` and `store_8x8()` functions we had in our beta submission that were extremely costly.
   - We optimized our bitwise transposition function (bitwise_transpose()) to use AVX2 intrinsics for efficient bit-reversal operations.
   - The RCR-inspired row flipping strategy was refined using AVX2 vectorized operations to reduce overhead.

4. *Efficient Memory Access and Cache Utilization* ***[Paul, Divya]***:
   - We reduced unnecessary memory calculations by using precomputed index arrays and gathering operations for batch processing.
   - Vectorized processing of bitwise operations further reduced register dependencies and improved throughput.
   - Our use of row pointers helped minimize redundant calculations, further improving cache locality.

5. *AVX2 Intrinsics for Bit Manipulation* ***[Jathin, Divya, Paul]***:
   - Our final implementation replaces scalar bitwise manipulations with SIMD-based bitwise operations, making it significantly faster for large matrices.
   - The `reverse_bits_avx2()` function utilizes AVX2 shuffle operations to perform parallelized bit reversals, reducing instruction count and latency.
   - We optimized `transpose_and_swap_64x64_blocks()` by batch-processing multiple tiles in parallel.

6. *Eliminated Function Call Overhead and Redundant Steps **[Paul]**:*
    - Instead of using `load_8x8()` and `store_8x8()`, we now operate directly on memory, significantly reducing overhead.
    - The `memcpy()` operation was optimized, ensuring faster in-place modifications instead of redundant temporary allocations.

Performance and Correctness Results

Performance:

Our final submission reached **41 tiers on Telerun**, a significant improvement from our beta submission's performance of 26 tiers. The overall execution time dropped significantly because we used vectorized transposition and optimized bitwise operations.

Correctness:

We passed all **50 test cases** consistently without any deviation.

Challenges and Failed Methodologies

1. *Prefetching Did Not Work Well **[Divya]***
    - We initially tried hardware prefetching, but due to the way our bitwise transposition and swap operations are structured, prefetching did not yield any significant improvements and even slowed down the rotation even further. Also tried prefetching at regular iterations (every 4 iterations) to avoid polluting the L1 Cache, but didn't seem to work.

2. *Memory Alignment Attempts Were Unsuccessful **[Divya]***
    - We attempted to align memory for vectorized operations, but since our approach primarily deals with bitwise transformations at the byte level, alignment did not provide correct rotations.

3. *Using the Quadrant Algorithm Initially Failed for Odd N Values **[Jathin]***
    - While implementing the quadrant-based transposition and flipping in one step, we encountered failures when N was an odd value.
    - The algorithm worked well for even values but failed to maintain correctness when handling the central block in odd-sized matrices.
    - It took manual rotation analysis to debug this issue. We realized that certain subcases required special handling for the central block in odd-dimension matrices.
    - By refining our index mapping strategy and adding conditionals to correctly handle these edge cases, we were able to fix the correctness while maintaining performance along with correctness.

4. *RCR Algorithm **[Paul]***
    - We attempted the RCR algorithm in the beginning but chose to do the transpose algorithm. We thought about changing multiple times because some of the loading

and intrinsics shown during the recitations seemed more intuitive with RCR. However, we stuck with transpose and made it work.

5. *Compressed Sparse Rows (CSR) technique* ***[Divya]***
   - We also tried to implement the Compressed Sparse Rows (CSR) that was mentioned in class but then realized that using that technique would make our algorithm complete the necessary rotations in place.
   - This is because we would be using additional memory to convert and store the given bit image into a CSR representation and then convert it back to the matrix form for displaying the output.

Project Log:

#	Date	Start time	Duration	Description
1	01-17-25	2 pm	1 hr	Created a group chat for the team. Went over the project description together briefly.
2	01-18-25	5 pm	30 mins	Went over the project description together in depth. Established work and ethics guidelines. Divided the work for the team contract. Proposed regular meeting times every week to review progress.
3	01-20-25	7 pm	2 hrs	Worked together to draft and format the team contract. Submitted on Gradescope and Github.
4	01-21-25	2 pm	1 hr	Met online to discuss key takeaways from the first recitation and lectures. We each decided to approach the RCR algorithm using our own implementation. Made 2 branches and brainstormed ideas.
5	01-23-25	7 pm	3 hrs	Tried to implement RCR but no results. Passing correctness tests but timing out for all the tiers. The initial ersion worked only when N=64 and not for multiples of 64.
6	01-25-25	2 pm	4 hrs	Started working on the Transpose Algorithm. Did a pair programming session. Failed correctness but passed 27 tiers.
7	01-28-25	2 pm	1 hr	Worked on the Transpose Algorithm more. Decided to divide our strategy - Paul would try out the RCR method, Divya would look into the CSR technique, and Jathin and Divya would try to get the transpose algorithm to pass the correctness tests.
8	01-28-25	6 pm	1 hr	Fixed transpose algorithm to pass correctness for the first time. (19 tiers with correctness). Still trying to reach a higher tier with the RCR algorithm.

9	01-30-25	8 pm	1 hr	Tried running some of the tools mentioned in HW 2 to analyze performance (perf, valgrind, and cachegrind)
10	02-03-25	5 pm	5 hrs	Debugging transposing function for Beta Submission. Got the transpose-flip algorithm to work. Passed 26 tiers and all the correctness tests. Beta Submission is done.
11	02-04-25	8 pm	1 hr	Worked on writing down an outline for our Project 1 Beta write-up. Divided the tasks among teammates.
12	02-05-25	5 pm	2 hr	Finished drafting and proofreading the Beta write-up. Submitted on Gradescope.
13	02-07-25	6 pm	2 hr	Tried to use intrinsics and vectorization methods to speed up the performance from 26 to 29 tiers.
14	02-12-25	5 pm	4 hrs	Tried implementing prefetching methods over the some basic speed up achieved by using intrinsics in certain functions. We came to a conclusion that prefetching wouldn't work with our algorithm. It slowed us down to 11 tiers.
15	02-16-25	3 pm	1 hr	Did a pair programming session. Brainstormed other possible strategies (more vectorization, intrinsics, memory alignment, using the restrict keyword to avoid memory aliasing, loop-up table for reversal). Reached a version with 36 tiers but was failing correctness tests. Looked at the feedback from TA on Gradescope to try to transpose 64x64 instead of 8x8 blocks.
16	02-19-25	1 pm	1 hr	Code reached 30 tiers. Need to work on transposing 64x64 instead of 8x8 blocks and cache optimization.
17	02-23-25	8 am	6 hr	Implemented 64 x 64 transposing after multiple failed tries. Removed the load and store 8x8 functions and stored array of uint64_t. Speed up around 4 tiers. Optimized cache too. Passed all the correctness tests and reached 34 tiers on telerun.
18	02-24-25	9:30 am	2 hr	Gave another shot at implementing prefetching for the load and store functions and using that logic in our new transpose function. Also tried prefetching at regular iterations (every 4 iterations) to avoid polluting the L1 Cache, but didn't seem to work. Slowed us down to 32 tiers.
19	02-25-25	5 pm	4 hrs	Worked on eliminating the flipping function (speed up 5 tiers). Then, used intrinsics for reverse function. Passed 36 tiers first. Then reached 39 tiers.
20	02-26-25	12 am	2 hrs	Implemented intrinsics for transpose function which sped up 2 tiers. Final tier reached is 41. Passed all correctness tests.

21	02-26-25	6:30 pm	30 mins	Merged all our divergent branches to main to ensure we have all the work we did in the main branch. Clean the repository for the final submission.
22	02-27-25	8 pm	3 hrs	Started drafting and working on the Final write-up.
23	02-28-25	1:30 pm	4 hrs	Finished writing out, formatting, and proofreading the final report. Submitted to Gradescope and Github.

Acknowledgement of Help from Course Staff

The recitations and lectures were helpful and gave us a good idea of how we should be approaching the project. It also introduced us to the concepts of Bentley Rules, the RCR algorithm, Vectorization, Prefetching, utilizing intrinsics, and other optimization techniques which we experimented with. We'd like to thank the TA and instructor for their support and insightful suggestions on Gradescope for our Beta Submission as well as during recitations on strategies to adopt to optimize our algorithm further. We'd also like to thank our peers in the class who shared their findings on Piazza regarding Telerun set-up errors and editing Makefiles to show STDERR and STDOUT while running on Telerun. We also got to talk with some of our peers during recitations to get ideas for our final submission. We did not specifically reference any of our peers' beta submission codes that were made publicly available after the first deadline, although we appreciate this collaborative learning experience. Though we did not implement RCR, the idea of how RCR can be applied gave us a great idea for transposing and flipping, which we later optimized with the help of vectorization, intrinsics, and other means. We would once again like to thank the course staff for their timely assistance and guidance both on Piazza and during class times.

Citations

1. ChatGPT and other generative AI tools to help us with the C syntax, writing comments/docstrings, and drafting an outline for our final submission writeup.

2. Transpose and Flip Algorithm is based on this Stack Exchange page: https://math.stackexchange.com/questions/1676441/how-to-rotate-the-positions-of-a-matrix-by-90-degrees

3. Wikipedia pages on Matrix Transposition - https://en.wikipedia.org/wiki/In-place_matrix_transposition

4. CSE 491 Course Slides on Vectorization and SIMD Instructions - https://software-performance-engineering.github.io/spring25/calendar/Lecture05.pdf

5. CSE 491 Course Slides on Bentley Rules - https://software-performance-engineering.github.io/spring25/calendar/Lecture03.pdf

6. CSE 491 Recitation 1.2 Slides on Prefetching and Intrinsics - https://software-performance-engineering.github.io/spring25/calendar/Recitation%203%202025.pdf